Recent Filesystem Optimisations in FreeBSD

Ian Dowse `<iedowse@freebsd.org>`
Corvil Networks.
David Malone `<dwmalone@freebsd.org>`
CNRI, Dublin Institute of Technology.

Abstract

In this paper we summarise four recent optimisations to the FFS implementation in FreeBSD: *soft updates*, *dirpref*, *vmiodir* and *dirhash*. We then give a detailed exposition of dirhash's implementation. Finally we study these optimisations under a variety of benchmarks and look at their interactions. Under micro-benchmarks, combinations of these optimisations can offer improvements of over two orders of magnitude. Even real-world workloads see improvements by a factor of 2–10.

1 Introduction

Over the last few years a number of interesting filesystem optimisations have become available under FreeBSD. In this paper we have three goals. First, in Section 2, we give a summary of these optimisations. Then, in Section 3, we explain in detail the *dirhash* optimisation, as implemented by one of the authors. Finally we present the results of benchmarking various common tasks with combinations of these optimisations.

The optimisations which we will discuss are *soft updates*, *dirpref*, *vmiodir* and *dirhash*. All of these optimisations deal with filesystem metadata. *Soft updates* alleviate the need to make metadata changes synchronously; *dirpref* improves the layout of directory metadata; *vmiodir* improves the caching of directory metadata; and *dirhash* makes the searching of directories more efficient.

2 Summaries

For each optimisation we will describe the performance issue addressed by the optimisation, how it is addressed and the tradeoffs involved with the optimisation.

2.1 Soft Updates

Soft updates is one solution to the problem of keeping on-disk filesystem metadata recoverably consistent. Traditionally, this has been achieved by using synchronous writes to order metadata updates. However, the performance penalty of synchronous writes is high. Various schemes, such as journaling or the use of NVRAM, have been devised to avoid them [14].

Soft updates, proposed by Ganger and Patt [4], allows the reordering and coalescing of writes while maintaining consistency. Consequently, some operations which have traditionally been durable on system call return are no longer so. However, any applications requiring synchronous updates can still use fsync(2) to force specific changes to be fully committed to disk. The implementation of soft updates is relatively complicated, involving tracking of dependencies and the roll forward/back of transactions. Thus soft updates trades traditional write ordering, memory usage and code complexity for significantly faster metadata changes.

McKusick's production-quality implementation [8] has been found to be a huge win in real-world situations. A few issues with this implementation persist, such as failure to maintain NFS semantics. Long standing issues, such as disks appearing full because of outstanding transactions have recently been resolved.

In FreeBSD 5.0-current, soft updates has been combined with snapshots to remove the need for a full `fsck` at startup [7]. Instead, the filesystem can be safely preened while in use.

2.2 Dirpref

Dirpref is a modification to the directory layout code in FFS by Orlov [10]. It is named after the `ffs_dirpref` function, which expresses a preference for which inode should be used for a new directory.

The original policy implemented by this function was to select from among those cylinder groups with above the average number of free inodes, the one with the smallest number of directories. This results in directories being distributed widely throughout a filesystem.

The new dirpref policy favours placing directories close to their parent. This increases locality of reference, which reduces disk seek times. The improved caching decreases the typical number of input and output operations for directory traversal. The obvious risk associated with dirpref is that one section of the disk may become full of directories, leaving insufficient space for associated files. Heuristics are included within the new policy to make this unlikely. The parameters of the heuristic are exposed via `tunefs` so that they may be adjusted for specific situations.

2.3 Vmiodir

Vmiodir is an option that changes the way in which directories are cached by FreeBSD. To avoid waste, small directories were traditionally stored in memory allocated with `malloc`. Larger directories were cached by putting pages of kernel memory into the buffer cache. Malloced memory and kernel pages are relatively scarce resources so their use must be restricted.

Some unusual access patterns, such as the repeated traversal of a large directory tree, have a working set that consists almost exclusively of directories. Vmiodir enables direct use of the virtual memory system for directories, permitting maximal caching of such working sets, because cached pages from regular files can be flushed out to make way for directories.

Here the trade-off is that while some memory is wasted because 512-byte directories take up a full physical page, the VM-backed memory is better managed and more of it is usually available. Once a modest amount of memory is present in the system, this results in better caching of directories. Initially added to FreeBSD in June 1999 by Dillon, vmiodir was disabled by default until real-world use confirmed that it was usually at least as good as the more frugal scheme.

2.4 Dirhash

A well-known performance glitch in FFS is encountered when directories containing a very large number of entries are used. FFS uses a linear search through the directory to locate entries by name and to find free space for new entries. The time consumed by full-directory operations, such as populating a directory with files, can become quadratic in the size of the directory.

The design of filesystems since FFS has often taken account of the possibility of large directories, by using B-Trees or similar structures for the on-disk directory images (e.g., XFS [16], JFS [1] and ReiserFS [13]).

Dirhash retrofits a directory indexing system to FFS. To avoid repeated linear searches of large directories, dirhash builds a hash table of directory entries on the fly. This can save significant amounts of CPU time for subsequent lookups. In contrast to filesystems originally designed with large directories in mind, these indices are not saved on disk and so the system is backwards compatible.

The cost of dirhash is the effort of building the hash table on the first access to the directory and the space required to hold that table in memory. If the directory is accessed frequently then this cost will be reclaimed by the saving on the subsequent lookups.

A system for indexing directories in Linux's Ext2 filesystem has also recently been developed by Phillips [11]. This system saves the index on disk in a way that provides a good degree of backwards compatibility.

3 Inside Dirhash

Dirhash was implemented by Ian Dowse in 2001 in response to discussions on the `freebsd-hackers` mailing list. Dirhash builds a summary of the directory on its first access and then maintains that summary as the directory is changed. Directory lookup is optimised using a hash table and the creation of new directory entries is optimised by maintaining an array of free-space summary information.

Directories are hashed only if they are over a certain size. By default this size is 2.5KB, a figure which should avoid wasting memory on smaller directories where performance is not a problem. Hashing a directory consumes an amount of memory proportional to its size. So, another important parameter for dirhash is the amount of memory it is allowed to use. Currently it works with a fixed-size pool of memory (by default 2MB).

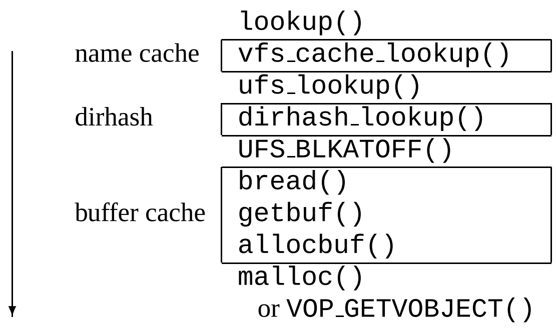


Figure 1: Directory Caching in FreeBSD. The major caches affecting directory lookups are shown on the left and the corresponding call stack is shown on the right.

Both these parameters of dirhash are exposed by sysctl, in addition to another option that enables extra sanity checking of dirhash's summary information.

3.1 Existing Directory Caching in FreeBSD

The translation of filenames to vnodes is illustrated in Figure 1. An important part of this process is the name cache, implemented in `vfs_cache.c`. Here a global hash table is used to cache the translation of <directory vnode, path component> pairs to vnodes.

A cache entry is added when a component is first looked up. Cache entries are made for both successful and failed lookups. Subsequent lookups then avoid filesystem-level directory searches, unless access to the underlying directory is required.

Name cache entries persist until the associated vnodes are reclaimed. Cache entries for failed lookups are limited to a fixed proportion of the total cache size (by default 1/16). Some operations result in the entries relating to a particular vnode or a particular filesystem being explicitly purged.

The name cache can help in the case of repeated accesses to existing files (e.g., Web serving) or accesses to the same non-existent file (e.g., Apache looking for `.htaccess` files). The name cache cannot help when the filename has not been previously looked up (e.g., random file accesses) or when the underlying directory must be changed (e.g., renaming and removing files).

3.2 Dirhash Based Name Lookups

Like the name cache, dirhash also maintains a hash table for mapping component names to offsets within the directory. This hash table differs from the global name cache in several ways:

- The tables are per directory.
- The table is populated on creation by doing a pass over the directory, rather than populated as lookups occur.
- The table is a complete cache of all the directory's entries and can always give a definitive yes or no answer whereas the name cache contains only the positive and negative entries added by lookups.
- Dirhash stores just the offset of directory entries within its table. This is sufficient to read the appropriate directory block from the buffer cache, which contains all the information needed to complete the lookup. In contrast, the VFS name cache is a filesystem-independent heavyweight cache storing the component name and references to both directory and target vnodes within its hash table.

Since dirhash only needs to store an offset, it uses an open-storage hash table rather than using linked lists (as the name cache does). Use of linked lists would increase the table size by a factor of two or three. Hash collisions within the table are resolved using linear-probing. This has the advantage of being simple and, unlike some other open storage schemes, some deleted entries can be reclaimed without the need to rebuild the table. Specifically, deleted entries may be marked as empty when they lie at the end of a chain.

Due to the initial population of the dirhash table, filling the name cache with a working set of m files from a directory of size n should cost $n + m$ rather than nm for random accesses without dirhash. If a working set of files is established in the name cache, then the dirhash hash table will no longer be accessed. The case of sequential directory access was traditionally optimised in the FFS code and would have given a cost of m. A sequential lookup optimisation is also present in dirhash; after each successful lookup, the offset of the following entry is stored. If the next lookup matches this stored value then the sequential optimisation is enabled. While the optimisation is enabled, dirhash first scans through the hash chain looking for the offset stored by the previous lookup. If the offset is found, then dirhash consults that offset before any other.

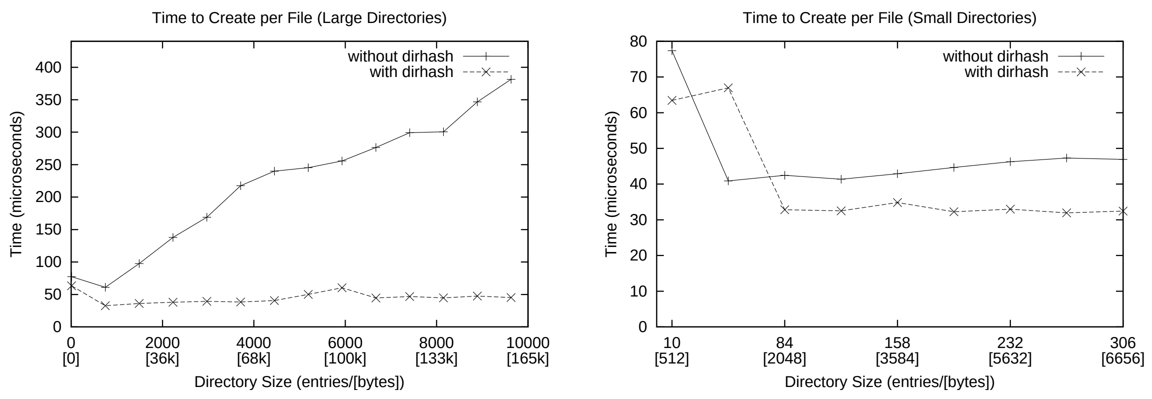


Figure 2: File Creation: Cost per Creation

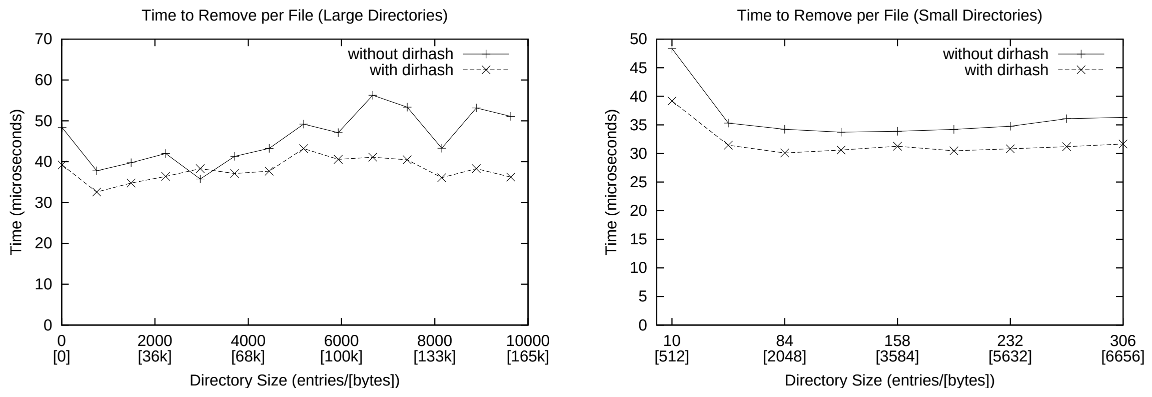


Figure 3: File Removal: Cost per Removal

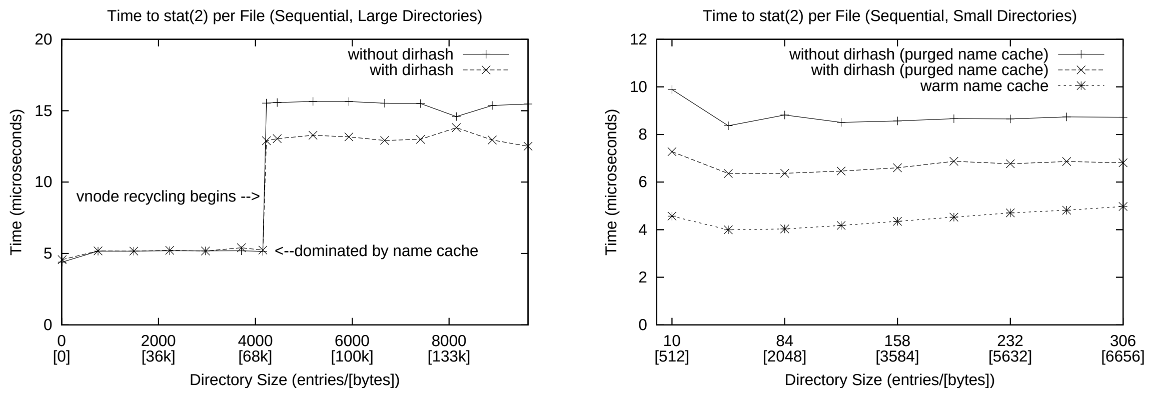
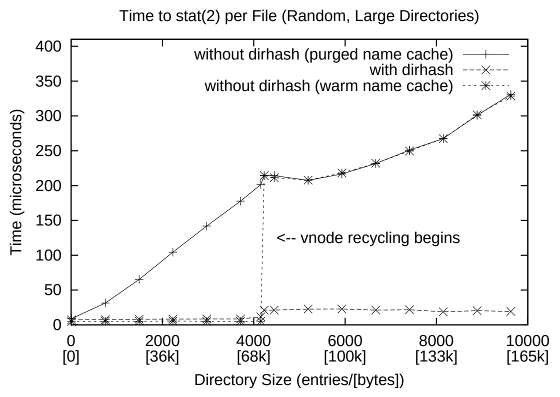


Figure 4: Sequential stat(2) of Files: Cost per stat(2)

A comparison between cost per operation of the traditional linear scheme and dirhash for file creation and removal is shown in Figures 2 and 3 respectively. The costs for large directories are shown on the left and those for small directories are on the right. All these tests were performed with our benchmarking hardware described in Section 4.2. For creation with the linear scheme, we can see the cost of creations increasing as the directory size increases. For dirhash the cost remains constant at about $50\mu s$. Files were removed in sequential order, so we expect the cost to be roughly constant for both schemes. This is approximately what we see (dirhash around $35\mu s$, linear around $45\mu s$), though the results are quite noisy (perhaps due to delayed operations).



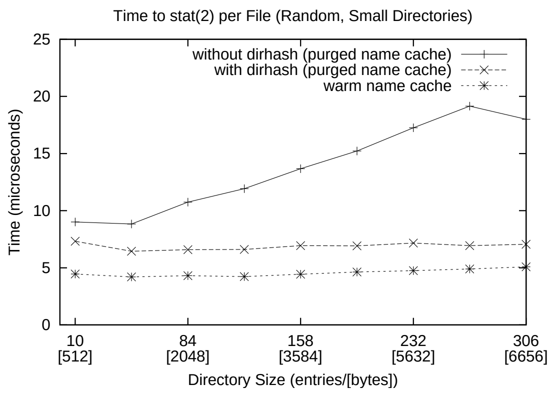
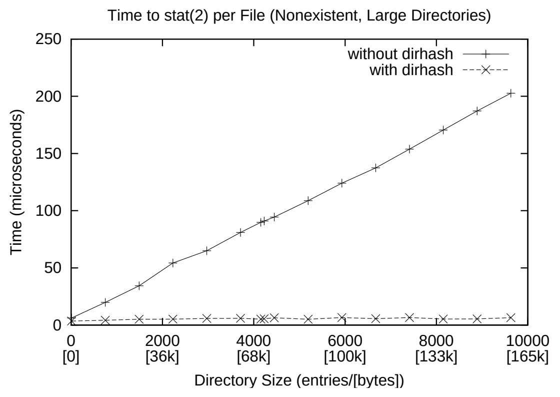


Figure 5: Random stat(2) of Files: Cost per stat(2)



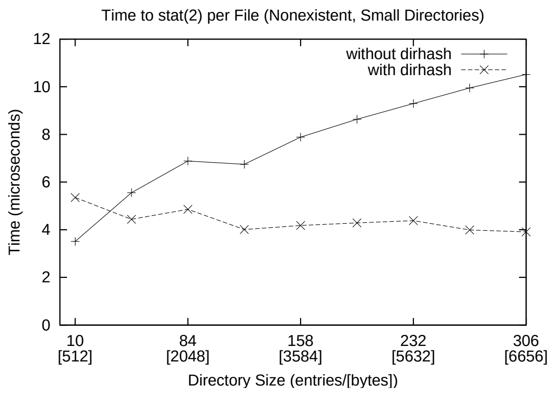


Figure 6: stat(2) of Nonexistent Files: Cost per stat(2)

The cost of `stat` operations, both with and without dirhash, are shown in Figure 4 for sequentially ordered files, in Figure 5 for randomly ordered files and Figure 6 for nonexistent files. Where the name cache has a significant impact on the results, we show figures for both a purged cache and a warm cache. Sequential operation here shows a cost of about $5\mu s$ if the result is in the name cache, $7\mu s$ for dirhash and $9\mu s$ for the linear search. For random ordering, results are similar for the name cache and dirhash. Here the linear search shows the expected linear growth rate. Both these graphs show an interesting jump at about 4000 files, which is explained below. Operating on nonexistent files incurs similar costs to operating on files in a random order. As no lookups are repeated, the name cache has no effect.

As mentioned, only directories over 2.5KB are usually indexed by dirhash. For the purposes of our tests, directories of any size were considered by dirhash, allowing us to assess this cut-off point. It seems to have been a relatively good choice, as dirhash look cheaper in all cases at 2KB and above. However, in practice, directories will be more sparsely populated than in these tests as file creation and removal will be interleaved, so 2.5KB or slightly higher seems like a reasonable cut-off point.

The elbow in Figure 2 and the jump in Figures 4 and 5 at around 4000 vnodes is caused by an interaction between the test used, the way vnodes are recycled and the name cache:

1. For this test, `stat` is called on the files. Files are not held open nor read, meaning that no reference is held on the vnode in the kernel.

2. Consequently the vnodes end up on a ready-to-reuse list, but otherwise remain valid. Once there are `kern.minvnodes` vnodes allocated in the system, vnodes on the ready-to-reuse list are used in preference to allocating new vnodes. At this stage the vnode is recycled and its old contents becomes invalid.

3. The invalidation of vnodes purges the name cache of entries referring to those vnodes.

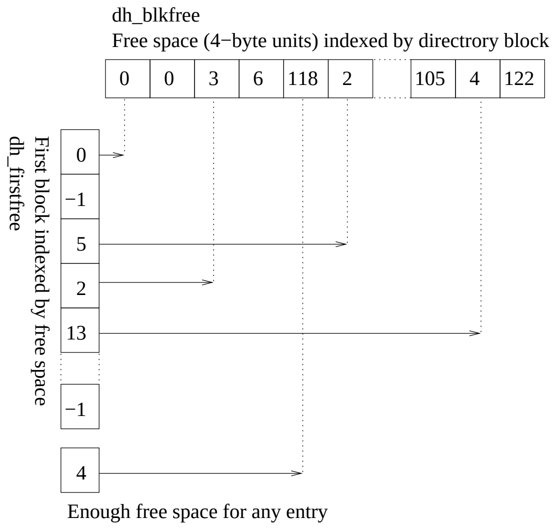

dh_blkfree
Free space (4−byte units) indexed by directrory block
0
0
3
6
118
2
105
4
122
First block indexed by free space
dh_firstfree
0
−1
5
2
13
−1
4
Enough free space for any entry

Figure 7: Example dh_blkfree and dh_firstfree. The free space in block is dh_blkfree[block]. dh_firstfree[space] gives the first block with exactly that much free space. The final entry of dh_firstfree is the first block with enough space to create an entry of any size.

The default value of kern.minvnodes on the test system is 4458. Altering this value causes the jump in the graph to move accordingly. In the usual situation where files are open and data is in the buffer cache, vnode reuse begins much later.

3.3 Locating Free Space with Dirhash

Dirhash also maintains two arrays of free-space summary information (see Figure 7 for an example). The first array, dh_blkfree, maps directory block numbers to the amount of free space in that block. As directory blocks are always 512 bytes and the length of entries is always a multiple of 4 bytes, only a single byte per block is required.

The second array, dh_firstfree, maps an amount of free space to the first block with exactly that much free space, or to -1 if there is no such block. The largest legal FFS directory entry has a size of 264 bytes. This limits the useful length of the dh_firstfree array because there is never a need to locate a larger slot. Hence, the array is truncated at this size and the last element refers to a block with at least enough space to create the largest directory entry.

As entries are allocated and freed within a block, the first array can be recalculated without rescanning the whole block. The second array can then be quickly updated based on changes to the first.

Without dirhash, when a new entry is created, a full sweep of the directory ensures that the entry does not already exist. As this sweep proceeds, a search for suitable free space is performed. In order of preference, the new entry is created in:

- the first block with enough contiguous free space;
- the first block with enough free space but requires compaction;
- a new block.

When dirhash is used, the full sweep is no longer necessary to determine if the entry exists and the dh_firstfree array is consulted to find free space. The new entry is created in the first block listed in this array that contains enough free space.

These two policies are quite different, the former preferring to avoid compaction and the latter preferring to keep blocks as full as possible. Both aim to place new entries toward the start of the directory. This change of policy does not seem to have had any adverse effects.

3.4 Memory Use, Rehashing and Hash Disposal

Dirhash uses memory for the dirhash structure, the hash table and the free-space summary information. The dirhash structure uses 3 pointers, 9 ints, 1 doff_t and an array of 66 ints for dh_firstfree, about 320 bytes in total.

The hash table is initially sized at 150% of the maximum number of entries that the directory could contain (about 1 byte for every 2 bytes of the directory's size). The table used is actually two-level, so an additional pointer must be stored for every 256 entries in the table, or one pointer for every 2KB on the disk. The dh_blkfree array is sized at 150% of the blocks used on the disk, about 1 byte for every 340 on disk. This gives a total memory consumption of 266 bytes$+0.505$dirsize on a machine with 32-bit pointers or 278 bytes $+\ 0.507$dirsize on a 64-bit platform.

The dirhash information is just a cache, so it is safe to discard it at any time and it will be rebuilt, if necessary, on the next access. In addition to being released when the directory's vnode is freed, the hash is freed in the following circumstances:

- Adding an entry would make the table more than 75% full. Due to the linear probing algorithm, some hash table slots for deleted entries may not be reclaimed and marked empty. These are also counted as "used" slots.
- The directory grows too large for the number of entries in the `dh_blkfree` array.
- The directory shrinks so that the `dh_blkfree` array is less than one eighth used. This corresponds to the directory being truncated to about 20% of its size at the time the hash was built.
- The hash is marked for recycling.

Hashes may be marked for recycling when dirhash wants to hash another directory, but has reached its upper memory limit. If dirhash decided to recycle other hashes, then the hash table and the `dh_blkfree` array are released, but the dirhash structure remains allocated until the next access to the corresponding directory. This simplifies the interaction between the locking of the vnode and the locking of the dirhash structure.

To avoid thrashing when the working set is larger than the amount of memory available to dirhash, a score system is used which achieves a hybrid of least-recently-used (LRU) and least-frequently-used (LFU). This is intended to limit the rate of hash builds when the working set is large.

3.5 Implementation Details and Issues

The implementation of dirhash is relatively straightforward, requiring about 100 lines of header and 1000 lines of C. The actual integration of the dirhash code was unobtrusive and required only small additions to the surrounding FFS code at points where lookups occurred and where directories were modified. This means it should be easy for other systems using FFS to import and use the code.

The hash used is the FNV hash [9], which is already used within the FreeBSD kernel by the VFS cache and in the NFS code. The hash table is maintained as a two-level structure to avoid the kernel memory fragmentation that could occur if large contiguous regions were allocated. The first level is an array of pointers allocated with `malloc`. These point to fixed-size blocks of 256 offsets, which are currently allocated with the zone allocator.

Some of the issues that emerged since the initial integration of dirhash include:

Expanded size of in-core inode: In FreeBSD-4, the addition of the dirhash pointer to the in-core inode structure resulted in the effective size of an in-core inode going from 256 bytes to 512 bytes. This exacerbated an existing problem of machines running out of memory for inodes. Some of the fields in the inode structure were rearranged to bring the size back down to 256 bytes.

Unexpected directory states: Dirhash's sanity checking code was too strict in certain unusual cases. In particular, `fsck` creates mid-block directory entries with inode number zero; under normal filesystem operation this will not occur. Also, dirhash has to be careful not to operate on deleted directories, as `ufs_lookup` can be called on a directory after it has been removed.

Bug in sequential optimisation: The sequential optimisation in dirhash was not functioning correctly until mid-November 2001, due to a typo which effectively disabled it. Once correctly enabled, sequential lookups seem a little faster (5%) rather than a little slower (1–2%) when compared to sequential non-dirhash lookups. This is probably a combination of two factors. First the non-dirhash code remembers the last block in which there was a successful lookup, but dirhash remembers the exact offset of the entry. Thus the non-dirhash code must search from the beginning of the block. Second, if the entry was at the end of a block, then the traditional optimisation may result in two blocks being fetched from the buffer cache, rather than one.

`dh_firstfree` bug: The last entry of the `dh_firstfree` array was not always updated correctly, unless there was a block with exactly enough space for the largest directory entry. This resulted in unnecessarily sparse directories.

More aggressive cleaning of deleted entries: If the deletion of a hash entry resulted in a chain ending with an empty slot, then dirhash can mark all the slots at the end of the chain as empty. Originally it was only marking the slots after the deleted entry as empty.

Improved hash function: Originally, the final stage of the hash function was to `xor` the last byte of the filename into the hash. Consequently, filenames differing only in the last byte end up closer together in the hash table. To optimise this common case, the address of the dirhash structure is hashed after the filename. This results in slightly shorter hash chains and also provides some protection against hash collision attacks.

The last three issues were uncovered during the writing of this paper (the version of dirhash used for the benchmarks predates the resolution of these issues).

On investigation, several dirhash bug reports have turned out to be faulty hardware. Since dirhash is a redundant store of information it may prove to be an unwitting detector of memory or disk corruption.

It may be possible to improve dirhash by storing hashes in the buffer cache and allowing the VM system to regulate the amount of memory which can be used. Alternatively, the slab allocator introduced for FreeBSD-5 may provide a way for dirhash to receive feedback about memory demands and adjust its usage accordingly. These options are currently under investigation.

Traditionally, directories were only considered for truncation after a create operation, because the full sweep required for creations was a convenient opportunity to determine the offset of the last useful block. Using dirhash it would be possible to truncate directories when delete operations take place, instead of waiting for the next create operation. This has not yet been implemented.

3.6 Comparisons with Other Schemes

The most common directory indexing technique deployed today is probably the use of on-disk tree structures. This technique is used in XFS [16] and JFS [1] to support large directories; it is used to store all data in ReiserFS [13]. In these schemes, the index is maintained on-disk and so they avoid the cost of building the index on first access. The primary downsides to these schemes are code complexity and the need to accommodate the trees within the on-disk format.

Phillips's HTree system [11] for Ext2/3 is a variant of these schemes designed to avoid both downsides. HTree uses a fixed-depth tree and hashes keys before use to achieve a more even spread of key values.

The on-disk format remains compatible with older versions of Ext2 by hiding the tree within directory blocks that are marked as containing no entries. The disk format is constructed cleverly so that likely changes made by an old kernel will be quickly detected and consistency checking is used to catch other corruption.

Again, HTree's index is persistent, avoiding the cost of dirhash's index building. However, the placing of data within directory blocks marked as empty is something which should not be done lightly as it may reduce robustness against directory corruption and cause problems with legacy filesystem tools.

Persistent indexes have the disadvantage that once you commit to one, you cannot change its format. Dirhash does not have this restriction, which has allowed the changing of the hash function without introducing any incompatibility. HTree allows for the use of different hashes by including a version number. The kernel can fall back to a linear search if it does not support the hash version.

Another obstacle to the implementation of a tree-based indexing scheme under FFS is the issue of splitting a node of the tree while preserving consistency of the filesystem. When a node becomes too full to fit into a single block it is split across several blocks. Once these blocks are written to the disk, they must be atomically swapped with the original block. As a consequence, the original block may not be reused as one of the split blocks. This could introduce complex interactions with traditional write ordering and soft updates.

A directory hashing scheme that has been used by NetApp Filers is described by Rakitzis and Watson, [12]. Here the directory is divided into a number of fixed length chunks (2048 entries) and separate fixed-size hash tables are used for each chunk (4096 slots). Each slot is a single byte indicating which 1/256th of the chunk the entry resides in. This significantly reduces the number of comparisons needed to do a lookup.

Of the schemes considered here, this scheme is the most similar to dirhash, both in design requirements (faster lookups without changing on-disk format) and implementation (it just uses open storage hash tables to implement a complete name cache). In the scheme described by Rakitzis and Watson the use of fixed-size hash tables, which are at most 50% full, reduces the number of comparisons by a factor of 256 for large directories. This is a fixed speed-up by a factor $N = 256$. As pointed out by Knuth [6], one attraction of hash tables is that as the number of entries goes to infinity the search time for hash table stays bounded. This is not the case for tree based schemes. In principal dirhash is capable of these larger speed-ups, but the decision of Rakitzis and Watson to work with fixed-size hash tables avoids having to rebuild the entire hash in one operation.

4 Testimonial and Benchmark Results

We now present testimonial and benchmark results for these optimisations, demonstrating the sort of performance improvements that can be gained. We will also look at the interaction between the optimisations.

First, our testimonials: extracting the X11 distribution and maintaining a large MH mailbox. These are common real-world tasks, but since these were the sort of operations that the authors had in mind when designing dirpref and dirhash, we should expect clear benefits.

We follow with the results of some well-known benchmarks: Bonnie++ [2], an Andrew-like benchmark, NetApp's Postmark [5], and a variant of NetApp's Netnews benchmark [15]. The final benchmark is the running of a real-world workload, that of building the FreeBSD source tree.

4.1 The Benchmarks

The `tar` benchmark consisted of extracting the `X410src-1.tgz` file from the XFree [17] distribution, running `find -print` on the resulting directory tree and then removing the directory tree.

The folder benchmark is based on an MH inbox which has been in continuous use for the last 12 years. It involves creating 33164 files with numeric names in the order in which they were found to exist in this mailbox. Then the MH command `folder -pack` is run, which renames the files with names 1–33164, keeping the original numerical order. Finally the directory is removed.

Bonnie++ [2] is an extension of the Bonnie benchmark. The original Bonnie benchmark reported on file read, write and seek performance. Bonnie++ extends this by reporting on file creation, lookup and deletion. It benchmarks both sequential and random access patterns. We used version 1.01-d of Bonnie++.

The Andrew filesystem benchmark involves 5 phases: creating directories, copying a source tree into these directories, recursive `stat(2)` of the tree (`find -exec ls -l` and `du -s`), reading every file (using `grep` and `wc`), and finally compilation of the source tree. Unfortunately, the original Andrew benchmark is too small to give useful results on most modern systems, and so scaled up versions are often used. We used a version which operates on 100 copies of the tree. We also time the removal of the tree at the end of the run.

Postmark is a benchmark designed to simulate typical activity on a large electronic mail server. Initially a pool of text files is created, then a large number of *transactions* are performed and finally the remaining files are removed. A transaction can either be a file create, delete, read or append operation. The initial number of files and the number of transactions are configurable. We used version 1.13 of Postmark.

The Netnews benchmark is the simplified version of Karl Swartz's [15] benchmark used by Seltzer et al. [14]. This benchmark initially builds a tree resembling inn's [3] traditional method of article storage (tradspool). It then performs a large number of operations on this tree including linking, writing and removing files, replaying the actions of a news server.

As an example of the workload caused by software development, the FreeBSD build procedure was also used. Here, a copy of the FreeBSD source tree was checked out from a local CVS repository, it was then built using `make buildworld`, then the kernel source code searched with `grep` and finally the object tree was removed.

4.2 Benchmarking Method

The tests were conducted on a modern desktop system (Pentium 4 1.6GHz processor, 256MB ram, 20GB IDE hard disk), installed with FreeBSD 4.5-PRERELEASE. The only source modification was an additional sysctl to allow the choice between the old and new dirpref code.

Each benchmark was run with all 16 combinations of soft updates on/off, vmiodir on/off, dirpref new/old and dirhash maxmem set to 2MB or 0MB (i.e., disabling dirhash). Between each run a sequence of `sync` and `sleep` commands were executed to flush any pending writes. As the directory in which the benchmarking takes place is removed between runs, no useful data could be cached.

This procedure was repeated several times and the mean and standard deviation of the results recorded. Where rates were averaged, they were converted into time-per-work before averaging and then converted back to rates. A linear model was also fitted to the data to give an indication of the interaction between optimisations.

In some cases minor modifications to the benchmarks were needed to get useful results. Bonnie++ does not usually present the result of any test taking less than a second to complete. This restriction was removed as it uses `gettimeofday()` whose resolution is much finer than one second. Recording the mean and standard deviation over several runs should help confirm the validity of the result.

Postmark uses `time()`, which has a resolution of 1 second. This proved not to be of high enough resolution in several cases. Modifications were made to the Postmark timing system to use `gettimeofday()` instead.

4.3 Analysis of Results

The analysis of the results was something of a challenge. For each individual test there were 4 input parameters and one output parameter, making the data 5 dimensional. The benchmarks provided 70 individual test results.

First analysis attempts used one table of results for each test. The table's rows were ranked by performance. A column was assigned to each optimisation and presence of an optimisation was indicated by a bullet in the appropriate column. An example of such a table is shown in Table 1.

To determine the most important effects, look for large vertical blocks of bullets. In our example, soft updates is the most important optimisation because it is always enabled in the top half of the table. Now, to identify secondary effects look at the top of the upper and lower halves of the table, again searching for vertical blocks of bullets. Since dirpref is at the top of both the upper and lower halves, we declare it to be the second most important factor. Repeating this process again we find vmiodir is the third most important factor. Dirhash, the remaining factor, does not seem to have an important influence in this test.

The table also shows the time as a percentage, where no optimisations is taken to be 100%. The standard deviation and the number of runs used to produce the mean result are also shown.

SU	DP	VM	DH	Mean	Time%	$\sigma/\sqrt{n}$	n
●	●	●	●	2032.85	46.78	0.31	4
●	●	●		2058.92	47.38	0.32	4
●	●			2067.24	47.57	0.40	4
●	●		●	2118.72	48.76	0.03	4
●		●	●	2386.64	54.92	0.06	4
●		●		2389.99	55.00	0.05	4
●			●	2479.80	57.07	0.08	4
●				2489.56	57.29	0.04	4
	●	●	●	3681.77	84.73	0.19	4
	●	●		3697.03	85.08	0.44	4
	●			3724.66	85.72	0.17	4
	●		●	3749.88	86.30	0.24	4
		●	●	4260.20	98.04	0.06	4
		●		4262.49	98.09	0.02	4
				4345.38	100.00	0.05	4
			●	4348.22	100.07	0.08	4

Table 1: Rank Table of Results for Buildworld

An attempt to analyse these results statistically was made by fitting to the data the following linear model, containing all possible interactions:

$$\begin{aligned} t &= C_{\text{base}} + \\ &\quad C_{\texttt{SU}}\delta_{\texttt{SU}} + C_{\texttt{DP}}\delta_{\texttt{DP}} + C_{\texttt{VM}}\delta_{\texttt{VM}} + C_{\texttt{DH}}\delta_{\texttt{DH}} + \\ &\quad \text{Coefficients for pairs, triples, } \ldots \end{aligned}$$

Here C_{opts} is a coefficient showing the effect of combining those optimisations and δ_{opt} is 1 if the optimisation is enabled and 0 otherwise. Finding the coefficients is a standard feature of many statistics packages.

Table 2 shows the results of fitting such a model to using the same input data used to build Table 1. Only coefficients that have a statistical significance of more than 1/1000 are shown. The coefficients suggests a 42.71% saving for soft updates, a 14.28% saving for dirpref and a 1.91% saving for vmiodir. The presence of a positive coefficient for the combination of dirpref and soft updates means that their improvements do not add directly, but overlap by 4.57 percentage points.

If a coefficient for a combination is negative, it means that the combined optimisation is more than the sum of its parts. An example is shown in Table 3, where the benefits of dirhash are dependent on the presence of soft updates (see Section 4.4 for more details).

In some cases the results of fitting the linear model was not satisfactory as many non-zero coefficients might be produced to explain a feature more easily understood by looking at a rank table, such as the one in Table 1, for that test.

Attempts to graphically show the data were also made,

Factor	Δ %Time	σ
base	100.00	0.24
DP	-14.28	0.34
VM	-1.91	0.34
SU	-42.71	0.34
DP:SU	4.57	0.48

Table 2: Linear Model Coefficients: Buildworld

Factor	Δ %Time	σ
base	100.00	0.01
SU	-98.67	0.01
DH:SU	-0.96	0.02

Table 3: Linear Model Coefficients: Removing MH mailbox

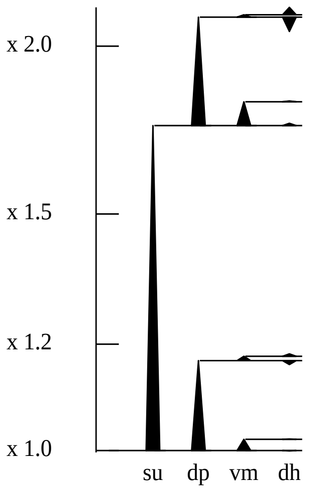


Figure 8: Improvement by Optimisation for Buildworld

but here the 5 dimensional nature of the data is a real obstacle. Displaying the transition between various combinations of optimisations was reckoned to be most promising. A decision was also made to display the data on a log scale. This would highlight effects such as those observed while removing an MH mailbox (Table 3), where soft updates reduces the time to 1.3% and then dirhash further reduces the time to 0.3%, which is a *relative* improvement of around 70%.

Figure 8 again shows the results of the build phase of the Buildworld benchmark, this time using a graphical representation where optimisations are sorted left to right by size of effect. The left-most triangle represents the benefit of enabling soft updates. The next column (two triangles) represents the effects of dirpref, starting with soft updates on and off. The third column (four triangles) represents vmiodir and the right-most triangles represent dirhash. Most of these triangles point up, representing improvements, but some point down, representing detrimental effects.

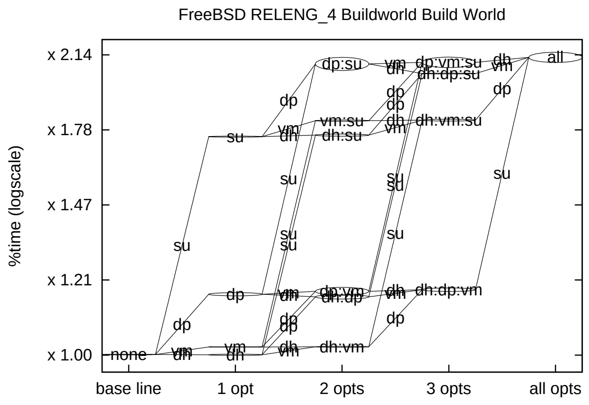


Figure 9: Improvement Lattice for Buildworld

These graphs make it relatively easy to identify important effects and relative effects as optimisations combine. They have two minor weaknesses however. First, not all transitions are shown in these graphs. Second, the variance of results is not shown.

In an effort to produce a representation that shows both of these factors, a graph of nodes was drawn, representing the various combinations of optimisations. An edge was drawn from one node to another to represent the enabling of individual optimisations. The x coordinate of the nodes indicates the number of enabled optimisations and the y coordinate the performance. The height of nodes shows the variance of the measurement.

Figure 9 shows the graph corresponding to the build phase of the Buildworld benchmark. There are many rising edges marked with soft updates and dirpref, indicating that these are significant optimisations. Although these graphs contain all the relevant information, they are often cluttered, making them hard to read.

4.4 Summary of Results

Figure 10 presents a summary of the benchmark results (the Bonnie++ file I/O and CPU results have been omitted for reasons of space). Studying the results indicates that soft updates is the most significant factor in almost all the benchmarks. The only exceptions to this are benchmarks that do not involve writing.

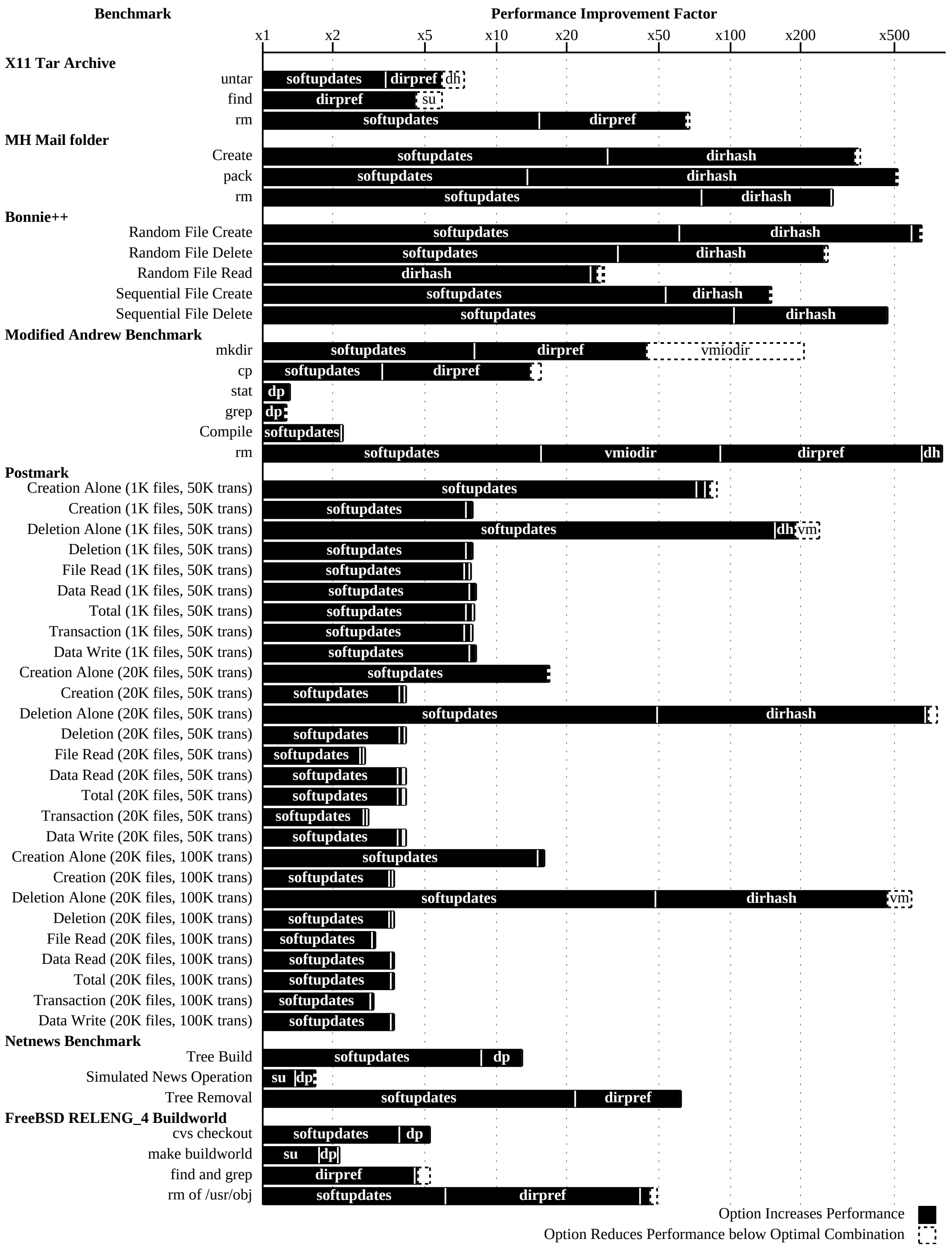


Figure 10: Summary of Results by Benchmark. Each bar shows the improvement factor for a single benchmark, and a breakdown of that gain among the different optimisations. Only the uppermost path across the improvement lattice is used, so larger gains appear on the left of the figure. For some tests, there are combinations of optimisations that result in better performance than having all options enabled. This gain is displayed as dashed regions of the bars; the end of the black region is the improvement factor with all optimisations on and the end of the whole bar is the gain for the best combination.

Of the tests significantly affected by soft updates, most saw the elapsed time reduced by a factor of more than 2 with soft updates alone. Even benchmarks where the metadata changes are small (such as Bonnie++'s sequential data output tests) saw small improvements of around 1%.

Some of the read-only tests were slightly adversely affected by soft updates (Andrew's grep 0.7%, Andrew's stat 2.4%). This is most likely attributable to delayed writes from other phases of the tests. In a real-world read-only situation this should not be a problem, as there will be no writes to interact with reading.

As expected, dirpref shows up as an important factor in benchmarks involving directory traversal: Andrew, Buildworld, Netnews and X11 Archive. For the majority of these tests dirpref improves times by 10%–20%. Of all the individual tests within these benchmarks, the only one that is not noticeably affected is the compile time in the Andrew benchmark.

Though it involves no directory traversal, the performance of Postmark with 10,000 files does seem to be very weakly dependent on dirpref. The most likely explanation for this is that the directory used by Postmark resides in a different cylinder group when dirpref is used and the existing state of the cylinder group has some small effect.

The impact of vmiodir was much harder to spot. On both the Andrew compile phase, and the build and remove phases of Buildworld, there was an improvement of around 2%. This is worthy of note as buildworld was used as a benchmark when vmiodir was assessed.

Vmiodir and dirpref have an interesting interaction on the grep and remove phases of the Andrew benchmark. Both produced similar-sized effects and the effects show significant overlap. This must correspond to vmiodir and dirpref optimising the same caching issue associated with these tasks.

As expected, dirhash shows its effect in the benchmarks involving large directories. On the Folder benchmark, dirhash takes soft updates's already impressive figures ($\times 29$, $\times 13$ and $\times 77$ for create, pack and remove respectively) and improves them even further ($\times 12$, $\times 38$ and $\times 3$ respectively). However, without soft updates, dirhash's improvements are often lost in the noise. This may be because dirhash saves CPU time and writes may be overlapped with the use of the CPU when soft updates is not present.

There are improvements too for Bonnie++'s large directory tests showing increased rates and decreased %CPU usage. The improvements for randomly creating/reading/deleting files and sequentially creating/deleting files are quite clear. Only Bonnie++'s sequential file reads on a large directory failed to show clear improvements.

Some improvements were also visible in Postmark. The improvements were more pronounced with 20,000 files than with 10,000. When combined with soft updates, overall gains were about 1%–2%. Dirhash presumably would show larger benefits for larger file sets.

5 Conclusion

We have described soft updates, dirpref, vmiodir and dirhash and studied their effect on a variety of benchmarks. The micro-benchmarks focusing on individual tasks were often improved by two orders of magnitude using combinations of these optimisations.

The Postmark, Netnews and Buildworld benchmarks are most representative of real-world workloads. Although they do not show the sort of outlandish improvements seen in the earlier micro-benchmarks, they still show large improvements over unoptimised FFS.

Many of these optimisations trade memory for speed. It would be interesting to study how reducing the memory available to the system changes its performance.

We also studied dirhash's design and implementation in detail. Given the constraint of not changing the on-disk filesystem format, dirhash seems to provide constant-time directory operations in exchange for a relatively small amount of memory. Whereas dirhash might not be the ideal directory indexing system, it does offer a way out for those working with large directories on FFS.

References

[1] S. Best. JFS overview, January 2000, `http://www-106.ibm.com/developerworks/library/jfs.html`.

[2] Russell Coker. The bonnie++ benchmark, 1999, `http://www.coker.com.au/bonnie++/`.

[3] ISC et al. INN: InterNetNews, `http://www.isc.org/products/INN/`.

[4] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the USENIX 1994 Symposium on Operating Systems Design and Implementation*, pages 49–60, Monterey, CA, USA, 14–17 1994. `http://citeseer.nj.nec.com/ganger94metadata.html`.

[5] J. Katcher. Postmark: A new file system benchmark, October 1997, `http://www.netapp.com/tech_library/3022.html`.

[6] Donald E. Knuth. *The Art of Computer Programming: Volume 3 Sorting and Searching*. Addison Wesley, second edition, 1998.

[7] Marshall Kirk McKusick. Running fsck in the background. In *Usenix BSDCon 2002 Conference Proceedings*, pages 55–64, February 2002, `http://www.usenix.org/publications/library/proceedings/bsdcon02/mckusick.html`.

[8] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. pages 1–17, 1999, `http://citeseer.nj.nec.com/mckusick99soft.html`.

[9] Landon Curt Noll. Fowler / Noll / Vo (FNV) hash, `http://www.isthe.com/chongo/tech/comp/fnv/`.

[10] Grigoriy Orlov. Directory allocation algorithm for FFS, 2000, `http://www.ptci.ru/gluk/dirpref/old/dirpref.html`.

[11] Daniel Phillips. A directory index for Ext2. In *Proceedings of the Fifth Anual Linux Showcase and Conference*, November 2001, `http://www.linuxshowcase.org/phillips.html`.

[12] B. Rakitzis and A. Watson. Accelerated performance for large directories, `http://www.netapp.com/tech_library/3006.html`.

[13] H. Reiser. ReiserFS, 2001, `http://www.namesys.com/res_whol.shtml`.

[14] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: asynchronous meta-data protection in file systems. In *Proceedings of 2000 USENIX Annual Technical Conference*, June 2000, `http://www.usenix.org/publications/library/proceedings/usenix2000/general/seltzer.html`.

[15] K. Swartz. The brave little toaster meets usenet, October 1996, `http://www.usenix.org/publications/library/proceedings/lisa96/kls.html`.

[16] A. Sweeney, D. Doucette, C. Anderson W. Hu, M. Nishimoto, and G. Peck. Scalability in the XFS file system, January 1996, `http://www.usenix.org/publications/library/proceedings/sd96/sweeney.html`.

[17] Various. The XFree86 Project, Inc., `http://www.xfree86.org/`.

Results

The complete set of scripts, code, tables and graphs are available at `http://www.cnri.dit.ie/~dwmalone/ffsopt`.

Acknowledgments

The authors would like to thank: Robert Watson for his comments and advice; Chris Demetriou and Erez Zadok for providing us with last-minute shepherding; Cathal Walsh for advice on linear models. Some of the work in this paper was developed or tested at the School of Mathematics, Trinity College Dublin where both authors are members of the system administration team.