CHAPTER 9

ReiserFS for Linux

2

Linux File Systems

One of the earliest journaling file systems for Linux was ReiserFS. This project was started by a remarkable person, Hans Reiser, a young Ph.D. in computer science. Hans Reiser is a harsh critic of mediocrity in operating system design and software design in general. As a strong believer in the thorough implementation of excellent design, he felt compelled to start his project as proof of his ideas about file systems. The result, ReiserFS, is remarkable most of all for it's scientific and intellectual approach to design and programming. However, it's biggest shortcoming is its poor interaction with other Linux kernel components (like NFS, for instance). Still, ReiserFS has now been in use for a few years and many users are quite satisfied with it, for example, the mp3.com Web site. The SuSE distribution was 'ReiserFS's earliest adopter and has been supporting it since release 6.2 of SuSE Linux.

The main goal of ReiserFS is to provide reasonably fast recovery after a crash, and to have transactional file system meta-data updates. ReiserFS is fast, especially for small files and also for directories with many files in them. One of the future goals is to have write-ahead logging at least as fast as the original file system with preserve lists turned on. In this chapter we will get to know the details of the ReiserFS design and concepts as well as more practical issues, such as installation, administration, etc.

THE FILE SYSTEM NAME SPACE

One of ReiserFS's central concepts is the unified name space. Ideally, Hans Reiser would like to create a file system composed of objects that are both 'files' and directories'.

In this example, bmoshe is a user, bmoshe/mail is an inbox, bmoshe/mail/Message-ID/20000615091245.A2500@moshebar.com is an e-mail, and bmoshe/mail/Message-ID/20000615091245.A2500@moshebar.com/to might be the To: field of the header of that e-mail. Combined with groupings, I should be able to look for:

mail/[from/Avivit phones]

to find all of Avivit's e-mails on phones.

Such a unified and closed name space potentially has a big impact on the ease of programming, especially object-oriented (OO) programming. As you can see from the line below, every object can naturally be thought of as it's own NamingContext, with its fields/accessor and mutator methods as the subnames. So, to paraphrase the above example:

moshe.getMailBox().getMessageByID("20000615091245.A2500@moshebar.com")

Those familiar with the Plan 9 and Inferno operating systems might find similarities, and rightly so. A unified name space, such as that proposed by Hans Reiser, is nothing but an "everything-is-a-file" concept, turned upside down.

There are, however, still some conceptual anomalies in his proposed abstraction of a file system. First of all, the problem with relying on attributes is that they break closure,

since attributes are not objects. In fact, they are not even object fields (although they may be exposures of such fields).

Unifying Reiser's groupings and orderings may prove a difficult task.As Reiser puts it, hypersemantics attempts "to pick a manageably few columns which cover all possible needs. Generalization, aggregation, classification, and membership correspond to the is-a, has-property, is-an-instance-of, and is-a-member-of columns, respectively." The problem, Reiser pointed out, is that you must know the relationship in order to be able to search for it. Using Reiser's example, you can't find Santa Claus without reindeer, unless you know how to decompose the propulsion-provider-for relationship into the above canonical relationships.

So, as you can see, name space unification and closure still remain to be explored. As such, using ReiserFS for a production system might prove reliable and efficient, but you will still be using a proof-of-concept file system or a research-and-development tool, and not a product by any means.

Let's now explore some more of 'ReiserFS's technical design concepts.

BLOCK ALIGNMENTS OF FILE BOUNDARIES

ReiserFS aligns file boundaries with blocks on the disk. It does so for a number of reasons: -to minimize the number of blocks a file is spread across (which is especially beneficial for multiple block files when locality of reference across files is poor); to avoid wasting disk and buffer space in storing every less-than-fully-packed block; to not waste I/O bandwidth with every access to a less-than-fully-packed block when locality of reference is present; to decrease the average number of block fetches required to access every file in a directory; and it results in simpler code.

The simpler code of block aligning file systems follows from not needing to create a layering to distinguish the units of the disk controller and buffering algorithms from the units of space allocation, and also from not having to optimize the packing of nodes, as is done in balanced tree algorithms.

Hans Reiser tried from the beginning to aggregate small files in a way so as to avoid wasting disk space. The simplest solution was to aggregate all small files in a directory, into either a file or the directory. But any aggregation into a file or directory wastes part of the last block in the aggregation. What does one do if there are only a few small files in a directory—aggregate them into the parent of the directory? What if there are only a few small files in a directory at first, and then there are many small files—how does the OS decide what level to aggregate them at, and when to take them back from a parent of a directory and store them directly in the directory?

Of course, this problem is closely related to the balancing of nodes in a balanced tree. The balanced tree approach, by using an ordering of files which are then dynamically aggregated into nodes at a lower level, rather than a static aggregation or grouping, avoids this set of questions.

In the ReiserFS approach, both files and filenames are stored in a balanced tree. This, along with small files, directory entries, inodes, and the tail ends of large files causes ev-

The closing single quote at the end of para OK?

erything to be more efficiently packed as a result of relaxing the requirements of block alignment, and eliminating the use of a fixed space allocation for inodes.

The body of large files is stored in unformatted nodes that are attached to the tree but are isolated from the effects of possible shifting by the balancing algorithms. Neither NTFS nor XFS aggregate files, they block align files, although they do store small files in the statically allocated block address fields of inodes if they are small enough to fit there.

Semantics (files), packing (blocks/nodes), caching (read-ahead sizes, etc.), and the hardware interfaces of disks (sectors) and paging (pages) all have different granularity issues associated with them. Understand that the optimal granularity of these often differs, and abstracting them into separate layers in which the granularity of one layer does not unintentionally impact other layers can improve space/time performance. ReiserFS innovates in that its semantic layer often conveys an un-granulated ordering to the other layers rather than one granulated by file boundaries. While reading the algorithms of ReiserFS's code, the reader is encouraged to note the areas in which ReiserFS needs to go further. '

BALANCED TREES AND LARGE FILE I/O

It is quite complex to understand the interplay between I/O efficiency and block size for larger files, and space does not allow a systematic review of traditional approaches. ReiserFS has the following architectural weaknesses that stem directly from the overhead of repacking to save space and increase block size:

- When the tail (files less than 4K are all tail) of a file grows large enough to occupy an entire node by itself, it is removed from the formatted node(s) it resides in, and converted into an unformatted node.
- A tail that is smaller than one node may be spread across two nodes, which requires more I/O to read if the locality of reference is poor.
- Aggregating multiple tails into one node introduces the separation of the file body from tail, which reduces read performance. For ReiserFS files near to the node in size the effect can be significant.
- When you add one byte to a file or tail that is not the last item in a formatted node, then on average half of the whole node is shifted in memory. If any of your applications perform I/O in such a way that they generate many small unbuffered writes, ReiserFS will make you pay a high price for not being able to buffer the I/O.

Most applications that create substantial file system load employ effective I/O buffering, often simply as a result of using the I/O functions in the standard C libraries.

By avoiding accesses in small blocks/extents, ReiserFS improves I/O efficiency. Extent-based file systems such as VFS, and write-clustering systems such as ext2fs, are not so effective in applying these techniques that they choose to use 512-byte blocks rather than

1K blocks as their defaults. Ext2fs reports a 20 percent speed-up when 4K rather than 1K blocks are used, but the authors of ext2fs advise the use of 1K blocks to avoid wasting space.

There are a number of worthwhile large file optimizations that have not been added to either ext2fs or ReiserFS, and both file systems are somewhat primitive in this regard, rReiserFS being the more primitive of the two. Large files simply were not my research focus, and this being a small research project, I did not implement the many well-known techniques for enhancing large file I/O. The buffering algorithms are probably more crucial than any other component in large file I/O, and partly out of a desire for a fair comparison of the approaches, I have not modified these. No significant optimizations for large files have been devised in ReiserFS, beyond increasing the block size. Except for the size of the blocks, there is not a large inherent difference between:

- ▼ The cost of adding a pointer to an unformatted node to a tree plus writing the node.
- ▲ Adding an address field to an inode plus writing the block. It is likely that, except for block size, the primary determinants of high performance large file access are orthogonal to the decision of whether to use balanced tree algorithms for small and medium sized files.

For large files there is an advantage to not having the tree more balanced than the tree formed by an inode which points to a triple indirect block. There is performance overhead due to the memory bandwidth cost of balancing nodes for small files.

(2)Serialization and Consistency

The issue of ensuring recoverability with minimal serialization and data displacement inevitably dominate high performance design. Let's define the two extremes in serialization so that the reason for this can be made clear. Consider the relative speed of a set of I/O's in which every block request in the set is fed to the elevator algorithms[1] of the kernel and serially to the disk drive firmware , each request awaiting the completion of the previous request. Now consider the other extreme, in which all block requests are fed to the elevator algorithms together, so that they are all sorted and performed close to their sorted order. The un-serialized extreme may be an order of magnitude faster due to the cost of rotations and seeks. Unnecessarily serializing I/O prevents the elevator algorithm from doing its job of placing all of the I/O's in their layout sequence rather than chronological sequence. Most high performance design centers around making I/O's in the order they are laid out on disk, and in the order that the I/O's will want to be issued.

ReiserFS employs a new scheme called `preserve lists` for ensuring recoverability, which avoids overwriting old meta-data by writing the meta-data nearby.

1 The elevator algorithms within the kernel schedule I/O operations to and from the disk(s).

Tree Definitions

Balanced trees are designed with a set of keys assumed to be defined by the application, and the purpose of the tree design is to optimize searching with these keys. In ReiserFS, the purpose of the tree is to optimize the reference locality and space-efficient packing of objects, and the keys are defined according to that algorithm. Keys are used in place of inode numbers in the file system, thus substituting the mapping of keys to a node location (the internal nodes) rather than a mapping of an inode number to a file location. Keys are longer than inode numbers, but o fewer of them need to be cached when more than one file is stored in a node.

ReiserFS trees require that a filename be resolved one component at a time. It is an interesting topic for future research whether this is necessary or optimal. It is a more complex issue than one might realize. Directory at a time lookup accomplishes a form of compression, makes mounting other name spaces and file system extensions simpler, makes security simpler, and makes future enhanced semantics simpler. Since small files typically lead to large directories, it is fortunate that, as a natural consequence of our use of tree algorithms, our directory mechanisms are much more effective for very large directories than most other file systems. The tree has three node types: internal nodes, formatted nodes, and unformatted nodes. The contents of internal and formatted nodes are sorted in the order of their keys. (Unformatted nodes contain no keys.)

Internal nodes consist of pointers to subtrees separated by their delimiting keys. The key that precedes a pointer to a subtree is a duplicate of the first key in the first formatted node of that subtree. Internal nodes exist solely to allow a determination of which formatted node contains the item corresponding to a key. ReiserFS starts at the root node, examines its contents and can determine which subtree contains the item corresponding to the desired key. From the root node ReiserFS descends into the tree, branching at each node, until it reaches the formatted node containing the desired item.

The first (bottom) level of the tree consists of unformatted nodes, the second level consists of formatted nodes, and all levels above consist of internal nodes. The highest level contains the root node. The number of levels is increased as needed by adding a new root node at the top of the tree.

All paths from the root of the tree to all formatted leaves are equal in length. The paths to unformatted leaves are also equal in length but are one node longer than paths to formatted leaves. This equality in path length, and the high fanout it provides is vital to high performance.

Formatted nodes consist of items, which are of four types: direct, indirect, directory, and stat data items. All items contain a unique key that is used to sort and find the item. Direct items contain the tails of files, which are the last part of the file (the last file_size modulo FS block size of a file). Indirect items consist of pointers to unformatted nodes. All but the tail of the file is contained in the unformatted nodes. Directory items contain the key of the first directory entry in the item followed by a number of directory entries.

A file consists of a set of indirect items followed by a set of up to two direct items, with the two direct items representing an example of a tail split across two nodes. If a tail is larger than the maximum size of a file that can fit into a formatted node, but is smaller

than the unformatted node size (4K), then it is stored in an unformatted node, and a pointer to it, plus a count of the space used, is stored in an indirect item.

Directories consist of a set of directory items, which, in turn,consist of a set of directory entries. Directory entries contain the filename and the key of the file which is named. There is never more than one item of the same item type from the same object stored in a single node (there is no reason one would want to use two separate items rather than combining). The first item of a file or directory contains its stat data.

When performing balancing, and when analyzing the packing of the node and its two neighbors, ReiserFS ensures that the three nodes cannot be compressed into two nodes.

Ordering Within the Tree

Some key definition decisions depend on usage patterns, and this means that someday one will select from several key definitions when creating the file system. For example, consider the decision of whether to pack all directory entries together at the front of the file system, or to pack the entries near the files they name. For large file usage patterns one should pack all directory items together, since systems with such usage patterns are effective in caching the entries for all directories. For small files the name should be near the file. Similarly, for large files the stat data should be stored separately from the body, either with the other stat data from the same directory, or with the directory entry.

It is feasible to pack an object completely independently of its semantics using these algorithms, and there may be applications for which a packing different than that determined by object names is more appropriate.

The Structure of a Key

Each file item has a key with a structure: locality_id, object_id, offset, and uniqueness. The locality_id is, by default, the object_id of the parent directory. The object_id is the unique id of the file, and is set to the first unused object_id when the object is created. This can result in successive object creations in a directory being adjacently packed which is advantageous for many usage patterns. For files, the offset is within the logical object of the first byte of the item. In version 0.2 all directory entries had their own individual keys stored with them and each were distinct items.In the current version, ReiserFS stores one key in the item that is the key of the first entry, and computes each entry's key as needed from this one key. For directories, the offset key component is the first four bytes of the filename; you may think of this as a lexicographic rather than numeric offset. For directory items, the uniqueness field differentiates identical filename entries in the first four bytes. For all items it indicates the item type and for the leftmost item in a buffer, it indicates whether the preceding item in the tree is of the same type and object. Placing this information in the key is useful when analyzing balancing conditions, but increases key length for non-directory items, and is a questionable architectural feature.

Every file has a unique object_id, but this cannot be used for finding the object, only keys are used for that. Object_ids merely ensure that keys are unique. If you never use the ReiserFS features that change an object's key then it is immutable, otherwise it is mutable. (This feature aids support for NFS daemons, etc.) Developers spent some time debating

whether the use of mutable keys for identifying an object had deleterious long-term architectural consequences. In the end, Hans Reiser decided it was acceptable if there was a requirement that any object recording a key should possess a method for updating the copy of itself.

This is the architectural price of avoiding having to cache a map of the object_id to location that might have very poor locality of reference due to object_ids not changing with object semantics. ReiserFS packs an object with the packing locality of the directory it was first created in unless the key is explicitly changed. It remains packed there even if it is unlinked from the directory. It will not move it from the locality where it was created without an explicit request, unlike some other file systems which store all multiple link files together and pay the cost of moving them from their original locations when a second link occurs. A file linked with multiple directories should at least get the locality reference benefits of one of those directories.

In summary, this approach first places files from the same directory together, and then places directory entries from the same directory together, and with the stat data for the directory. Note that there is no interleaving of objects from different directories in the ordering, and that all directory entries from the same directory are contiguous. This does not actually pack the files of small directories with common parents together, and does not employ the full partial ordering in determining the linear ordering. It merely uses parent directory information. The appropriate place for employing full tree structure knowledge is in the implementation of an FS cleaner, not in the dynamic algorithms.

The balancing of nodes in the tree happens according to the following ordered priorities:

1. Minimize the number of nodes used.
2. Minimize the number of nodes affected by the balancing operation.
3. Minimize the number of uncached nodes affected by the balancing operation.
4. If shifting to another formatted node is necessary, maximize the bytes shifted by priority.
5. Is based on the assumption that the location of an insertion of bytes into the tree is an indication of the likely future location of an insertion, and that policy 4 will, on average, reduce the number of formatted nodes affected by future balance operations.

There are also more subtle effects. If you randomly place nodes next to each other, and have a choice between those nodes being somewhat efficiently packed or packed to an extreme—either well or poorly packed—you are more likely to be able to combine more of the nodes if you choose the policy of extremism. Extremism is a virtue in space-efficient node packing. The maximum shift policy is not applied to internal nodes, as extremism is not a virtue in time-efficient internal node balancing.

BUFFERING AND THE PRESERVE LIST

Version 0.2 of ReiserFS implemented a system of write ordering that tracked all shifting of items in the tree. It ensured that no node that an item had been shifted from was written before the node that had received the item was written. This is necessary as it will prevent a system crash causing the loss of an item that might not be recently created. This tracking approach worked, and the overhead it imposed was not measurable by our benchmarks. When, in the next version, we changed to partially shifting items and increased the number of item types, this code grew out of control in its complexity. I decided to replace it with a scheme that was far simpler to code and was also more effective in typical usage patterns. This scheme is as follows.

If an item is shifted from a node, change the block that its buffer will be written to. Change it to the nearest free block to the old block's left neighbor, and rather than freeing it, place the old block number on a "preserve list''. (Saying nearest is slightly simplistic, in that the blocknr assignment function moves from the left neighbor in the direction of increasing block numbers.) When a "moment of consistency'' is achieved, free all of the blocks on the preserve list. A moment of consistency occurs when there are no nodes in memory into which objects have been shifted. If disk space runs out, force a moment of consistency to occur. This is sufficient to ensure that the file system is recoverable. Note that during the large file benchmarks, the preserve list was freed several times in the middle of the benchmark. The percentage of buffers preserved is small in practice except during deletes, and you can arrange for moments of consistency to occur as frequently as necessary.

This approach may not be better than the Soft Updates approach of BSD or by ReiserFS in version 0.2. However, those tracking orders of writes are more complex than this approach for balanced trees, which partially shifts items. ReiserFS might shift back to the old algorithm in the future, however, as preserve lists substantially hamper performance for files in the 1-10K size range.

ReiserFS Structures

The ReiserFS tree has a maxmum tree hight, called Max_Height = N (current default value for N = 5). The tree resides in the disk blocks. Each disk block that belongs to the ReiserFS tree has a block head.

Everything in the file system is stored as a set of items. Each item has its item_head. The item_head contains the key of the item, its free space (for indirect items), and specifies the location of the item itself within the block.

The disk block containing the (internal node of the tree is the place for keys and pointers to disk blocks looks like this:

Block_Head	Key 0	Key 1	Key 2	--	Key N	Pointer 0	Pointer 1	Pointer 2	--	Pointer N	Pointer N+1	……..Free Space……..

Within the tree, each leaf—each with *n* items and their corresponding headers, has a corresponding disk block with the following:

Block_Head	IHead 0	IHead 1	IHead 2	--	IHead N	Pointer 0	……..Free Space……..	Item N	--	Item 2	Item 1	Item 0

There are also disk blocks containing an unformatted node of the above-mentioned tree. These kinds of disk blocks contain data and thus look structurally empty from the outside (although they may contain data):

……...

The maximum number of objects in a ReiserFS namespace (including files and directories) are calculated this way:

2^32-4

which equals the maximum number of 4,294,967,292.

Internal Node Structures In the following table, you can find the structure of the inode block as stored on disk. The ReiserFS inode is just one node of the ReiserFS tree, storing keys and pointers to disks data blocks:

Block_Head	Key 0	Key 1	Key 2	--	Key N	Pointer 0	Pointer 1	Pointer 2	--	Pointer N	Pointer N+1	……..Free Space……..

Here you get the description of the *key* structure; notice how all variable are 32bit in size:

Field Name	Type	Size (in bytes)	Description
k_dir_id	__u32	4	ID of the parent directory
k_object_id	__u32	4	ID of the object (also it is the number of inode)
k_offset	__u32	4	Offset from beginning of the object to the current byte of the object
k_uniqueness	__u32	4	Type of the item (Stat Data = 0, direct = -1, InDirect = -2, Directory = 500)
	total	**16**	**(6) 8 bytes for internal nodes; (22) 24 bytes for leaf nodes**

Finally, here is the disk_child structure, which is the actual pointer to the disk block:

Field Name	Type	Size (in bytes)	Description
dc_block_number	unsigned long	4	Disk child's block number.
dc_size	unsigned short	2	Disk child's used space.
	total	**6**	**(6) 8 bytes**

Leaf Node Structures Now, we continue analyzing the disk block, which is a node of the ReiserFS tree and stores items and their headers, as shown in the following table. There are four types of items: stat data item, directory item, indirect item, and direct item, which are in this case self-explanatory.

Block_Head	IHead 0	IHead 1	IHead 2	--	IHead N	Pointer 0	……..Free Space……..	Item N	--	Item 2	Item 1	Item 0

Again, we go through the individual objects, starting with the block_head structure in the disk block:

	Type	Size (in bytes)	Description
blk_level	unsigned short	2	Level of block in the tree (1-leaf; 2,3, 4,…-internal
blk_nr_item	unsigned short	2	Number of keys in an Internal block. Or number of items in a leaf block
blk_free_space	unsigned short	2	Block free space in bytes
blk_right_delim_key	struct key	16	Right delimiting key for this block (for Leaf nodes only)
	total	**22**	**(22) 24 bytes for leaf nodes**

Each item head contains various variables that allow the item to know its exact position in the item tree and get some information about the space used, as well as free space left in the item.

Field Name	Type	Size (in bytes)	Description
ih_key	struct key	16	Key to search the item. All item headers is sorted by this key
u.ih_free_space u.ih_entry_count	__u16	2	Free space in the last unformatted node for an indirect item; 0xFFFF for a sirect item; 0xFFFF for a stat data item. The number of directory entries for a directory item.
ih_item_len	__u16	2	Total size of the item body
ih_item_location	__u16	2	An offset to the item body within the block
ih_reserved	__u16	2	Used by reiserfsck
	total	24	**24 bytes**
sd_mode	__u16	2	File type, permissions
sd_nlink	__u16	2	Number of hard links
sd_uid	__u16	2	Owner id
sd_gid	__u16	2	Group id

Field Name	Type	Size (in bytes)	Description
sd_size	__u32	4	File size
sd_atime	__u32	4	Time of last access
sd_mtime	__u32	4	Time file was last modified
sd_ctime	__u32	4	Time inode (stat data) was last changed (except changes to sd_atime and sd_mtime)
sd_rdev	__u32	4	Device
sd_first_direct_byte	__u32	4	Offset from the beginning of the file to the first byte of direct item of the file. (-1) for directory (1) for small files (file has direct items only) (>1) for big files (file has indirect and direct items) (-1) for big files (file has indirect, but has not direct item)
	total	**32**	**32 bytes**

The directory object just contains filenames, which can be either small files or big files:

deHead 0	deHead 1	deHead 2	—	deHead N	**filename N**	—	**filename 2**	**filename 1**	**filename 0**

The small file is called the *direct item*, because it is addressable by just one pointer:

……Small File Body……

Bigger files (those that require more than one disk block), need some pointer acrobatics to find all the subsequent blocks, and are therefore called *indirect items*:

unfPointer 0	unfPointer 1	unfPointer 2	—	unfPointer N

This needs some explanation. The unfPointer is a pointer (32bits) to an unformatted block containing the body of a big file. In the following table you see how the pointers find that unformatted block:**Field Name**	**Type**	**Size (in bytes)**	**Description**
deh_offset	__u32	4	Third component of the directory entry key (all reiserfs_de_head sorted by this value)

deh_dir_id	__u32	4	Object_id of the parent directory of the object, that is referenced by directory entry
deh_objectid	__u32	4	Object_id of the object that is referenced by directory entry
deh_location	__u16	2	Offset of name in the whole item
deh_state	__u16	2	1) Entry contains stat data (for future) 2) Entry is hidden (unlinked)
	total	**16**	**16 bytes**

Filename here represents the name of the file (array of bytes of variable length). The maximum length of filename = blocksize - 64 (for a 4K blocksize, the maximum name length is 4032 bytes).

USING THE TREE TO OPTIMIZE LAYOUT OF FILES

There are four levels at which layout optimization is performed:

- The mapping of logical block numbers to physical locations on disk.
- The assigning of nodes to logical block numbers.
- The ordering of objects within the tree.
- The balancing of the objects across the nodes they are packed into.

Physical Layout

The mapping of logical block numbers to physical locations on the disk is performed by the disk drive manufacturer for SCSI, and by the device driver for IDE drives. There can, of course, be a higher level of software like LVM, discussed in a previous chapter, which abstracts that mapping further. The logical block number to physical location mapping by the drive manufacturer is usually done using cylinders. The ReiserFS developers found that minimizing the distance in logical blocks of semantically adjacent nodes without tracking cylinder boundaries accomplishes an excellent approximation of optimizing according to actual cylinder boundaries. That simplicity also makes for a more elegant implementation.

Node Layout

When ReiserFS places nodes of the tree on the disk, it searches for the first empty block in the bitmap of used block numbers, which it finds by starting at the location of the left neighbour of the node in the tree ordering, and then moves in the direction it last moved in.

This was found, by experimentation, to be better than the following alternatives for the benchmarks:

1. Taking the first non-zero entry in the bitmap.
2. Taking the entry after the last one that was assigned in the direction last moved in (this was three percent faster for writes and 10-20% slower for subsequent reads).
3. Starting at the left neighbor and moving in the direction of the right neighbor. When changing block numbers for the purpose of avoiding the overwriting of sending nodes before shifted items reach the disk in their new recipient node (see description of preserve lists), the benchmarks employed were approximately ten percent faster than when starting the search from the left neighbor rather than the node's current block number, even though it adds significant overhead to determine the left neighbor (the current implementation risks I/O to read the parent of the left neighbor).

It used to be that ReiserFS could reverse direction when the end of the disk drive was reached. The developers checked to see if it mada a difference which direction one moves in when allocating blocks to a file, and found it made a significant difference to always allocate in the increasing block number direction. This may be due to matching disk spin direction by allocating increasing block numbers.

Write-Ahead Logging

Most meta-data operations involve more than one block, and meta-data will usually be corrupted if only some of the blocks involved in an operation get updated. With write-ahead logging, blocks are written to a log before they are allowed to hit their real locations. After a crash, the log is replayed to bring the file system back to a consistent state. This replay is much faster than an fsck, and its time is bounded by the size of the log area instead of the size of the file system.

ReiserFS Journaling Features

Journaling requires some kind of logging and the serialization of that locking. That is why a journaled file system is necessarily slower than a nonlogging counterpart. Therefore, the journaling requires some kind of fundamental operation to perform that logging and serialization. Let's look at what these are in ReiserFS.

Transactions Each transaction in the journal consists of a description block, followed by a number of data blocks and a commit block. The description and commit blocks contain a sequential list of the real disk locations for each log block. The log must preserve the order of updates, so if a writer logs blocks A, B, C, and D, and then A again, they will be ordered in the log as B, C, D, A.

While a block is in an uncommitted transaction, it must remain clean, and must have a reference count of at least one. Once a transaction has all its log blocks on disk, the real buffers are dirtied and released.

Batched Transactions I allow multiple transactions to be combined into a single atomic unit. So if transaction one logs blocks A, B, C and transaction two logs blocks A, C, and D, the resulting joined transaction would write a description block, then blocks B, A, C, and D, and then a commit block. This allows fewer total blocks to be written to the log, but increases the chance of file system changes being undone by a crash. There are a number of tuning parameters to control how and when transactions are batched together. Take a look at the tuning section for the details.

Asynchronous Commits This is an extension of the batched transactions. I allow a transaction to end without flushing all of its log blocks to disk. This adds a great deal of complexity, but makes it possible for operations to return faster, and release any locks they might hold. Bdflush takes care of forcing old asynchronous log blocks to disk.

New Blocks Can Die in the Cache If a block is allocated and then freed before being written to disk, or logged and then freed before its transaction is completed, the block is never written to the log or its real disk location. This is inherent in many file systems, but took a little work to get right in ReiserFS.

Selective Flushing This is actually more of a requirement than a feature. Many blocks (bitmaps, super-blocks, etc) tend to get logged over and over again. When a block is in an uncommitted transaction, it can't be dirtied, and can't be sent to disk. But before a log area can be reused, any transactions contained in it must have all their blocks flushed to their real locations.

Even if a multiple logged block could be flushed somehow, it has changed in relation to all the other blocks in the older transaction, and the meta-data could be corrupted after a crash. Instead of trying to flush blocks that will also be in future transactions, I force the future transaction's' log blocks to disk. After a crash, log replay should make everything consistent.

This means that frequently logged blocks might only get written once per transaction to the log, and then once to their real location on file system unmount.

Data Block Logging 'Data blocks are logged when they are part of a direct to indirect item conversion. These conversions are done when small files grow beyond what can fit in a direct item, and on the mmap of files that contains direct items. Since the conversion is sometimes done to old data, I want to make sure the data won't be lost after a crash.

The problem is that once a block is in the log, you must continue logging it while there is any chance log replay will overwrite the block after a crash. So, mark_buffer_dirty is never called directly. Instead, a journal call exists to only log a block if it is in the current transaction, or in a transaction that might be replayed.

This is a performance hit on average size files with tails enabled, because many of the data blocks will be logged. Possible solutions include implementing packing on file close, or mounting without tails enabled. The ReiserFS team will probably be looking into these and other ideas over the next few months.

Tuning

The size of the log area has the biggest effect on your performance. Make it too small, and you will have to flush blocks to their real locations too frequently. Make it too large and your replay times will be much too long.

How do you find the right size? Well, in beta1, you need to try a many different ones until your benchmark gets as fast as it's going to get. Beta2 will have a mount option to add informational statements while flushing the real blocks. These should make tuning much easier.

The max transaction size, max batch size, and various time limits for how old things can get are also very important. It will be a while before I really have the chance to explore these.

ReiserFS Drops

Consider dividing a file or directory into drops, with each drop having a separate key, and no two drops from one file or directory occupying the same node without being compressed into one drop. The key for each drop is set to the key for the object (file or directory) plus the offset of the drop within the object. For directories the offset is lexicographic and by filename, for files it is numeric and in bytes. In the course of several file system versions we have experimented with and implemented solid, liquid, and air drops. Solid drops were never shifted, and drops would only solidify when they occupied the entirety of a formatted node. Liquid drops are shifted in such a way that any liquid drop which spans a node fully occupies the space in its node. Like a physical liquid it is shiftable, but not compressible. Air drops merely meet the balancing condition of the tree.

ReiserFS 0.2 implemented solid drops for all but the tail of files. If a file was at least one node in size it would align the start of the file with the start of a node, block-aligning the file. This block alignment of the start of multi-drop files was a design error that wasted space. Even if the locality of reference is so poor as to make one not want to read parts of semantically adjacent files, if the nodes are near to each other then the cost of reading an extra block is thoroughly dwarfed by the cost of the seek and rotation to reach the first node of the file. As a result the block alignment saves little in time, though the cost is significant space for 4-20K files.

ReiserFS with block alignment of multi-drop files and no indirect items experienced the following rather interesting behavior that was partially responsible for making it only 88% space-efficient for files that averaged 13K (the Linux kernel) in size. When the tail of a larger than 4K file was followed in the tree ordering by another file larger than 4K, since the drop before was solid and aligned, and the drop afterwards was solid and aligned, no matter what size the tail was, it occupied an entire node.

In the current version we place all but the tail of large files into a level of the tree reserved for full unformatted nodes, and create indirect items in the formatted nodes which point to the unformatted nodes. This is known in the database literature as the approach. This extra level added to the tree comes at the cost of making the tree less balanced (I consider the unformatted nodes pointed to as part of the tree) and increasing the maximum depth of the tree by one. For medium-sized files, the use of indirect items increases the

cost of caching pointers by mixing data with them. The reduction in fanout often causes the read algorithms to fetch only one node at a time, as one waits to read the uncached indirect item before reading the node with the file data. There are more parents per file read with the use of indirect items than with internal nodes, as a direct result of reduced fanout due to mixing tails and indirect items in the node. The most serious flaw is that these reads of various nodes, necessary to the reading of the file, have additional rotations and seeks compared to drops. With my initial drop approach they are usually sequential in their disk layout, even the tail, and the internal node parent points to all of them in such a way that all of them that are contained by that parent or another internal node in cache can be requested at once in one sequential read. Non-sequential reads of nodes are more costly than sequential reads, and this single consideration dominates effective read optimization.

Unformatted nodes make file system recovery faster and less robust, in that one reads their indirect item rather than insert them into the recovered tree, and one cannot read them to confirm that their contents are from the file that an indirect item says they are from. In this, they make ReiserFS similar to an inode-based system without logging.

A moderately better solution would have simply eliminated the requirement for placement of the start of multi-node files at the start of nodes, rather than introducing BLOBs, and to have depended on the use of a file system cleaner to optimally pack the 80% of files that don't move frequently, using algorithms that move even solid drops. Yet that still leaves the problem of formatted nodes not being efficient for mmap() purposes (one must copy them before writing rather than merely modifying their page table entries, and memory bandwidth is expensive even if the CPU is cheap).

For this reason I have the following plan for the next version. I will have three trees: one tree maps keys to unformatted nodes, one tree maps keys to formatted nodes, and one tree maps keys to directory entries and stat data. This would seem to mean that to read a file and first access the directory entry and stat data, the unformatted node, and then the tail, one must hop long distances across the disk, going first to one tree and then the other It took me two years to realize that it could be made to work. My plan is to interleave the nodes of the three trees according to the following algorithm:

Block numbers are assigned to nodes when the nodes are created, or preserved, and someday will be assigned when the cleaner runs. The choice of block number is based on first determining what other node it should be placed near, and then finding the nearest free block t in the elevator's current direction. Currently we use the left neighbor of the node in the tree as the node it should be placed near.

The new scheme will continue to first determine the node it should be placed near, and then start the search for an empty block from that spot, but it will use a more complicated determination of what node to place it near. This new method will cause all nodes from the same packing locality to be near each other, will cause all directory entries and stat data to be grouped together within that packing locality, and will interleave formatted and unformatted nodes from the same packing locality. Pseudo-code is best for describing this:

```
/* for use by reiserfs_get_new_blocknrs when determining where in the bitmap to
start the search for a free block, and for use by read-ahead algorithm when
there are not enough nodes to the right and in the same packing locality for
packing locality reading ahead purposes */
get_logical_layout_left_neighbors_blocknr(key of current node)
{
/* Based on examination of current node key and type, find the virtual neighbor
of that node. */
    If body node
        if first body node of file
             if (node in tail tree whose key is less but is in same packing
locality exists)
                 return blocknr of such node with largest key
             else
                 find node with largest key less than key of current node in
stat_data tree
                    return its blocknr
        else
             return blocknr of node in body tree with largest key less than key
of current node
    else
        if tail node
            if (node in body tree belonging to same file as first tail of
current node exists)
                 return its blocknr
            else if (node in tail tree with lesser delimiting key but same
packing locality exists)
                     return blocknr of such node with largest delimiting key
           else
                 return blocknr of node with largest key less than key of
current node in stat_data tree
    else /* is stat_data tree node */
        if stat_data node with lesser key from same packing locality exists
            return blocknr of such node with largest key
        else /* no node from same packing locality with lesser key exists */
}
/* for use by packing locality read-ahead */
get_logical_layout_right_neighbors_blocknr(key of current node)
{
    right-handed version of get_logical_layout_left_neighbors_blocknr logic
}
```

Code Complexity

I thought it appropriate to mention some of the notable effects of simple design decisions on our implementation's code length. When we changed our balancing algorithms to shift parts of items rather than only whole items, so as to pack nodes tighter, this had an impact on code complexity. Another multiplicative determinant of balancing code complexity was the number of item types. Introducing indirect items doubled this, and

changing directory items from liquid drops to air drops also increased it. Storing stat data in the first direct or indirect item of the file complicated the code for processing those items more than if I had made stat data its own item type.

When one finds oneself with an NxN coding complexity issue, it usually indicates the need for adding a layer of abstraction. The NxN effect of the number of items on balancing code complexity is an instance of that design principle, and we will address it in the next major rewrite. The balancing code will employ a set of item operations which all item types must support. The balancing code will then invoke those operations without needing to understand any more of the meaning of an item's type than it determines which item-specific item operation handler is called. Adding a new item type, e.g., a compressed item, will then merely require writing a set of item operations for that item rather than requiring a modicication of most parts of the balancing code as it does now.

We now feel that the function to determine what resources are needed to perform a balancing operation, fix_nodes(), might as well be written to decide what operations will be performed during balancing since it pretty much has to do so anyway. That way, the function that performs the balancing with the nodes locked, do_balance(), can be gutted of most of its complexity.

INSTALLING AND CONFIGURING REISERFS ON A LINUX KERNEL

The ReiserFS file system is quite easy to install. From Linux kernel 2.4.3, Linus Torvalds included ReiserFS in the standard Linux source. This means for newer kernels you don't need to do anything to the kernel source; it is ready to be compiled with ReiserFS turned on. For older kernels there is a somewhat tedious procedure to follow to obtain a patch from the www.namesys.com Web site and then apply the patch to the standard Linux source code.

Linux-2.2.X Kernels

For Linux-2.2.X kernels, follow these steps:

1. Get the latest ReiserFS patch from one of our mirrors.
   Suppose you get linux-2.2.19-reiserfs-3.5.32-patch.bz2,
   put it somewhere, for example: /usr/src/2.2.19/
2. Get the kernel sources of 2.2.19 on http://www.kernel.org.
   Put it somewhere, for example: /usr/src/2.2.19/linux.
   Now if you perform "ls" in /usr/src/2.2.19/,
   the result will look like:
   # ls
   linux linux-2.2.19-reiserfs-3.5.32-patch.bz2

3. Apply ReiserFS patch to it:
   # cd /usr/src/2.2.19
   # bzcat linux-2.2.19-reiserfs-3.5.32-patch.bz2 | patch -p0
4. Compile the linux-kernel, set rReiserFS support:
   # cd /usr/src/2.2.19/linux
   # make mrproper; make menuconfig
5. Set rReiserFS support here.You might need to turn on experimental features, depending on the exact kernel you are using:
   # make dep; make bzImage
6. When configuring, say y or n on ReiserFS support question. Read our Configuration Web page. If you set rReiserFS as a module, please also do the following:
   # make modules
   # make modules_install

   If you upgrade rReiserFS sometime later, don't think that you only have to recompile the module. It is a nice theory t, but not a reality, mainly because interfaces to the file system are rapidly changing all of the time.

   Bugs due to recompiling only the module tend to be completely cryptic, and the developers know it is because you didn't recompile the whole because somebody else already made that error.

   The kernel image will be in:
   /usr/src/2.2.19/linux/arch/i386/boot/bzImage
7. Compile and install the ReiserFS utils:
   # cd /usr/src/2.2.19/linux/fs/reiserfs/utils
   # make ; make install
8. Copy a new Linux kernel image with ReiserFS support to its proper place: (it is "/boot" directory usually.)
9. Change the /etc/lilo.conf file, so that you can boot with new kernel. Perform lilo command, please use lilo-21.6 or newer:
   Lilo-21.6-or-newer
10. Boot with the built kernel, mkreiserfs spare partition, and mount it:
    # mkreiserfs /dev/xxxx
    # mount /dev/xxxx /mount-point-dir
    or
    # mount -t reiserfs /dev/xxxx /mount-point-dir
11. Have fun.

Linux-2.4.0 to 2.4.2

ReiserFS code is inside Linux kernel from Linux 2.4.1-pre4.

1. Get Linux 2.4.2: http://www.kernel.org.
2. Get the latest ReiserFS-3.6.x patch.
3. Apply it:
   # zcat linux-2.4.2-reiserfs-20010327-full.patch.gz | patch -p0
4. Compile the kernel (as previously described, and turn on experimental features).
5. Get the ReiserFS utils: reiserfsprogs.
6. Untar in any dir, then compile and install rReiserFS utils:
   # tar -xzvf reiserfsprogs-3.x.0i-1.tar.gz
   # cd reiserfsprogs-3.x.0i-1
   # ./configure
   # make; make install
7. Boot with the built kernel, mkreiserfs spare partition.

Configuration There are compile-time options that affect ReiserFS functionality. You can set them up during the configuration stage of a Linux kernel build:

```
make config
make menuconfig
make xconfig
```

Turn on the ReiserFS option in the kernel configurator. This will build the ReiserFS external module.

Build ReiserFS. It will be either built into the kernel or as a stand-alone kernel module.

Option	Description
CONFIG REISERFS CHECK	If you set this to yes during the kernel configuration , then ReiserFS will perform every check possible of its internal consistency throughout its operation. It will also go substantially slower. Use of this option allows our team to check for consistency when debugging without fear of its effect on end-users. If you are on the verge of sending in a bug report, say yes and you might get a useful error message. Almost everyone should say no.

Option	Description
CONFIG REISERFS RAW	Setting this to yes will enable a set of ioctls that provide raw interface to ReiserFS tree, bypassing directories, and automatically removing aged files. This is an experimental feature designed for squid cache directories. See **Documentation/filesystems/reiserfs_raw.txt**. This was designed specifically to use ReiserFS as a back-end for the Squid. The general idea is that it is possible to bypass all file system overhead and address the ReiserFS internal tree directly. This is not in the stock kernels.
USE INODE GENERATION COUNTER	Use `s_inode_generation` field in the **3.6** super-block to keep track of inode generations. If not defined, use global `event` counter for this purpose (as do ext2 and most other file systems). The behavior of inode generations is important for NFS. This variable is unavailable through kernel configuration procedures, edit include/linux/reiserfs_fs.h manually.
REISERFS HANDLE BADBLOCKS	Enable ioctl for manipulating the bitmap. This can be used as crude form of bad block handling, but a real solution is underway. This variable is unavailable through kernel configuration procedures, edit include/linux/reiserfs_fs.h manually. Then, take a look at the available mount options.