2010 International Workshop on Storage Network Architecture and Parallel I/Os

hashFS: Applying Hashing to Optimize File Systems for Small File Reads

Paul Lensing
Paderborn Center for Parallel Comp.
University Paderborn
Paderborn, Germany
plensing@uni-paderborn.de

Dirk Meister
Paderborn Center for Parallel Comp.
University Paderborn
Paderborn, Germany
dmeister@uni-paderborn.de

André Brinkmann
Paderborn Center for Parallel Comp.
University Paderborn
Paderborn, Germany
brinkman@uni-paderborn.de

***Abstract*—Today's file systems typically need multiple disk accesses for a single read operation of a file. In the worst case, when none of the needed data is already in the cache, the metadata for each component of the file path has to be read in. Once the metadata of the file has been obtained, an additional disk access is needed to read the actual file data. For a target scenario consisting almost exclusively of reading small files, which is typical in many Web 2.0 scenarios, this behavior severely impacts read performance. In this paper, we propose a new file system approach, which computes the expected location of a file using a hash function on the file path. Additionally, file metadata is stored together with the actual file data. Together, these characteristics allow a file to be read in with only a single disk access. The introduced approach is implemented extending the ext2 file system and stays very compatible with the Posix semantics. The results show very good random read performance nearly independent of the organization and size of the file set or the available cache size. In contrast, the performance of standard file systems is very dependent on these parameters.**

I. Introduction

Today many different file systems exist for different purposes. While they include different kinds of optimizations, most aim to be "general purpose" file systems, supporting a wide range of application scenarios. These file systems treat small files very similar to gigabyte-sized database files. This general approach, however, has a severe impact on performance in certain scenarios.

The scenario considered in this paper is a workload for web applications serving small files, e.g. thumbnail images for high-traffic web servers. Real world example of such scenarios are given at different places and scales. Jason Sobel reports that Facebook accesses small profile pictures (5-20 KB) at a rate of more than 200k requests per second, so that each unnecessary disk seek has to be avoided[1]. Another example for such a scenario is "The Internet Archive", whose architecture has been described by Jaffe and Kirkpatrick [1]. The Internet Archive is built up from 2500 nodes and more than 6000 disks serving over a PB of data at a rate of 2.3 Gb/sec.

[1]A summary of Jason Sobel's talk can be found at http://perspectives.mvdirona.com/default,date,2008-07-02.aspx

Such workloads have very different properties from usual desktop or server workloads. Our main assumptions are that the files are small (4–20 KB) and that the file size distribution is nearly uniform. This is different from web traffic that is shown to have a heavy-tailed size distribution [2]. Additionally, we assume that the ratio between the available main memory and the disk capacity is small, which limits the amount of cache that can be used for inodes, directory entries and files.

We also assume that:

- accesses are almost exclusively reads.
- filenames are nearly randomly generated or calculated (e.g. based on the user id, a timestamp or a checksum of the contents) and have no inherent meaning and also do not constitute an opportunity for name-based locality. A directory-based locality as used by most general purpose file systems cannot be used with hashFS.
- the traffic is generated by a high number of concurrent users, limiting the ability to use temporal-locality.
- maintaining the last access time of a file are not important.

The design goal for the file system presented in this paper has been to optimize small file read-accesses, while still retaining a fully functional file system that supports features like large files and hard links without much performance loss.

While caching web server data in memory is an immense help, it is usually impossible to buffer everything due to the sheer amount of existing data. It has been shown that web requests usually follow a Zipf-like distribution [3], [2]. This means that, while the traffic is highly skewed, the hit-ratio of caches grows only logarithmically in the cache size, so that very large caches would be necessary to absorb most requests. A similar argument can be made based on results presented for the Internet Archive data [1].

Accordingly, it is important to optimize small file hard disk accesses. Accessing files with state-of-the-art file systems typically results in multiple seeks. At first the location of the metadata (inode) has to be located using directory information, which results – if not cached – in multiple seeks. After the metadata is read, another head movement is required to read the actual file data. In a scenario, where

978-0-7695-4025-2/10 $26.00 © 2010 IEEE
DOI 10.1109/SNAPI.2010.12

a huge number of small files needs to be accessed, these head movements can slow access times tremendously. Of course, the caches of the operating system (block cache, inode cache, dentry cache) can avoid some or most of these lookups. We will show, however, that the caches themselves are not sufficient for efficient small file reads.

Contributions of this paper:: In this paper, we show that extending an existing file system by a hashing approach for the file placement is able to significantly improve its read throughput for small files. Based on the ext2 file system, we use randomized hashing to calculate the (virtual) track of a file based on its name and path. Adding additional metadata for each (virtual) track, we are able to access most small files with a single head movement.

Our file system can be used without any changes on existing applications and infrastructure. Also existing file management tools (even file system checks) work out of the box, which significantly reduces administration overhead. In contrast to other approaches to improve small file read access, we do neither need a huge memory cache nor solid state disks to achieve a very good performance, independent from the total number of files.

After discussing related work in Section II, we present our ext2 extensions in Section III. The analysis of the approach is based on simulations and experiments. The results presented in Section IV show that our hashing approach improves file system performance by a factor of more than five in realistic environments without requiring any additional hardware.

II. File System Basics and Related Work

In this section, we discuss some file system design basics, as well as some general-purpose file systems and related approaches for small file read performance.

A. Hard Disk Geometry

In order to understand the arguments for the file system proposed in Section III, a basic understanding of hard disks themselves is necessary. A hard disk (also called magnetic disk drive) basically consists of one or more plates / disks containing concentric tracks, which are themselves subdivided into sectors. While there are several possible configurations, the most commonly used consists of two read/write heads for each plate, one hovering above the plate and another beneath it. All heads are locked together in an assembly of head arms, which means that they can only be moved together. The set of all tracks that can be accessed with a fixed head position is referred to as a cylinder.

When an I/O request needs to be served, the service time is the sum of the head positioning time (seek time), which is the time required to move the head from its current position to its target track, the rotation latency, which is the time until the desired sector has rotated under the head and the transfer time, which is needed to read or write the data [4].

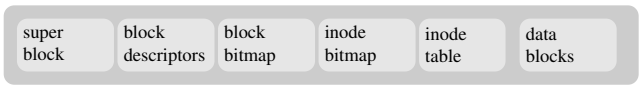

Figure 1. Structure of an Ext2 Block Group

B. Linux Virtual File System (VFS)

The virtual file system (VFS) is part of the Linux kernel and many other Unix operating systems and defines the basic conceptual interfaces between the kernel and the file system implementations. Programs can use generic system calls like open(), read() or write() for file system operations regardless of the underlying physical medium or file system. These are passed to the VFS where the appropriate method of the file system implementation is invoked.

C. ext2 File System

The second extended file system was one of the first file systems for Linux and is available as part of all Linux distributions [5]. The ext2 file system is divided into block groups.

Figure 1 shows the composition of an ext2 block group. Information stored in the super block includes, for example, the total number of inodes and blocks in the file system and the number of free inodes left in the file system. The block descriptors contain pointers to the inode and block bitmaps as well as the inode tables of all block groups. The block and inode bitmaps represent usage of data blocks / entries in the inode table. This enables each block to store its usage in a quickly accessible manner. The inode table stores the actual inodes of files stored in this block. Because a big file can span multiple block groups, it is possible that the inode that corresponds to the data blocks of a block group is stored in another block group.

An ext2 inode of a file contains all metadata information for this file as well as pointers to the data blocks allocated to the file. Since an ext2 inode has a fixed size of 128 bytes, it cannot store direct pointers to all data blocks of a file. Therefore, each inode only stores 12 direct data pointers to the first 12 data blocks of a file. If the file is bigger than 12 data blocks, the inode also contains an indirect pointer. This is a pointer to a data block that contains pointers to the data blocks allocated to the file. Because the block size is normally four kilobytes, an indirect block can store 1024 pointers. If this is still not enough, there exists a double and a triple indirect pointer.

D. Additional General Purpose File Systems

The third extended file system ext3 adds journaling modes to ext2, but stays otherwise completely compatible to ext2 [6]. The newest member of the ext family is ext4, which addresses several performance limits of ext3 and removes the 16 TB maximum filesystem and the 2 TB file size limit [7].

Most other filesystems use B+ trees or B* trees to manage their metadata. They either provide a global tree (ReiserFS [8], btrfs [9]), one tree for each directory (JFS [10]), or for each allocation group (XFS [11], [12]). ReiserFS and btrfs additionally provide an efficient packing for very small files.

E. Related Work

There are several existing approaches to improve small file performance. Ganger and Kasshoek proposed two improvements, which are both based on name-based locality. Firstly, they embed inodes into the directory entries of the parent directory and secondly, they co-locate related files on adjacent disk locations [13]. Because theses improvements are based on a name-based locality, they will fail in our scenario. Another approach, proposed as "Same Track File system (STFS)", optimizes small file writes by always storing file metadata and file contents in the same track [14]. The basic approach can be compared to the hashFS file system, which is proposed in this paper. Nevertheless, not reading the complete track, they have to use standard directory lookup procedures, which adds a significant overhead in scale-out environments.

Additional related research can be found in the context of web cache file systems, which are based on similar, but not identical assumptions. The web cache scenario differs from our scenario concerning

1) Data reliability: If a file is lost in a web cache scenario, it can be re-fetched from the original source.
2) Scalability: It is allowed for a web cache file system to replace old files using a cache replacement strategy, if the amount of files in a file system gets too large
3) Locality: In an highly distributed scale-out scenario, we cannot allow to bet on any kind of locality. Existing caching layers in form of content delivery networks (CDN) in front of the actual file delivery servers often remove all kind of locality.

Dámelo! is a web cache user-space file system, which does without directories and is based on user-defined group numbers, where the same group number implies locality [15]. It is focused on small files and large files are stored in a separate general purpose file system. Similar to our approach, they prefetch larger chunks of 16 KB to 256 KB.

Another reduced functional, specialized web cache file system is UCFS [16]. They hash the filepath to an in-memory table storing a mapping to a cluster of 32 KB to 256 KB size. Similar to the other web cache approach, related files are stored on adjacent disk locations by grouping them in a cluster. While this approach, similar to ours, eliminates disk accesses during lookup, the cluster table gets prohibitive large for larger file systems, requiring 96 MB RAM for each 4M files.

The Hummingbird file system is also a specialized file system for web caches, which co-locates related files in clusters and does its own memory cache management in the proxy to avoid unnecessary copies [17].

Independently from our work, Badam et al. presented the HashCache user-level storage system that also hashes to bins containing multiple files and used in its basic version no in-memory index structure [18]. This makes it suitable for servers with a large Disk/RAM ratio. However, HashCache also is highly specialized for caching and does not provide a full filesystem interface. The evaluation of HashCache compares it to web proxies. We think that a comparison with file systems and with an eye on the different file systems caches is insightful.

Jaffe and Kirkpatrick examined, how an SSD-based cache is able to reduce the IO load of the backend storage [1]. In contrast to them, we aim to improve the performance without additional hardware. DualFS is a journaling file system that separates metadata and data and stores them on different disks [19]. Especially, if metadata is stored on solid state disks with high random read performance, it might improve the overall performance at moderate costs, while compared to our report still requiring additional hardware.

III. Design and Implementation

The design and implementation of a new file system from scratch has not been within the scope of this paper. For this reason, the file system extensions are based on the ext2 file system, which has been the most practical candidate. Its implementation is relatively simple while still showing good benchmarking results. Furthermore, its main missing feature – journaling – does not impact the results of our evaluation. It should be noted that our design is not limited to ext2 and can be incorporated into other filesystems, too.

It is important to notice that our approach stays, with one exception, completely POSIX compliant. The only difference is that we are not able to evaluate the directory access rights of a file path during file operations, while we are still able to evaluate the access rights to the file itself. We assume that these missing directory access checks are not an issue. In our scenario, security is handled at the application layer, not at the file system. It will not update the last access times of the directory in the pathnames of the file.

The aim of a file system targeted to improve small file read performance is to decrease the number of necessary seeks. The main idea of the proposed file system is to perform only one seek to the metadata and data. Therefore, the proposed file system stores all necessary metadata together with the actual data, enabling one read operation to read both parts of information. To circumvent additional disk accesses needed to find the location of this information, a hash function on the complete path name is used in order to compute the expected location. Accordingly, the file system will be referred from now on as "hashFS".

A. File System Layout

The simplest and most accurate way to describe a location on a hard drive is by using a sector number. It is, however, not practical in our case. Hash collisions can occur and future hash results cannot be predicted. Accordingly, it is impossible to keep sectors needed by future writes free. For this reason, the track number has been chosen as the target for the hash function to identify the location of a file. This has multiple benefits:

1) A whole track can be read in at once without producing multiple head movements.
2) Multiple files can be written to the same track until the track runs out of free blocks.
3) It is possible to reserve space on a track for future writes.

Disk Geometry Information: If files are to be hashed to track numbers, some information about the disk geometry has to be known: At least the total number of tracks and their start and end sectors are necessary to identify a target range for the hash function. This information can be extracted using tools like the "Disk Geometry Analyzer (DIG)" [20]. During this paper, we work with "virtual tracks", where we assume that each track has the same number of sectors.

Block Allocation: Allocating blocks for a file on the computed track is not as trivial as it appears at first glance. It might not be possible to store all blocks of the file on the specified track because the size of each track is limited. A one megabyte file already spans up to four tracks on today's disks. Furthermore, if space is reserved on a track for future writes, then the number of sectors on that track which can be assigned to a single file is further limited. At the same time, the file system approach can only promise performance gains if a file is completely stored on its hashed track.

The described design conflict – reserving sectors on a track for later use and at the same time trying to store files completely on their hashed track – can only be solved with reference to the expected workload. The target scenario consists almost exclusively of reads to small files. Therefore, the important point is to succeed in storing small files on their respective tracks, while larger files can be handled differently. The size of a small file, however, is not explicitly defined by the scenario. It depends on the particular application case. Accordingly, the size of files which are considered to be "small files" should be configurable.

The following block allocation strategy, which we will call "Free Percentage Algorithm", is the result of these considerations: The first x blocks of a file are allocated on the hashed track, where x defines the maximum file size for which the file system is optimized. If a file is larger than x blocks, then the remaining blocks are only allocated on the same track if after the allocation the remaining space on the track would be sufficient for future writes. The percentage of a track that is reserved in this way depends on the file system load. At the beginning, 50% of each track will be reserved, and, as the hard disk fills up, the reserved space will shrink reflecting the diminishing total free space. It is, of course, still possible that a track runs out of free space. In this case, all blocks would have to be allocated on a different track.

Tracks that are filled with standard ext2 metadata, as, for example, copies of the super block or group descriptors, are an additional issue. Using default ext2 settings 1.63% of all tracks on the disks used are completely filled with ext2 metadata, additional 0.26% are partially filled. Files hashed to these tracks cannot be stored there, which results in a decrease of performance. However, since the allocation of this ext2 metadata is done statically during the formatting, it is easy to calculate these tracks and remove them from the disk geometry information. As a result no files are hashed to these tracks.

Track Metadata: Metadata and data for each file have to be stored on the same track to optimize read performance. We store this metadata in a data structure called "track inode" at the beginning of each track. Every file has a track inode on its hashed track. The normal ext2 inodes are used as normal to support other file system operations and to handle large files.

A track inode is not a simple copy of a normal ext2 inode, because not all information contained in an ext2 inode is needed. Even a few unnecessary bytes in a track inode will have a severe impact in our scenario. The track inode stores the inode number, the file size, the security attributes and direct pointers to the first x blocks of the file and the hash value for the file's path.

Another hash of the path is necessary to identify the correct track inode for a file out of all track inodes on the same track. It is not possible to store the pathname directly for identification purposes, because it can be arbitrarily large. To rely on hashing for pathname comparison creates the possibility for hash collision problems. The hash length must be chosen so that a collision of the track hash and of the name has is nearly impossible, e.g. by using a combined hash length of 96 bits on a 10 TB disk written full with 1 KB files results in probability of $6.3e^{-12}$ of a data loss due to hash collisions (birthday paradox).

B. Lookup Operations

We will now describe the pathname lookup operation of hashFS. In Linux, the default lookup strategy is to read in the metadata of every component of the file path iteratively, until the meta data of the actual file has finally been read. This is necessary, as the location of the metadata of a path component is stored in the previous path component, which corresponds to the directory enclosing the current path component. The pathname lookup strategy used by hashFS, in contrast, simply hashes the complete pathname to a track and reads the complete track. If the track inode corresponding

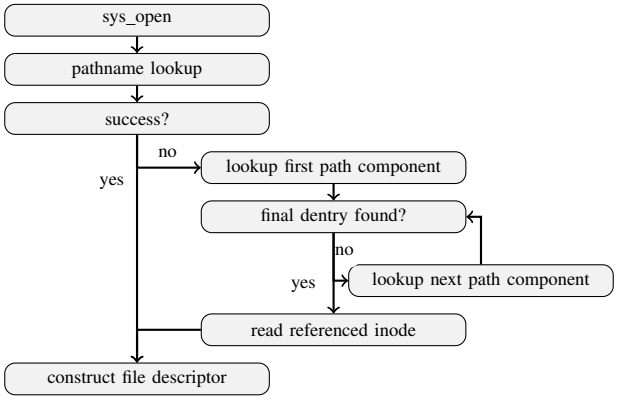

Figure 2. Lookup Strategy with Pathname Lookup Approach

to the path is found, the real on-disk inode is not read separately. Because the lookup strategy is implemented in the Virtual File System (VFS) layer, it has been expanded to allow for an alternative, pathname based lookup approach by the underlying file systems.

The general logic of a pathname lookup operation is presented in Figure 2. If the pathname lookup is successful, the whole iterative lookup process for each path component can be completely bypassed and it therefore requires exactly one read operation.

Similar to the original strategy, a dentry data structure (a cached, in-memory representation of a directory entry) is created for the file. However, because the path components are not processed independently, the hierarchical information normally available is missing. This is solved by interpreting the pathname starting from the mountpoint as a filename. So the dentry of the mountpoint is used as parent directory. Therefore, the dentry cache works as usual.

It is worth noticing that other POSIX file operations (like renames of files and directories, links) as well as larger files are still supported by HashFS using the normal ext2 metadata. Files that have not been stored at the hashed location will be found using the default ext2 lookup.

IV. Evaluation

This section presents simulation results as well as the experimental evaluation of hashFS. Inside the simulation part of this section, we discuss fundamental properties of the hashing approach, e.g. the number of hash conflicts for certain utilization degrees. The behavior of the properties is much more difficult to evaluate in real experiments and simulation offers an opportunity to evaluate many different settings.

The results for our hashFS implementation are described in the experimental part of this section.

Table I
Development of Track Inode Errors for Different Number of Files

M Files	Disk Util.	Average Track Inode Errors	Per Mille Values Average	Conf. Interval
1-12	<64.5%	0.0	0‰	[0.0, 0.0]
13	69.6%	0.0	0‰	[0.0, 0.0]
14	74.7%	1.9	<0.001‰	[0.0, 0.0]
15	79.8%	25.7	0.002‰	[0.002, 0.002]
16	84.9%	277.9	0.017‰	[0.017, 0.018]
17	90.0%	1946.6	0.115‰	[0.113, 0.116]
18	95.1%	10170.0	0.565‰	[0.561, 0.569]

A. Simulation

Prior to the implementation of the file system, a simulation tool was used to analyze different hashing properties. The simulation tool initially reserves the same blocks on the virtual disk, which are reserved for ext2 metadata during file system creation. Additionally, it reserves the track inode block for each track and simulates block allocation accurately. It does not simulate normal ext2 directory creation, however, and thus fails to allocate the data blocks of each directory. As a result, the observed disk utilization is slightly below the disk utilization that would occur in reality. Nevertheless, simulations allow examining allocation problems for a high number of possible file sets, which would be impractical using only the actual file system.

The simulation uses the geometry of a WDC WD800BB-00CCB0 hard disk, extracted using the DIG track boundary detection tool. This 80 GB hard disk has $225,390$ tracks, whose size varies between 500 and 765 sectors per track.

For all simulations, a block size of 4 KB and a configured maximum size of four blocks for "small" files is used. Using that configuration, 113 track inodes can be stored in a single track inode block. Observed results are given as average values and 95% confidence intervals of 30 runs.

Taking into account the modified geometry information, two possible causes for allocation problems remain, which might slow down the hashing file system:

Track Inode Miss:
: No free track inode remains in the track inode, where the file has been hashed to.

Data block Miss:
: Not enough blocks remain at the hashed track to allow the allocation of the minimal configured amount of data blocks.

The first problem results in a failure of the pathname lookup approach for the associated file. The resulting normal lookup operation can cause one disk access for each path component. Compared to that, the second problem incurs a lesser performance decrease: Because the data location is still obtained from the read in track inode, only a single additional disk access is needed to read the corresponding data blocks.

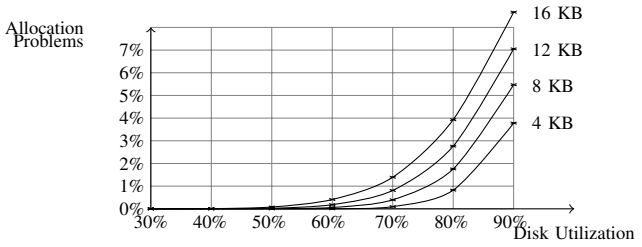

Figure 3. Development of Allocation Problems for Different File Sizes

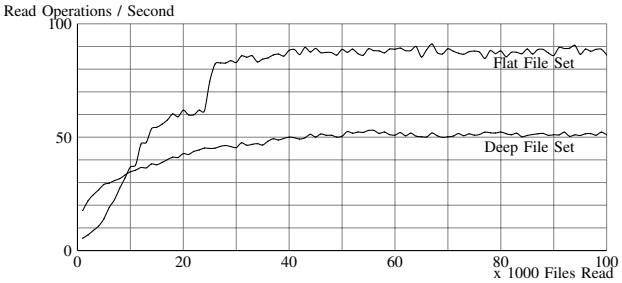

Figure 4. Ext2 Performance starting with a Cold Cache

Relation of Track Inode Misses to the Number of Files: At first, we examine track inode misses. These misses solely depend on the number of files hashed to a track, and therefore the total number of file allocated in the file system. Because the size of a track inode is constant, the actual size of the allocated files is non-significant. The file size is fixed at four kilobytes for the first set of simulations. All files are in the same directory, because the directory structure makes no difference for these simulations.

Table I shows the results of the simulation for different numbers of files. Because the observed number of track inode misses was too small to be expressed as a percentage value compared to the total number of files, they are expressed as a per mille value.

The percentage of files for which no track inode could be generated is less than 0.057% for 18 million files, which corresponds to a disk utilization of 95%, which is also the worst case in our setting. If the average file size would be 8 KB, less than 10 million files could be stored on the disk. As can be seen in the table, no track inode misses are expected in this case.

Simulations without removing the ext2 metadata tracks from the disk geometry show a 1.63% higher track allocation miss rate. This shows how important it is to handle these metadata collisions separately.

Relation of Allocation Problems to the Size of Files: The following simulations examine the impact of differing file sizes on the failure rate on the allocation of data blocks on the hashed track. File sets are generated in the same manner as for the previous simulation runs, differing only in the size of the allocated files. Because each file occupies a multiple of the block size, we have examined the file sizes 4, 8, 12, and 16 KB. The obtained average values are plotted in Figure 3 in order to compare the results. All observed confidence intervals are less than 0.02% and are therefore not additionally shown.

The allocation problems increases with increasing file size. The explanation for this phenomenon is that the increased allocation requirements for each file causes the file system to become less forgiving towards a less-than-optimal distribution. Viewed from another perspective, the allocation of a file with a file size of 8 KB is the same as allocating two 4 KB files to the same track. Thus, achieving a certain disk utilization with 8 KB files is the same as using every computed hash value twice for 4 KB files. Deviations from an optimal distribution are thus increasingly worse for increasing file sizes.

B. Experimental Results

The benchmarks used to evaluate hashFS performance are based on "SolidFSBench", a benchmark environment specifically developed to benchmark very large file sets. Existing benchmark tools, e.g. filebench [21], have shown weaknesses when dealing with millions of different files.

The file set generation is configurable by the root directory of the file set, the number of files, the size and size distribution of the files, the number of directories and the maximum directory depth. The created directory tree is a balanced n-ary tree with n chosen in a way so that the maximum depth is not exceeded. Since no in-memory representation of the whole tree is kept, there is no limit besides disk space and file system limits regarding the maximum number of directories or files created. Differently from filebench the workload generation is done offline instead of online during the workload execution. Main advantages of this approach are that it is now possible to execute exactly the same workload multiple times and that it is no longer necessary to have knowledge of the whole file set during workload execution, which circumvents limits regarding the possible workload size. Furthermore, the usage of computation intensive randomization during workload generation has no impact on the benchmark performance.

Each file set configuration is benchmarked 10 times. The file systems are mounted with the *noatime* and *nodiratime* options, preventing unnecessary write accesses.

The benchmarks were run on four test platforms, each having exactly the same hardware and software configuration with two WDC WD5002ABYS-0 500 GB hard disks. The benchmarked file sets were located on a dedicated hard drive.

In order to limit the size of the file sets needed to achieve high disk utilizations and the hardware requirements, partitions with a size of 90 GB, beginning at sector 0, were created. In order to have realistic disk/RAM ratios

and comparable results, we scaled down the available RAM using kernel boot options.

We only used the number of tracks of a disk as geometry information. Furthermore, we will show in the following that it can be even beneficial to use a different than the real number of tracks in certain scenarios. For all benchmarks where the disk geometry is not specified, a geometry with 500 sectors per track is used. Given the partition size of 90 GB and removing all tracks used by ext2 metadata, the resulting geometry contains a total of $377,438$ tracks. For each track, a 4 KB track inode block is allocated in the hashing file system. As a result, the utilization of the benchmark partitions for the same number of allocated files is around 1.6% greater for the hashing file system than it is for the ext2 file system.

Two file sets with different directory structures were generated in order to examine the behavior of the different lookup approaches of the file systems.

Flat File Set:
: The first file set contains relatively few directories with $10,000$ files in each directory. The tree has a maximum depth of 2.

Deep File Set:
: The second file set contains one hundred times as many directories, with a maximum depth of 6.

Influence of Cache State on ext2 Performance: The ext2 read performance strongly depends on the cache size and state. This can be explained by the ext2 lookup strategy: Each lookup operation for a path component of the file results in disk I/O, unless the metadata for that path component is already available in the cache. The more lookup operations are executed, the more metadata is cached. This naturally increases the probability that at least parts of the required metadata are already cached. Because the available cache size is limited by main memory, performance will stabilize after a certain number of lookup operations, as all available cache space is already occupied and new metadata read from disk will replace old cached metadata. After this state is achieved, the cache is referred to as "hot".

The amount of permanent read operations in the target scenario will quickly warm-up the cache. Therefore, benchmarking results obtained with a cold cache are non-significant. Accordingly, before each benchmark of the ext2 file system, a number of read operations were executed in order to warm-up the cache. Figure 4 shows the development of the number of achieved read operations per second. The configuration contained 18 million files for both file sets and a main memory size of 1024 MB, the second largest used in the benchmarks. After approximately $50,000$ read operations, the cache is warmed-up for both file sets and a relatively steady read performance is achieved.

In the following, we will perform $100,000$ read operation as warm-up for all benchmarks. It is safe to assume that it will result in a warmed-up cache for all other configurations as well.

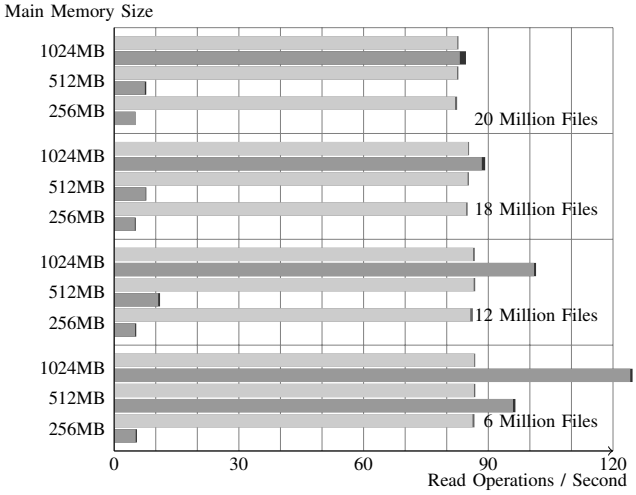

Figure 5. Random Read Performance for the Flat File Set (Light = hashFS, Dark = ext2)

Uniformly Distributed Read Benchmarks: The first set of benchmarks consists entirely of uniformly distributed random read benchmarks. Random reads can be considered as the worst case scenario for a file system, because the probability that a file which is read in is already contained in the cache is very small. As such, the obtained results show the worst case behavior of the file systems.

We highlight that the ratio between the available main memory to disk capacity/number of files must be considered carefully. For this purpose, the number of files in the benchmarked file sets is varied between 6, 12, 18 and 20 million files, corresponding to a utilization of approximately 27%, 54%, 81% and 90% of the benchmark partitions. Furthermore, each benchmark is performed with a main memory size of 256 MB, 512 MB and 1024 MB.

Figure 5 shows the achieved read performance for the flat file set, given as completed read operations per second. Darker bars show ext2 file system performance, lighter bars the performance of hashFS. The darker colored sections at the end of the bars indicate the corresponding confidence intervals of 95%. Figure 6 shows the analogous data for the deep file set. One thing becomes immediately apparent: Results for the hashing file system are almost constant across all benchmarked parameters, while ext2 performance heavily depends on cache sizes.

Table II shows the average number of read operations which were executed in the block layer for a read of single read operation of a file. The corresponding confidence intervals were very small and therefore omitted. Comparing the scheduled read operations for each file with the achieved performance, it is obvious that both parameters inversely correlate: An increasing number of disk accesses leads to a

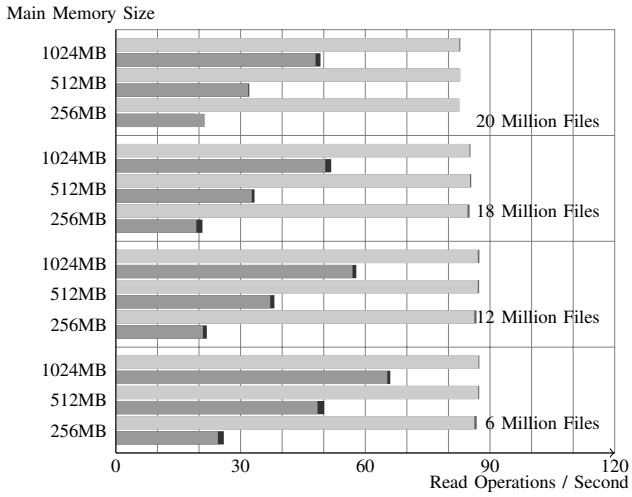

Figure 6. Random Read Performance for the Deep File Set (Light = hashFS, Dark = ext2)

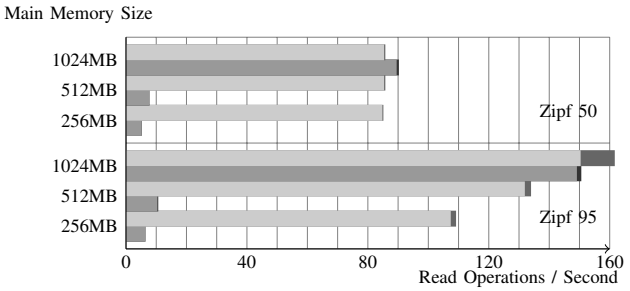

Figure 7. Zipf Performance for Flat File Set (Light = hashFS, Dark = ext2)

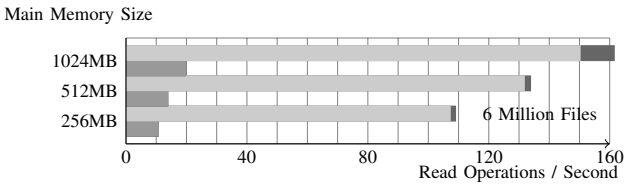

Figure 8. btrfs Performance for Flat File Set (Light = hashFS, Dark = btrfs)

decreasing performance in all cases. It is interesting to see that the block layer splits every read operation for hashFS into two adjacent block read operations.

Benchmarks using a Zipf Distribution: Access patterns for files on the internet have been shown to closely resemble a Zipfian distribution with an approximate α-value of 0.95 [3], [2]. While one or more cache layers flatten the skew of the distribution by processing frequently accessed files, they probably will not result in a uniformly distribution as assumed in the last section. We have generated in our environment a second workload with skew factors of $\alpha = 0.95$ and $\alpha = 0.5$.

Figure 7 shows the obtained results for the flat file set. Results for the second file set look very similar. A skew factor of 0.5 only shows a marginal performance increase compared to uniform random reads. It seems that even the maximum main memory size of 1024 MB is not big enough to capitalize on the uneven distribution. The observed results for a skew factor of 0.95 differ significantly. In all cases, performance is improved, as more reads can be directly served from the cache. This effect increases with increasing cache size. Once again, the performance of the hashing file system is not influenced by the different file sets, while the performance of ext2 has the same dependencies, which have been observed in the random read benchmarks.

Table II
AVERAGE BLOCK READ OPERATIONS PER FILE

	Flat File Set				Deep File Set			
	6M	12M	18M	20M	6M	12M	18M	20M
HashFS								
256MB	1.985	2.001	2.061	2.123	1.984	1.992	2.076	2.125
512MB	1.968	1.970	2.047	2.103	1.971	1.976	2.067	2.118
1024MB	1.968	1.971	2.047	2.103	1.970	1.976	2.067	2.118
ext2								
256MB	26.738	27.558	28.088	28.696	6.860	7.804	8.338	8.295
512MB	2.293	12.372	18.684	20.022	3.702	4.704	5.295	5.427
1024MB	1.776	1.941	2.141	2.261	2.470	3.126	3.540	3.702

Benchmarks using the btrfs file system: The ext2 file system, which was the foundation of our development and the file system against we compared hashFS against up to now, is relatively simple, still often used, but arguably not the most modern file system available.

To show that our results do not only rely on a bad performance of ext2, we performed additional tests using the state-of-the-art btrfs file system [9]. Figure 8 shows the results using the flat file set with 6 million files with different amounts of main memory. The btrfs file system is used in its version 0.16 using default settings and the same system environment as before.

Surprisingly, the btrfs file system performs much less read operations per second in our random read scenario than hashFS and ext2. Additionally, the number of reads per seconds decreases if less memory is available to cache inodes and dentry data. However, with only 256 MB of main memory, the performance is better than ext2's whose performance has dropped in that configuration.

Influence of Track Size on hashFS Performance: All hashFS benchmarks so far have been executed using a geometry with a fixed track size of 500 sectors. The following benchmarks examine the effect of different track sizes in the supplied geometry on the performance of the hashing file system. As previous benchmarks have already shown, hashFS performance is independent of the composition of the file set and, at least with a uniform access distribution,

Figure 9. hashFS Performance for Different Track Sizes (Flat File Set, 6M files, 1024 MB RAM)

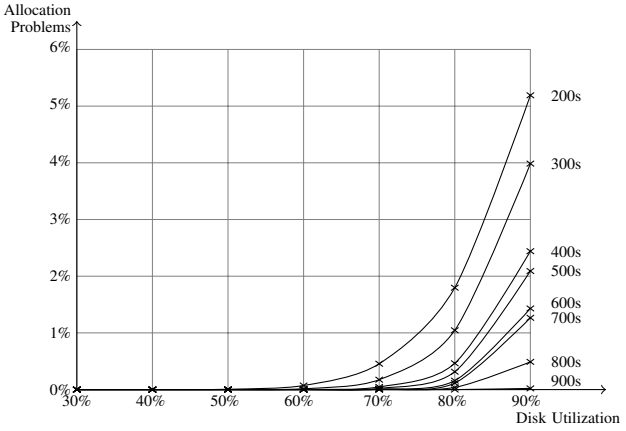

Figure 10. Development of Allocation Problems for Different Geometries

independent of the cache size. Furthermore, the number of files allocated in the file set only plays a role above a certain level of disk utilization. The following benchmarks were executed with constant values for these parameters: The flat file set, containing 6 million files, was used, and main memory was kept constant at 1024 MB.

The observed impact of the track size on the performance is very distinct: Increasing the size by 100 sectors resulted in a decrease in performance by approximately 3 ± 1 reads per second (see Figure 9). The performance gain for smaller track sizes can be explained by smaller read operations. At first glance, the obtained benchmarking results suggest choosing a very small track size in order to maximize the performance. There are, however, some implications to changing track sizes. On the one hand, the smaller the track size, the more overall tracks exist, increasing their required capacity. For a track size of 100 sectors, the track inodes already occupy 8% of the disk capacity. On the other hand, decreasing the size of the tracks increases the expected allocation problems, because there is not enough space available at the hashed tracks. The influence of this effect is shown in Figure 10.

V. Conclusion

The evaluation of the experimental hashFS file system shows that the hashing-based pathname lookup approach is able to increase small file read performance for a workload typical in many Web 2.0 scenarios. A single small file read is performed with a single seek nearly independent of the organization and size of the file set or the available cache. In contrast, we have shown that ext2 heavily depends on the ability to cache inodes and directory information to perform well. Additionally, our approach does not rely on a name-based or temporal locality or large in-memory lookup tables. We also described how the pathname lookup strategy can be integrated into the VFS layer, so that other file systems are able to make similar optimizations.

References

[1] E. Jaffe and S. Kirkpatrick, "Architecture of the internet archive," in *Proceedings of the 2nd SYSTOR Conference*, 2009.

[2] L. Breslau, P. Cue, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *Proceedings of the IEEE International Conference on Computer and Communications (INFOCOM)*, 1999, pp. 126–134.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira, "Characterizing reference locality in the www," in *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

[4] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *IEEE Computer*, vol. 27, no. 3, pp. 17–28, 1994.

[5] R. Card, T. Ts'o, and S. Tweedie, "Design and implementation of the second extended filesystem," in *Proceedings of the First Dutch International Symposium on Linux*, 1994.

[6] S. Tweedie, "Ext3: Journaling filesystem," in *Proceedings of the Ottawa Linux Symposium*, 2000.

[7] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux Symposium*, 2007.

[8] V. Prabhakaran, A. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 105–120.

[9] C. Mason, "Btrfs: A copy on write, snapshotting fs," LINUX Kernel Developing Mailing List (LKML), 2009.

[10] S. Best, D. Gordon, and I. Haddad, "Kernel korner: Ibm's journaled filesystem," *Linux Journal*, vol. 2003, no. 105, p. 9, 2003.

[11] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system," in *Proceedings of the USENIX Annual Technical Conference*, 1996, pp. 1–14.

[12] D. Chinner and J. Higdon, "Exploring high bandwidth filesystems on large systems," in *Ottawa Linux Symposium*, 2006.

[13] G. R. Ganger and M. F. Kasshoek, "Embedded inodes and explicit grouping: exploiting disk bandwidth for small files," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 1997.

[14] L. Jun, L. Xianliang, L. Guangchun, H. Hong, and Z. Xu, "Stfs: a novel file system for efficient small writes," *ACM SIGOPS Operating Systems Review*, 2002.

[15] J. Ledlie, "Dámelo! an explicitly co-locating web cache file system," Master's thesis, University of Wisconsin, 2000.

[16] J. Wang, R. Min, Y. Zhu, and Y. Hu, "Ucfs - a novel user-space, high performance, customized file system for web proxy servers," *IEEE Transactions on Computers*, 2002.

[17] E. Gabber and E. Shriver, "Let's put netapp and cacheflow out of business!" in *Proceedings of the 9th Workshop on ACM SIGOPS European workshop*, 2000, pp. 85 – 90.

[18] A. Badam, K. Park, V. S. Pai, and L. L. Peterson, "Hashcache: cache storage for the next billion," in *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. USENIX Association, 2009.

[19] J. Piernas, T. Cortes, and J. M. Garcia, "Dualfs: a new journaling file system without meta-data duplication," in *Proceedings of the 16th international Conference on Supercomputing*, 2002.

[20] J. Gim, Y. Won, J. Chang, J. Shim, and Y. Park, "Dig: Rapid characterization of modern hard disk drive and its performance implication," in *Proceedings of the 5th International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2008, pp. 74–83.

[21] A. Wilson, "The new and improved filebench," Work-in-Progress Report at the 6th USENIX Conference on File and Storage Technologies (FAST), 2008.