World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:1, No:5, 2007

An Architecture for High Performance File System I/O

Mikuláš Patočka

Abstract**—** This paper presents an architecture of current filesystem implementations as well as our new filesystem SpadFS and operating system Spad with rewritten VFS layer targeted at high performance I/O applications. The paper presents microbenchmarks and real-world benchmarks of different filesystems on the same kernel as well as benchmarks of the same filesystem on different kernels — enabling the reader to make conclusion how much is the performance of various tasks affected by operating system and how much by physical layout of data on disk. The paper describes our novel features — most notably continuous allocation of directories and cross-file readahead — and shows their impact on performance.

Keywords**—** Filesystem, operating system, VFS, performance, readahead

I. Introduction

NOVEL approaches to filesystem design are needed because of growing disk sizes and transfer rates. When doing operations involving many files, performance in current filesystems is CPU-limited rather than disk-limited, thus filesystems achieve much lower transfer rate than disk's nominal speed. In this paper we present new filesystem — SpadFS [1] — as well as design of new kernel with improved filesystem interface that addresses these issues.

We benchmark our filesystem and other popular file systems — Ext2 [2], Ext3 [3], ReiserFS [4], Reiser4 [5], XFS [6], JFS [7] on two operating system kernels — Linux 2.6 and an experimental Spad [8].

We benchmark different filesystems on the same operating system kernel as well as the same filesystem on different operating systems. Thus we can compare the impact of physical data layout and kernel interface design on performance of various filesystem operations. SpadFS and Ext2 were benchmarked twice — once on Linux kernel and once on Spad kernel. The filesystems have the same layout of disk structures on both operating systems, but different code was driving it.

It will be shown that performance in certain workloads depends more on design of interface between kernel and filesystem driver (VFS) than on actual layout of data on disk. We emphasize the importance of research of the VFS architecture.

The paper has the following structure: in section 2 we describe general interface to filesystem implementations within operating system kernels and we present architecture of our new Spad kernel. In section 3 we present an overview of features of Spad filesystem. In section 4 we describe features of all benchmarked filesystem. In section 5 we describe the configuration of benchmarks and in section 6 we present the results and discuss them.

[1]This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838).

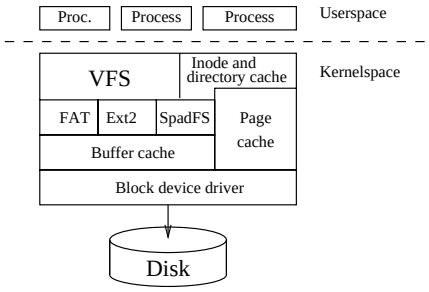

Fig. 1. Filesystem integration in operating systems

II. System architecture

A. Development of Filesystem Architecture

Old operating systems had only one filesystem. The filesystem code was mixed with other parts of kernel. When the need for more filesystems arose, the developers found that many tasks (for example caching) were identical to all different filesystems. Thus they split the filesystem code into filesystem-dependent part and filesystem-independent part [9] (often called VFS — virtual file system). The architecture of filesystem implementations in operating systems is shown in Figure 1.

VFS is specific to a given operating system and it is used by all filesystem implementations running on that system. Over the past years, more and more functionality has been moved from filesystem-dependent part to the VFS: Inodes are stored in the VFS in filesystem-independent format. Unified page cache was created in Linux 2.0 and FreeBSD 2 so that when taking page faults, data do not have to be copied from filesystem buffers to pages. Directory lookup caches were added to Linux 2.2 and FreeBSD 2 to save time when resolving long directory paths.

Filesystem-dependent part only fills inode and directory caches and maps logical blocks in a file to physical blocks on the disk; any further reading or writing is done by the VFS without any interaction with filesystem driver.

Thus, the performance of the VFS becomes even more important than the performance of the filesystem driver. We benchmarked filesystems with the same data layout (SpadFS [1], Ext2 [2]) on two operating systems' VFS implementations — Linux 2.6 and Spad, to show the impact of the VFS on filesystem performance.

B. Shortcomings of Linux VFS Design

Linux page cache first appeared in Linux 2.0 — it worked only for reading, writing was done via buffer cache. In Linux 2.4 page cache was redesigned to handle writes as well.

The most important problem of the page cache is that it works on block basis. Each page has linked list of buffer heads, each buffer head describes one block[1]. Thus, Linux kernel is extremely inefficient with small block sizes — the largest possible block size has to be used for decent performance.

Another inefficiency comes from the fact that most Linux VFS caches are write-through — they handle reads without interacting with the filesystem, but they call filesystem driver on writes.

- Page cache calls filesystem driver for each block when extending the file — the overhead of these calls causes high CPU consumption.
- Directory cache is completely write-through — all directory modifications are propagated into the filesystem driver immediately. The filesystem driver than places the modifications into buffer cache (thus they are not written to disk immediately), but passing the modifications between VFS, filesystem driver and buffer cache consumes a lot of CPU.
- Inode cache on Linux is write-back — it holds modifications to the inode and passes them to the filesystem driver asynchronously at later time. However, filesystem driver is called for each allocation or free of an inode.

C. Design of the SPAD VFS

SPAD VFS is completely redesigned filesystem interface where we tried to solve the above shortcomings.

Page cache doesn't contain list of blocks anywhere, it is completely extent-based. Each page has 4 values: start of valid data, end of valid data, start of dirty data, end of dirty data. The advantage is that no additional structures for blocks are required to manage data (thus filesystems on the top of the SPAD VFS can run with arbitrarily small block size without negative performance impact). This design has a slight disadvantage too — when the user needs to write block m in a page and later he writes block n, all blocks between m and n must be written (while Linux VFS could only write specified blocks m and n). The impact of this disadvantage is minimal — for disk with 50MB/s transfer speed, reading or writing 4kB page takes 82us, one seek usually takes 8ms.

SPAD VFS caches are write-back. When the user extends a file, the file is written to cache and no calls to filesystem are made. When the user wants to flush the file (with `sync` or `fsync` syscall), the whole file is allocated at once on disk. It reduces CPU load on the block allocator. Linux VFS would call filesystem allocator on block-by-block basis in this scenario.

SPAD VFS caches not only file writes, but also file and directory creating and deleting. When the user creates directory tree, the tree is created in the VFS cache and it is allocated later at once in a continuous block. Similarly when more files or directories are deleted, they are only marked for deleting in the cache and the real deallocation of blocks is done later.

To make delayed allocation work properly, the filesystem must provide upper bounds of number of allocated blocks for each operation (creating file, creating directory, extending file) — the VFS makes sure that it will never run out of space when delaying an allocation.

The effect of delayed allocation is reduction in CPU consumption. Another result is continuous allocation of files and directories. When extending a file, Linux VFS calls filesystem driver on block-by-block basis, the filesystem driver doesn't know file size in advance and can't optimally allocate file. On the SPAD VFS with delayed allocation, filesystem driver knows the size of the file when allocating it and can allocate continuous space for the file or all files in a directory.

III. Design of SPAD Filesystem

A. Directories

The SPAD filesystem uses modified Fagin's extendible hashing [10] for directory management. File name is hashed into 32-bit word and the word is used to index hash table.

We made several modifications to Fagin's method:

- In the hash word, the least significant bit is taken first and the most significant bit is taken last (because the least significant bit varies more in our implementation).
- We cannot allocate arbitrarily large hash table on the disk, so from certain size of hash table, we create a tree of tables.
- If there are more hash collisions than number of objects per page, the Fagin's method will fail. Such failure would be unacceptable in a filesystem. Thus, we added link pointers to pages — in case of extreme collisions, the pages will form a double-linked list of arbitrary length.

To optimize file access performance, the SPAD filesystem stores inodes directly in directory entries if these two conditions are met:

- There are no hard links to the inode.
- The inode has at most two extents.

B. File Allocation Information

File allocation information is stored in direct/indirect blocks, similar to the traditional Unix filesystem [11]. But instead of using array of blocks, we use array of extents — to conserve space and improve performance [12].

The allocation structure has 20 direct blocks (each containing one extent) and 10 indirect blocks of indirection levels from 1 to 10 — each containing pointers to other allocation structures.

C. Free Space Management

The SPAD filesystem uses extents to describe free space. Free space is stored in double linked list containing pairs (starting block, number of blocks). These double-linked lists are contained in allocation pages. Global index maps areas on disk to allocation pages. When the space in allocation page overflows, the page is split to two pages and index is updated. In the case of extreme fragmentation, double linked list is converted to a bitmap.

The process is described in detail in our technical report [13].

[1]The limitation that block size must not be larger than page size in Linux filesystems comes from this design.

IV. Filesystem Features

Now we describe features that other tested filesystems have. In table 1 you can see individual features of each tested configuration.

Fast recovery
: The possibility to quickly recover after a crash without scanning the whole filesystem. On most filesystems this is accomplished by using journal. SpadFS implements a different method — crash counts [14].

Indexed directories
: The ability of a filesystem to quickly find a file in a large directory without scanning the whole directory. In most filesystems BTrees or B+Trees are used. They may be used either directly with filename as a key (JFS [7]) or in such a way that filename is hashed and the hash value is used as a key to BTree (Ext3, ReiserFS [4], Reiser4, XFS [6]). SpadFS uses different method — a slightly modified variant of Fagin's extendible hashing [10] (see section 3.1).

Extents
: The filesystem stores file allocation information in triplets (*logical block*, *physical block*, *size*) rather than having list of all physical blocks forming a file. Extents significantly reduce the space needed for metadata of large files. The effect of extents on filesystem performance has been experimentally evaluated by Z. Zhang and K. Ghose [12].

Optimized small files
: The filesystem can efficiently store large amount of small files. ReiserFS and Reiser4 can store file content directly in the tree so that it consumes exactly the required amount of space, no padding to the block size is done. In the Spad VFS we took a different design course to achieve efficiency for small files — we allowed the filesystem to run with the block size of 512 bytes — thus small files are padded up to 512-byte blocks, unlike 4096-byte blocks most commonly used in Linux filesystems. When SpadFS runs on the top of Linux kernel, it is recommended to use 4096-byte blocks because Linux kernel page cache handles small blocks inefficiently.

Delayed allocation
: Most filesystems allocate space on disk when syscall `write` is called. This makes the syscall slow. An obvious improvement is to allocate space on disk when the cache is about to flush file blocks, not when the data were put to cache with `write` syscall — this is called delayed allocation. Delayed allocation has two advantages: it reduces time of cached writes significantly (as will be seen in benchmarks in next section) and it reduces file fragmentation, because continuous space can be allocated for file size exactly. XFS has delayed allocation for file data [6].
: Reiser4 has delayed manipulation of tree — tree entries are created in the time of the syscall but the nodes are allocated when the filesystem is flushing cache. Unfortunately Reiser4 gets this (and other) features by reimplementing a lot of work that is expected to be done in the VFS, which lead to refusal of Linux kernel developers to include it in standard kernel [16]
: In the Spad VFS we took this approach further and moved the delayed allocation framework from filesystem-dependent driver to the filesystem-independent VFS. Spad VFS can delay almost all operations — it has delayed allocation of file data, delayed adding of file's directory entry to a directory, delayed delete of files and delayed allocation of whole directories — these features are integrated in the Spad VFS and thus any filesystem on top of it automatically uses them. Delayed allocation of whole directories enable another novel feature: continuous allocation of a directory. When the filesystem is about to allocate space on disk, it sums the size of all new files in a directory and allocates continuous space for all of them. This improves efficiency of another feature — cross-file readahead.

Cross-file readahead
: The ability to read ahead across boundaries of individual files. Read ahead within files is implemented in all current operating system. We widened this principle, doing read ahead when sequentially reading directories containing many small files. As with delayed allocation, this feature is integral part of the Spad VFS and thus all filesystems on top of it benefit from it.
: Aggressive readahead has been already envisioned by researchers [17]. It is based on the fact that transfer speed of devices increases more and more while seek time remains roughly the same, thus the cost of readahead decreases. Our algorithm works in a simple way: if the file being read is smaller than 32KiB, we read 16KiB data past file end. If access to some other file hits the prefetched area, we read ahead following 64KiB data (experiments showed that reading more doesn't increase performance).
: Reiser4 has different kind of cross-file read ahead — it attempts to read the tree in advance if it detects sequential access to it. It does not attempt to read ahead file data though.

V. Benchmarking Methodology

With rising memory sizes, a lot of data is kept in cache. Thus, performance of cached operations is often even more important than performance of raw disk I/O speed — so both cached and uncached operations will be benchmarked.

For server applications running many processes, I/O throughput is not the only factor that matters. Another important feature is the CPU consumption — i.e. how much CPU time does the process performing filesystem I/O consume and how much CPU time does it leave to other processes. Both throughput and CPU consumption will be measured. Operating system tools such as "top" or "time" commands or `clock()` function are not precise enough to measure CPU consumption

TABLE I
FILESYSTEM FEATURES

	Fast recovery	Indexed directories	Extents	Optimized small files	Delayed allocation	Cross-file readahead
Linux/Ext2	No	No	No	No	No	No
Linux/Ext3	Yes	Optional	No	No	No	No
Linux/ReiserFS	Yes	Yes	No	Yes	No	No
Linux/Reiser4	Yes	Yes	Yes	Yes	Yes	Tree only
Linux/XFS	Yes	Yes	Yes	No	File content	No
Linux/JFS	Yes	Yes	Yes	No	No	No
Linux/SpadFS	Yes	Yes	Yes	No	No	No
Spad/SpadFS	Yes	Yes	Yes	Partial	Yes	Yes
Spad/Ext2	No	No	No	No	Yes	Yes

(they work by looking at process state in 10ms intervals and computing approximate CPU consumption) so, instead, we run two threads, the first thread performing I/O and the second thread spinning in a loop and incrementing single variable. From the number of iterations done by the second thread we can compute with almost CPU tick-level granularity how much time the first thread consumed. The second thread has priority lowered to minimum, so that it does not take processing power from the main thread when doing CPU-intensive workload. We disabled hyperthreading so that the measuring is accurate.

CPU consumption is measured only for raw uncached I/O benchmarks. For cached benchmarks, CPU consumption is obviously 100% and disk utilization is 0%, because the benchmark reads all the data from the cache. We do not report CPU consumption as percentage, but rather as seconds of time consumed. Percentage reporting is misleading — the filesystem that can finish the task faster reports higher percentage of CPU time consumed.

A. Hardware and Software Configuration

Benchmarks were performed on Pentium 4 processor running at 3GHz with 16kB L1 cache, 2MB L2 cache and 1GiB RAM. The computer has ASUS mainboard with ICH-7 chipset, dual DDR-2 533 memory modules, 160GB Western Digital Caviar SATA disk.

Linux benchmarks were done on SUSE Linux 10.0 with ReiserFS root partition with generic kernel 2.6.18.6. Spad benchmarks were done on the current development version with SpadFS root partition.

Tests were done on a clean 20GB partition located at the end of the disk, reformatted to a particular filesystem before each set of tests. The test partition was small compared to the size of the whole disk to minimize effects of transfer speed differences due to ZCAV [18] [19]. Transfer speed at the beginning of disk is 48.6MB/s, transfer speed at the end is 38.5MB/s.

All filesystems were created with default settings, except for the following two exceptions: Ext3 directory index was enabled (Ext3 supports indexed directories [20] [21] for fast lookup in large directories but they are not enabled by default) and SpadFS metadata checksums were disabled (all the other filesystems don't have metadata checksums, thus we did not want to give SpadFS unfair disadvantage).

B. Benchmarks Performed

We performed the following microbenchmarks showing the performance of particular filesystem functions: reading a small file from cache, writing a small file to cache, uncached read of a large file, uncached write of a large file, uncached rewrite of a large file (i.e. writing a file that is already allocated).

We performed some benchmarks consisting of real-workload: extracting of a tar archive, copying a directory tree, reading a directory tree, comparing two directory trees, deleting a directory tree.

The source code for benchmarking program can be seen in [15]. Cached benchmarks were performed repeatedly for 0.1 seconds. File read and write benchmarks were performed five times and they showed no more than 1% variance. Operations on the directory tree were performed five times and they showed no more than 5% variance.

VI. RESULTS

A. Cached Read

In figure 2 we can see the time required to read from the cache. Because reading from cache does not depend on filesystem code at all and depends only on the VFS code, we can see the only difference between Spad and Linux VFS's, not between actual filesystems. We can clearly see the performance degradation when 2MB CPU cache fills up. From this point on, the performance does not depend on operating system at all.

Reiser4 is the only Linux filesystem that does not line up with the others — the reason for this is that it replaces common Linux VFS functions with its own implementation [16].

B. Cached Write

Figure 3 presents similar benchmark — time to write to the cache. Most filesystems do allocation of blocks at this stage, so the time is dependent on actual filesystem implementation — it depends on the speed of its block allocator. On smaller file sizes Spad VFS clearly wins, being 6 times faster than fastest Linux filesystem, Ext2. It is because Spad does not do any block allocation at this stage, it does delayed block allocation when physically writing data.

As expected, performance of Spad/SpadFS and Spad/Ext2 is the same because Spad VFS does not call any filesystem-specific code in this scenario.

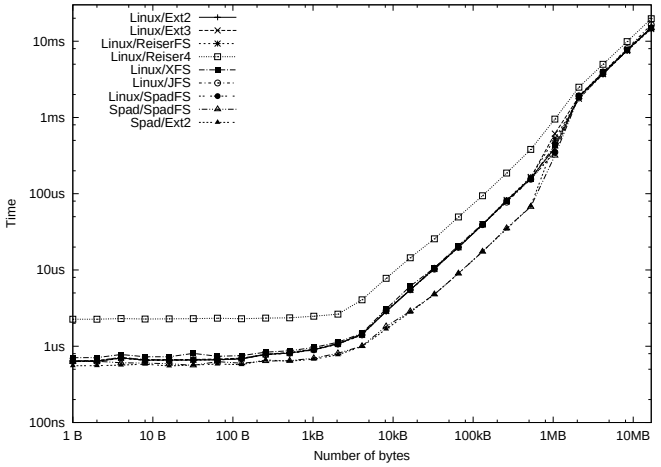

Fig. 2. Time to read a file from cache depending on file size

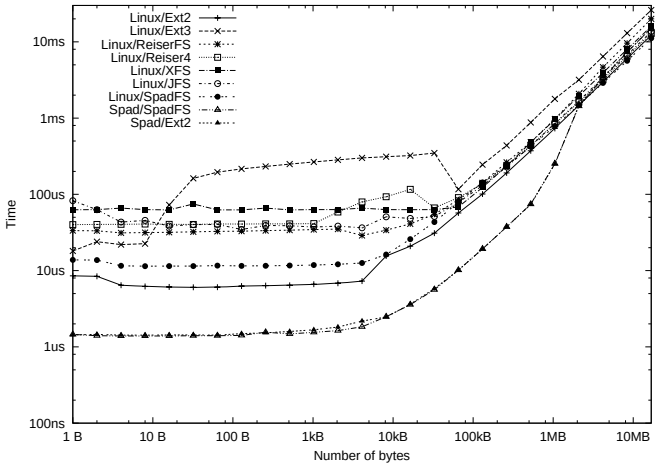

Fig. 3. Time to write a file to cache depending on file size

XFS has delayed allocation too, but it scores bad at this benchmark — one of the main reasons is that it is internally written for IRIX VFS and has a layer that translates Linux VFS calls to it [22].

Again, once the size of the L2 cache is exceeded, the benchmark does not depend on filesystem or operating system at all.

Next, we measured uncached operations (Fig. 4) — we created 8GiB file, rewrote the file without truncating it and read it. Ext2, Ext3 and ReiserFS clearly lose on rewrite operation — it is because they contain table of all allocated blocks instead of list of extents and they need to read a lot of data to determine locations of blocks that need to be rewritten. Spad VFS shows slightly lower write throughput compared to the same data layout on Linux VFS.

However, raw throughput is not the only important factor. Another issue is CPU consumption. CPU consumption of the same operations is showed in figure 5. We can see Ext3 clearly losing for writing and Reiser4 for reading. Spad VFS has the lowest consumption for reading. It can be seen that CPU consumption during reading depends more on the VFS layer

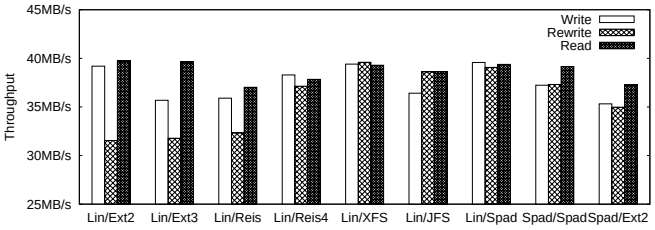

Fig. 4. RAW I/O throughput of filesystems

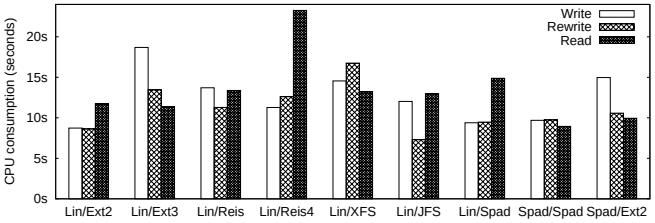

Fig. 5. CPU consumption of filesystems when creating/rewriting/reading 8GiB file

than on the filesystem itself — it is because in all modern operating systems reading is done by the kernel with very little interaction with the filesystem driver — filesystem driver only passes block numbers to the kernel.

C. Operations with Directory Tree

Next, some real-workload benchmarks were performed. We downloaded sources of OpenSolaris operating system — `on-src-20060814.tar.bz2` from address `http://www.opensolaris.org/`. This file was selected because it is one of the largest software archives on the Internet. We decompressed it, creating 365MB `.tar` file containing 6275 directories, and 34411 files and the following operations were done with it: we unpacked the file with `tar xf` command, we copied the unpacked directory tree with `cp -a` command, we read the whole directory tree with `grep -r` command, we compared both copies with `diff -r` command, finally we deleted the directory tree with `rm -rf` command. Cache was invalidated between all these tests. Times of these operations are shown in figure 6.

Ext2, despite being the oldest filesystem, performs exceptionally well, winning directory read and directory copy benchmarks. The reason for this is its simplicity — this benchmark consists of many small files (so extents do not create an advantage) and directories with few entries (so directory index will likely hurt the performance). The simplicity of Ext2 makes it consume small amount of CPU time and win.

SpadFS on Spad and Reiser4 come next, SpadFS very slightly winning copy and Reiser4 very slightly winning extract. SpadFS wins by order of magnitude on the delete test due to the fact that it concentrates metadata in a dedicated zone.

It can be seen that filesystems with cross-file readahead (both filesystems on Spad VFS and Reiser4) win by order of magnitude in the diff test. The diff test makes the disk head

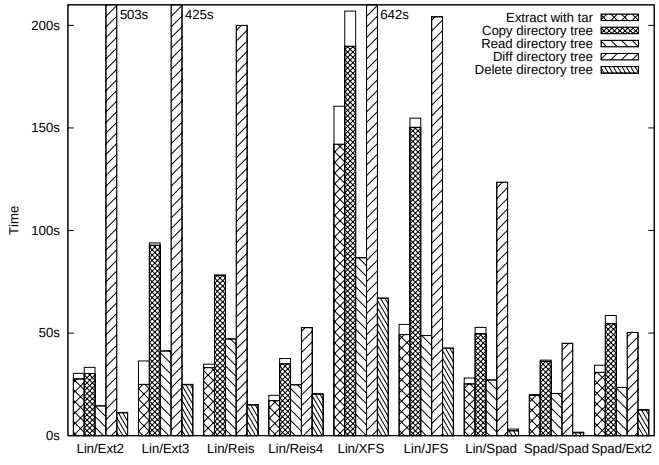

Fig. 6. Time to do operations with directory tree. Unfilled bar represents the time needed to flush cache.

seek vigorously from one tree to the other, filesystems without cross-file readahead would read one file from one tree (having a few kB size) and seek to the other while filesystems with cross-file readahead can read more files in advance.

VII. Conclusion

We benchmarked various Linux filesystems and our novel SpadFS filesystem and Spad kernel.

It was shown that SpadFS performs comparable to other filesystems and outperforms them at some occasions.

It was shown that Spad VFS is a promising design of a new filesystem interface architecture providing fastest cached operations, fastest creating and reading of directory with many small files, fastest extraction and copying of archive and smallest CPU consumption during uncached reading.

Journaling significantly slows down performance on directory tree manipulation benchmarks compared to filesystem without fast crash recovery (Ext2) and filesystem with crash counts [14] (SpadFS). The only journaled filesystem that performs well in this benchmark is Reiser4. On CPU consumption test, it can be seen that all journaled filesystems consume larger amount of CPU time — this is due to the implementation complexity of journaling code.

It can be seen that the novel feature ”cross-file readahead” can guarantee decent performance when accessing more directory trees simultaneously.

In this paper we have shown that for certain workloads the design of interface between kernel and filesystem driver affects performance more than physical layout of data on disk. The architecture of this interface layer thus becomes the new promising course of filesystem research.

References

[1] M. Patočka: SpadFS filesystem. http://artax.karlin.mff.cuni.cz/~mikulas/spadfs/
[2] R. Card, T. Y. Ts’o, S. Tweedie: Design and Implementation of the Second Extended Filesystem. Proceedings of the First Dutch International Symposium on Linux (1994)
[3] S. Tweedie: Journaling the Linux Ext2fs Filesystem. LinuxExpo (1998) 25–29
[4] F. Buchholz: The structure of the Reiser file system. http://www.cerias.purdue.edu/homes/florian/reiser/reiserfs.php
[5] N. Danilov: Reiser4, will double Linux FS performance. http://www.ussg.iu.edu/hypermail/linux/kernel/0211.0/0056.html
[6] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck: Scalability in the XFS File System. Proceedings of the USENIX Technical Conference (1996) 1–14
[7] S. Best, D. Kleikamp: JFS layout. http://mics.org.uk/usr/share/doc/packages/jfsutils/jfslayout.pdf
[8] M. Patočka: Prerelease of the Spad VFS layer. http://artax.karlin.mff.cuni.cz/~mikulas/spad-prerelease/
[9] S. R. Kleiman: Vnodes: An Architecture for Multiple File System Types in Sun UNIX. USENIX Summer Conference Proceedings (1986) 238–247
[10] R. Fagin: Extendible Hashing: A Fast Access Mechanism for Dynamic Files. ACM Transactions on Database Systems (1979) 315–344
[11] M. K. McKusick, W, N. Joy, S. J. Leffler, R. S. Fabry: A Fast File System for UNIX. ACM Transactions on Computer Systems 2, 3 (August 1984) 181–197
[12] Z. Zhang, K. Ghose: yFS: A Journaling File System Design for Handling Large Data Sets with Reduced Seeking. Proceedings of the 2nd USENIX Conference on File and Storage Technologies (2003) 59–72
[13] M. Patočka: File allocation information and block allocation subsystem. Technical report, Charles University in Prague
[14] M. Patočka: Using crash counts to maintain filesystem consistency. Proceedings of the CITSA (2006) vol. 3, 118–122
[15] M. Patočka: Filesystem benchmarking software. http://artax.karlin.mff.cuni.cz/~mikulas/bench/
[16] J. Andrews: Linux: Reiser4 and the Mainline Kernel. http://kerneltrap.org/node/5679
[17] A. E. Papathanasiou, M. L. Scott: Aggressive Prefetching: An Idea Whose Time Has Come. HotOS X, Tenth Workshop on Hot Topics in Operating Systems (2005)
[18] R. Van Meter: Observing the Effects of Multi-Zone Disks Information. Proceedings of the USENIX Annual Technical Conference (1997)
[19] P. Triantafillou, S. Christodoulakis, C. Georgiadis: A Comprehensive Analytical Performance Model for Disk-Storage Device Technologies. IEEE Transactions on Knowledge and Data Engineering, **14** (2002) no. 1, 140–155
[20] D. Phillips: A Directory Index for Ext2. Proceedings of the Annual Linux Showcase and Conference (2001)
[21] T. Y. Ts’o: Planned Extensions to the Linux Ext2/Ext3 Filesystem. Proceedings of the USENIX Annual Technical Conference (2002) 235–243
[22] J. Mostek, W. Earl, D. Koren: Porting the SGI XFS File System to Linux. The 6th Linux Kongress and the Linux Storage Management Workshop (1999)