Atomic Filesystems: Going Beyond Journaling

Namesys

This work was sponsored by the Defense Advanced Research Projects Agency of the United States Government (DARPA). DARPA does not endorse this work, it merely sponsors it.

Hans Reiser <reiser@namesys.com>

Vladimir Saveliev <vs@namesys.com>

Vladimir Saveliev <vs@namesys.com>

Nikita Danilov <nikita@namesys.com>

Alexander Zarochentcev <zam@namesys.com>

Vladimir Demidov <demidov@namesys.com>

Vitaly Fertman <vitaly@namesys.com>

Oleg Drokin <green@namesys.com>

Yury Umanetz <umka@namesys.com>

Edward Shushkin <edward@namesys.com>

Elena Gryaznova <grev@namesys.com>

Table of Contents

Abstract

I will discuss the performance aspects of Reiser4 design that allow us to both 1) implement filesystem operations as *atomic* operations, and 2) increase overall filesystem performance compared to ReiserFS V3. I will also suggest

that fully atomic filesystems keep your data more secure than journaling filesystems, and allow application writers greater ease in avoiding certain types of security holes.

Reducing The Damage of Crashing

When a computer crashes there is data in RAM which has not reached disk that is lost. You might at first be tempted to think that we want to then keep all of the data that *did* reach disk.

Suppose that you were performing a transfer of $10 from bank account A to bank account B, and this consisted of two operations 1) debit $10 from A, and 2) credit $10 to B.

Suppose that 1) but not 2) reached disk and the computer crashed. It would be better to disregard 1) than to let 1) but not 2) take effect, yes?

When there is a set of operations which we will ensure will all take effect, or none take effect, we call the set as a whole an *atom*. Reiser4 implements all of its filesystem system calls (requests to the kernel to do something are called *system calls*) as fully atomic operations, and allows one to define new atomic operations using its plugin infrastructure. Why don’t all filesystems do this? Performance.

Reiser4 possesses employs new algorithms that allow it to make these operations atomic at little additional cost where other filesystems have paid a heavy, usually prohibitive, price to do that. We hope to share with you how that is done.

A Brief History Of How Filesystems Have Handled Crashes

Filesystem Checkers

Originally filesystems had filesystem checkers that would run after every crash. The problem with this is that 1) the checkers can not handle every form of damage well, and 2) the checkers run for a long time. The amount of data stored on hard drives increased faster than the transfer rate (the rate at which hard drives transfer their data from the platter spinning inside them into the computer’s RAM when they are asked to do one large continuous read, or the rate in the other direction for writes), which means that the checkers took longer to run, and as the decades ticked by it became less and less reasonable for a mission critical server to wait for the checker.

Fixed Location Journaling

A solution to this was adopted of first writing each atomic operation to a location on disk called the *journal* or *log*, and then, only after each atom had fully reached the journal, writing it to the *committed area* of the filesystem (in describing ReiserFS V3 we used to call this the real area, but nobody much liked that term....).

The problem with this is that twice as much data needs to be written. On the one hand, if the workload is dominated by seeks, this is not as much of a burden as one might think. On the other hand, for writes of large files, it halves performance because such writes are usually transfer time dominated.

For this reason, meta-data journaling came to dominate general purpose usage. With meta-data journaling, the filesystem guarantees that all of its operations on its meta-data will be done atomically. If a file is being written to, the data in that file being written may be corrupted as a result of non-atomic data operations, but the filesystem’s internals will all be consistent. The performance advantage was substantial. V3 of reiserfs offers both meta-data and data journaling, and defaults to meta-data journaling because that is the right solution for most users. Oddly enough, meta-data journaling is much more work to implement because it requires being precise about what needs to be journaled. As is so often the case in programming, doing less work requires more code.

In sum, with fixed location data journaling, the overhead of making each operation atomic is too high for it to be appropriate for average applications that don’t especially need it --- because of the cost of writing twice. Applications that do need atomicity are written to use fsync and rename to accomplish atomicity, and these tools are simply terrible for that job. Terrible in performance, and terrible in the ugliness they add to the coding of applications. Stuffing a transaction into a single file just because you need the transaction to be atomic is hardly what one would call flexible semantics. Also, data journaling, with all its performance cost, still does not necessarily guarantee that every system call is fully atomic, much less that one can construct sets of operations that are fully atomic. It usually merely guarantees that the files will not contain random garbage, however many blocks of them happen to get written, and however much the application might view the result as inconsistent data. I hope you understand that we are trying to set a new expectation here for how secure a filesystem should keep your data, when we provide these atomicity guarantees.

Wandering Logs

One way to avoid having to write the data twice is to change one's definition of where the log area and the committed area are, instead of moving the data from the log to the committed area.

There is an annoying complication to this though, in that there are probably a number of pointers to the data from the rest of the filesystem, and we need for them to point to the new data. When the commit occurs, we need to write those pointers so that they point to the data we are committing. Fortunately, these pointers tend to be highly concentrated as a result of our tree design. But wait, if we are going to update those pointers, then we want to commit those pointers atomically also, which we could do if we write them to another location and update the pointers to them, and.... up the tree the changes ripple. When we get to the top of the tree, since disk drives write sectors atomically, the block number of the top can be written atomically into the superblock by the disk thereby committing everything the new top points to. This is indeed the way WAFL, the Write Anywhere File Layout filesystem invented by Dave Hitz at Network Appliance, works. It always ripples changes all the way to the top, and indeed that works rather well in practice, and most of their users are quite happy with its performance.

Writing Twice May Be Optimal Sometimes

Suppose that a file is currently well laid out, and you write to a single block in the middle of it, and you then expect to do many reads of the file. That is an extreme case illustrating that sometimes it is worth writing twice so that a block can keep its current location. If one does write twice, one does not need to ripple pointer updates to the top. If one writes a node twice, one also does not need to update its parent. Our code is a toolkit that can be used to implement different layout policies, and one of the available choices is whether to write over a block in its current place, or to relocate it to somewhere else. I don't think there is one right answer for all usage patterns.

If a block is adjacent to many other dirty blocks in the tree, then this decreases the significance of the cost to read performance of relocating it and its neighbors. If one knows that a repacker will run once a week (a repacker is expected for V4.1, and is (a bit oddly) absent from WAFL), this also decreases the cost of relocation. After a few years of experimentation, measurement, and user feedback, we will say more about our experiences in constructing user selectable policies.

Do we pay a performance penalty for making Reiser4 atomic? Yes, we do. Is it an acceptable penalty? This is hard to measure, because we picked up a lot more performance from other improvements in Reiser4 than we lost to atomicity, but I am fairly confident that the answer is yes. If changes are either large or batched together with enough other changes to become large, the performance penalty is low and drowned out by other performance improvements. Scattered small changes threaten us with read performance losses compared to overwriting in place and taking our chances with the data's consistency if there is a crash, but use of a repacker will mostly alleviate this scenario. I have to say that in my heart I don't have any serious doubts that for the general purpose user the increase in data security is worthwhile. The users though will have the final say.

Committing

A transaction preserves the previous contents of all modified blocks in their original location on disk until the transaction commits, and commit means the transaction has hereby reached a state where it will be completed even if there is a crash.

The dirty blocks of an atom (which were captured and subsequently modified) are divided into two sets, *relocate* and *overwrite*, each of which is preserved in a different manner.

The *relocatable* set is the set of blocks that have a dirty parent in the atom.

The *relocate* set is those members of the relocatable set that will be written to a new or first location rather than overwritten.

The *overwrite* set contains all dirty blocks in the atom that need to be written to their original locations, which is all those not in the relocate set. In practice this is those which do not have a parent we want to dirty, plus also those for which overwrite is the better layout policy despite the write twice cost. Note that the superblock is the parent of the root node and the free space bitmap blocks have no parent. By these definitions, the superblock and modified bitmap blocks are always part of the overwrite set.

The *wandered set* is the set of blocks that the overwrite set will be written to temporarily until the overwrite set commits.

An interesting definition is the *minimum overwrite* set, which uses the same definitions as above with the following modification. If at least two dirty blocks have a common parent that is clean then its parent is added to the minimum overwrite set. The parent's dirty children are removed from the overwrite set and placed in the relocate set. This policy is an example of what will be experimented with in later versions of Reiser4 using the layout toolkit.

For space reasons, we leave out the full details on exactly when we relocate vs. overwrite, and the reader should not regret this because years of experimenting is probably ahead before we can speak with the authority necessary for a published paper on the effects of the many details and variations possible.

When we commit we write a wander list which consists of a mapping of the wander set to the overwrite set. The wander list is a linked list of blocks containing pairs of block numbers. The last act of committing a transaction is to update the super block to point to the front of that list. Once that is done, if there is a crash, the crash recovery will go through that list and "play" it, which means to write the wandered set over the overwrite set. If there is not a crash, we will also play it.

There are many more details of how we handle the deallocation of wandered blocks, the handling of bitmap blocks, and so forth. You are encouraged to read the comments at the top of our source code files (e.g. wander.c) for such details....

Reiser4's Performance Design

Tree Design Concepts

Height Balancing versus Space Balancing

Height Balanced Trees are trees such that each possible search path from root node to leaf node has exactly the same length (*Length* = number of nodes traversed from root node to leaf node). For instance the height of the tree in Figure 2 is four while the height of the left hand tree in Figure 1.3 is three and of the single node in figure 1.0 is 1.

The term *balancing* is used for several very distinct purposes in the balanced tree literature. Two of the most common are: to describe balancing the *height*, and to describe balancing the *space usage* within the nodes of the tree. These quite different definitions are unfortunately a classic source of confusion for readers of the literature.

Most algorithms for accomplishing height balancing do so by only growing the tree at the top. Thus the tree never gets out of balance.

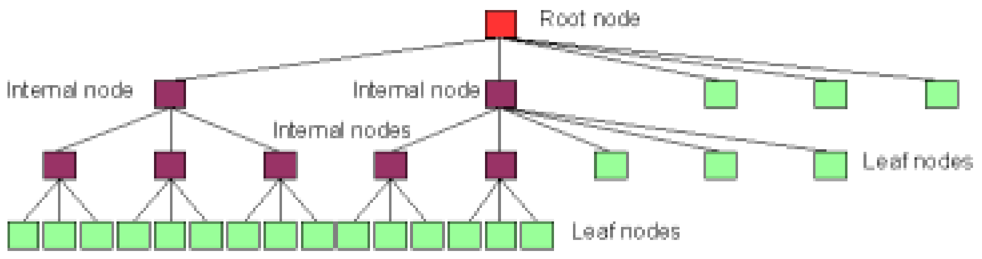


Fig. 1.11. This is an *unbalanced tree*. It could have originally been balanced and then lost some of its internal nodes due to deletion or it could have once been balanced but now be growing by insertion without yet undergoing rebalancing.

Three principle considerations in tree design

Three of the principle considerations in tree design are:

- the *fanout* rate (see below)
- the tightness of packing
- the amount of the shifting of items in the tree from one node to another that is performed (which creates delays due to waiting while things move around in RAM, and on disk).

Fanout

The fanout rate *n* refers to how many nodes may be pointed to by each level's nodes. (see Figure 2-1) If each node can point to *n* nodes of the level below it, then starting from the top, the root node points to n internal nodes at the next level, each of which points to n more internal nodes at its next level, and so on... *m* levels of internal nodes can point to *n* in power of *m* leaf nodes containing items in the last level. The more you want to be able to store in the tree, the larger you have to the fields in the key that first distinguish the objects (the *objectids*), and then select parts of the object (the *offsets*). This means your keys must be larger, which decreases fanout (unless you compress your keys, but that will wait for our next version....).

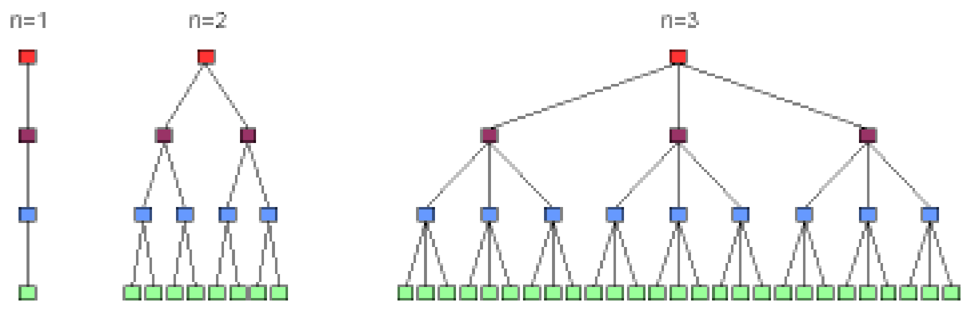
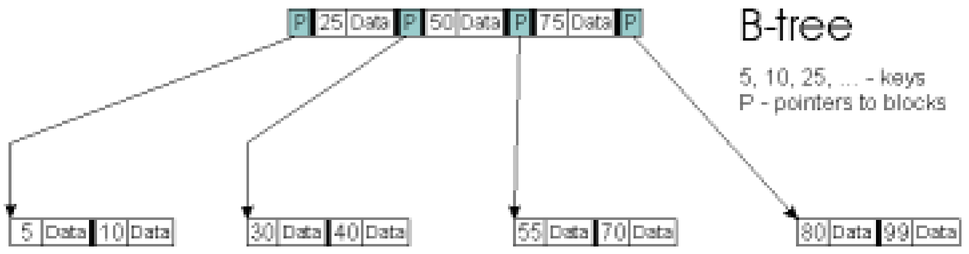


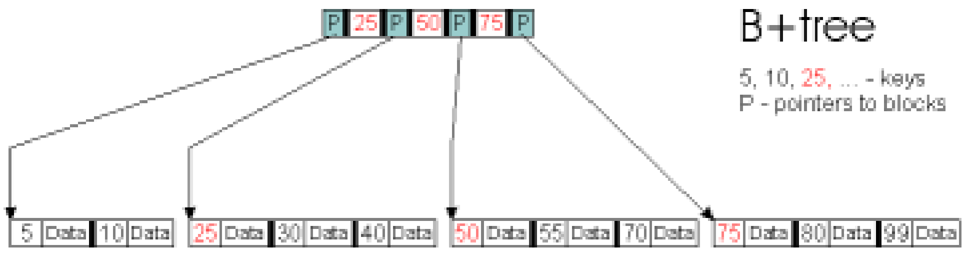
Figure 2-1. Three 4 level, height balanced trees with *fanouts* n = 1, 2, and 3. The first graph is a four level tree with fanout n = 1. It has just four nodes, starts with the (red) root node, traverses the (burgundy) internal and (blue) twig nodes, and ends with the (green) leaf node which contains the data. The second tree, with 4 levels and fanout n = 2, starts with a root node, traverses 2 internal nodes, each of which points to two twig nodes (for a total of four twig nodes), and each of these points to 2 leaf nodes for a total of 8 leaf nodes. Lastly, a 4 level, fanout n = 3 tree is shown which has 1 root node, 3 internal nodes, 9 twig nodes, and 27 leaf nodes.

What Are B+Trees, and Why Are They Better than B-Trees

It is possible to store not just pointers and keys in internal nodes, but also to store the objects those keys correspond to in the internal nodes. This is what the original B-tree algorithms did.

Then B+trees were invented in which only pointers and keys are stored in internal nodes, and all of the objects are stored at the leaf level.





Warning! I found from experience that most persons who don't first deeply understand why B+trees are better than B-Trees won't later understand explanations of the advantages of putting extents on the twig level rather than using BLOBs. The same principles that make B+Trees better than B-Trees, also make Reiser4 faster than using BLOBs like most databases do. So make sure this section fully digests before moving on to the next section, ok?;-)

B+Trees Have Higher Fanout Than B-Trees

Fanout is increased when we put only pointers and keys in internal nodes, and don't dilute them with object data. Increased fanout increases our ability to cache all of the internal nodes because there are fewer internal nodes.

Often persons respond to this by saying, "but B-trees cache objects, and caching objects is just as valuable". It is not, on average, is the answer. Of course, discussing averages makes the discussion much harder.

We need to discuss some cache design principles for a while before we can get to this.

Cache Design Principles

Reiser's Untie The Uncorrelated Principle of Cache Design

Tying the caching of things whose usage does not strongly correlate is bad.

Suppose:

- you have two sets of things, A and B.
- you need things from those two sets at semi-random, with there existing a tendency for some items to be needed much more frequently than others, but which items those are can shift slowly over time.
- you can keep things around after you use them in a cache of limited size.
- you tie the caching of every thing from A to the caching of another thing from B. (that means, whenever you fetch something from A into the cache, you fetch its partner from B into the cache)

Then this increases the amount of cache required to store everything recently accessed from A. If there is a strong correlation between the need for the two particular objects that are tied in each of the pairings, stronger than the gain from spending those cache resources on caching more members of B according to the LRU algorithm, then this might be worthwhile. If there is no such strong correlation, then it is bad.

But wait, you might say, you need things from B also, so it is good that some of them were cached. Yes, you need some random subset of B. The problem is that without a correlation existing, the things from B that you need are not especially likely to be those same things from B that were tied to the things from A that were needed.

This tendency to inefficiently tie things that are randomly needed exists outside the computer industry. For instance, suppose you like both popcorn and sushi, with your need for them on a particular day being random. Suppose that you like movies randomly. Suppose a theater requires you to eat only popcorn while watching the movie you randomly found optimal to watch, and not eat sushi from the restaurant on the corner while watching that movie. Is this a socially optimum system? Suppose quality is randomly distributed across all the hot dog vendors: if you can only eat the hot dog produced by the best movie displayer on a particular night that you want to watch a movie, and you aren't allowed to bring in hot dogs from outside the movie theater, is it a socially optimum system? Optimal for you?

Tying the uncorrelated is a very common error in designing caches, but it is still not enough to describe why B+Trees are better. With internal nodes, we store more than one pointer per node. That means that pointers are not separately cached. You could well argue that pointers and the objects they point to are more strongly correlated than the different pointers. We need another cache design principle.

Reiser's Maximize The Variance Principle of Cache Design

If two types of things that are cached and accessed, in units that are aggregates, have different average temperatures, then segregating the two types into separate units helps caching.

For balanced trees, these units of aggregates are nodes. This principle applies to the situation where it may be necessary to tie things into larger units for efficient access, and guides what things should be tied together.

Suppose you have R bytes of RAM for cache, and D bytes of disk. Suppose that 80% of accesses are to the most recently used things which are stored in H (hotset) bytes of nodes. Reducing the size of H to where it is smaller than R is very important to performance. If you evenly disperse your frequently accessed data, then a larger cache is required and caching is less effective.

1. If, all else being equal, we increase the variation in temperature among all aggregates (nodes), then we increase the effectiveness of using a fast small cache.
2. If two types of things have different average *temperatures* (ratios of likelihood of access to size in bytes), then separating them into separate aggregates (nodes) increases the variation in temperature in the system as a whole.
3. Conclusion: If all else is equal, if two types of things cached several to an aggregate (node) have different average temperatures then segregating them into separate nodes helps caching.

Pointers To Nodes Have A Higher Average Temperature Than The Nodes They Point To

Pointers to nodes tend to be frequently accessed relative to the number of bytes required to cache them. Consider that you have to use the pointers for all tree traversals that reach the nodes beneath them and they are smaller than the nodes they point to.

Putting only node pointers and delimiting keys into internal nodes concentrates the pointers. Since pointers tend to be more frequently accessed per byte of their size than items storing file bodies, a high average temperature difference exists between pointers and object data.

According to the caching principles described above, segregating these two types of things with different average temperatures, pointers and object data, increases the efficiency of caching.

Segregating By Temperature Directly

Now you might say, well, why not segregate by actual temperature instead of by type which only correlates with temperature? We do what we can easily and effectively code, with not just temperature segregation in consideration. There are tree designs which rearrange the tree so that objects which have a higher temperature are higher in the tree than pointers with a lower temperature. The difference in average temperature between object data and pointers to nodes is so high that I don’t find such designs a compelling optimization, and they add complexity. I could be wrong.

If one had no compelling semantic basis for aggregating objects near each other (this is true for some applications), and if one wanted to access objects by nodes rather than individually, it would be interesting to have a node repacker sort object data into nodes by temperature. You would need to have the repacker change the keys of the objects it sorts. Perhaps someone will have us implement that for some application someday for Reiser4.

BLOBs Unbalance the Tree, Reduce Segregation of Pointers and Data, and Thereby Reduce Performance

BLOBs, Binary Large OBjects, are a method of storing objects larger than a node by storing pointers to nodes containing the object. These pointers are commonly stored in what is called the leaf nodes (level 1, except that the BLOBs are then sort of a basement "level B" :-\ ) of a "B*" tree.

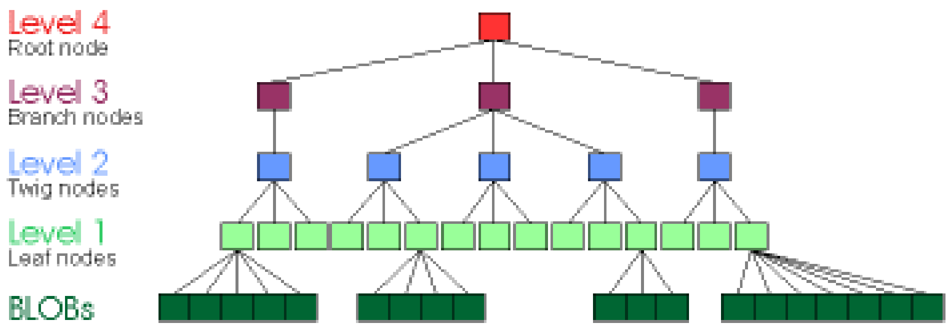


Fig. 1.12. A Binary Large OBject (*BLOB*) has been inserted with, in a leaf node, pointers to its blocks. This is what a ReiserFS V3 tree looks like.

BLOBs are a significant unintentional definitional drift, albeit one accepted by the entire database community. This placement of pointers into nodes containing data is a performance problem for ReiserFS V3 which uses BLOBs (Never accept that "let's just try it my way and see and we can change it if it doesn't work" argument. It took years and a disk format change to get BLOBs out of ReiserFS, and performance suffered the whole time (if tails were turned on.)). Because the pointers to BLOBs are diluted by data, it makes caching all pointers to all nodes in RAM infeasible for typical file sets.

Reiser4 returns to the classical definition of a height balanced tree in which the lengths of the paths to all leaf nodes are equal. It does not try to pretend that all of the nodes storing objects larger than a node are somehow not part of the tree even though the tree stores pointers to them. As a result, the amount of RAM required to store pointers to nodes is dramatically reduced. For typical configurations, RAM is large enough to hold all of the internal nodes.

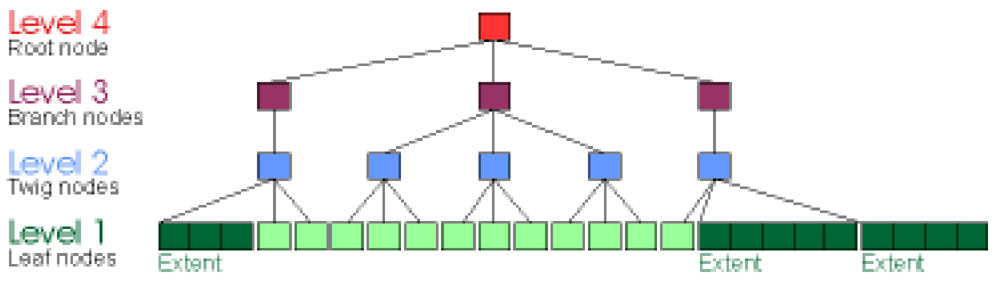


Fig. 1.13. A *Reiser4, 4 level, height balanced tree* with *fanout* = 3 and the *BLOB* (extent) stored in the level 1 leaf nodes. (For reasons of space, it is set below the other leaf nodes, but its extent-pointer is in a level 2 twig node like every other item's pointer.

Gray and Reuter say the criterion for searching external memory is to "minimize the number of different pages along the average (or longest) search path. ....by reducing the number of different pages for an arbitrary search path, the probability of having to read a block from disk is reduced." (1993, Transaction Processing: concepts and techniques, Morgan Kaufman Publishers, San Francisco, CA, p.834 ...)

My problem with this explanation of why the height balanced approach is effective is that it does not convey that you can get away with having a moderately unbalanced tree provided that you do not significantly increase the total number of internal nodes. In practice, most trees that are unbalanced do have significantly more internal nodes. In practice, most moderately unbalanced trees have a moderate increase in the cost of in-memory tree traversals, and an immoderate increase in the amount of IO due to the increased number of internal nodes. But if one were to put all the BLOBs together in the same location in the tree, since the amount of internal nodes would not significantly increase, the performance penalty for having them on a lower level of the tree than all other leaf nodes would not be a significant additional IO cost. There would be a moderate increase in that part of the tree traversal time cost which is dependent on RAM speed, but this would not be so critical. Segregating BLOBs could perhaps substantially recover the performance lost by architects not noticing the drift in the definition of height balancing for trees. It might be undesirable to segregate objects by their size rather than just their semantics though. Perhaps someday someone will try it and see what results.

Block Allocation

Parent First Pre-Order Is What We Try For

Version 3 Too Readily Adds Disk Seeks Between Nodes In Response To Changes In The Tree

In ReiserFS we attempt to assign keys for what we store in the tree such that typical accesses are accesses to objects in the order of their keys. Since one generally has to access a parent before accessing its children, this means accesses will be in parent first pre-order, as we draw it below (the below is an unbalanced tree).

In the block allocation code we then attempt to assign block numbers such that accesses to nodes in parent first preorder involve accessing nodes in the order of their block numbers.

This added layer of abstraction in which we control layout by key assignment policy, and assign blocknumbers by tree order, is very flexible in practice, and we find it improves substantially our ability to perform more experiments with layout alternatives, and to adapt for particular application needs.

In Version 3, whenever we insert a node into the tree, we assign it a block number at the time we insert it. We assign it a block number as close as we can to its left neighbor in the tree.

One problem is that the tree can be very different in its structure at the time we insert a given block compared to when we finish some batch operation such as copying a directory and all of its files, or even just copying the current file if it is a large file. The less fanout there is, the more that shifts of items tend to lead to structural changes in the tree that change what the optimal block allocation is. The less accurate we are in allocating blocks, the more seeks.

In Version 4, we adopt new techniques to reduce this problem: we increase fanout by getting rid of BLOBs as described at www.namesys.com/v4/v4.html, and we allocate block numbers at flush time (when dirty nodes are written to disk) instead of at node insertion time.

We determined, as part of the art of design compromise, that placing twigs before their children sadly matters enough to performance to merit doing it, but placing branches before their children does not because branches are more likely to be cached and are relatively rare. Branches are therefor write optimized rather than read-optimized in their layout --- that means we write the branches at the end of flushing an atom, and we try to write them all together in a location likely to not require a seek away from what was written just before them. In respect to allocating blocks for twigs, I say sadly, because the added complexity of allocating blocks for twigs just before their children is immense when you consider that we are also squeezing them, and squeezing and encrypting and compressing their children, as we allocate blocks, and the effect of the squeezing and allocating and encrypting can affect the number of extents required to store a file which affects how much fits into a twig which affects who is a child of what twig.... We did it, but not easily, I regret to say. If a future designer finds that he can neglect read-optimizing twigs, perhaps as a result of a design improvement or a different usage pattern, it will immensely simplify the code, and this point is worth noting.

Dancing Trees

Traditional balanced tree algorithms apply a fixed criterion for whether a node is tightly enough packed, or should be combined with its neighbors. For example, some will check on each modification whether the node modified plus its left and its right neighbor can be squeezed into one fewer nodes. ReiserFS V3 does this on the leaf level of the tree. The stricter this algorithm is, and the more neighbors it considers, the tighter the packing is, and the more bytes must be shifted on average per modification in order to make space for that modification. That means more overhead in both CPU and IO.

For this reason, some databases actually employ free on empty as their static space usage balancing criterion.

"Garbage collection" algorithms are a different way of managing space usage in memory. When space runs short, they run through memory repacking it and reclaiming unused space, rather than repacking with each usage or freeing of memory as they might otherwise have done. This is like not driving to the recycler every time you finish a bottle of milk. (ReiserFS is environment friendly --- we recycle memory rather than discarding it....)

With Reiser4, when we are short on space in RAM, the memory manager asks us to write a dirty node to disk. We collect together the maximal set of nodes that include the node we were asked to flush, are contiguous together with each other, and are dirty. We call this set a *slum*. We squeeze the slum into the minimum number of nodes it can squeeze into. If the slum is, say, 64 nodes large after squeezing, that will be much tighter than a fixed balancing criterion tree would pack it because it will be at least 63/64ths full, and most tree balancing algorithms examine only the immediate neighbors when deciding how tight they can pack. (If the slum is one node in size, we have several options we are experimenting with, including the one we suspect will be best for "typical" use, which is leaving it for the repacker to optimize well.)

The effect of this approach is that hot spots become relatively loosely packed so that insertions are cheap, and slums that are no longer hot spots get very tightly packed before being sent to disk.

An area for possible future work is to allow for slum squeezing to eliminate the need for performing IO if enough memory is freed by it. This might require changes to the kernel memory manager though.

This resembles traditional garbage collection algorithms. So far as we know though, applying this approach to trees is unique to reiser4, and we call a tree whose packing is managed in this way a "dancing tree" rather than a "space usage balanced tree". (Remember, there is a difference between height balancing and space usage balancing, and Reiser4 trees are still (very) height balanced trees....)

Repacker

Another way of escaping from the balancing time vs. space efficiency tradeoff is to use a repacker. 80% of files on the disk remain unchanged for long periods of time. It is efficient to pack them perfectly, by using a repacker that runs much less often than every write to disk. This repacker goes through the entire tree ordering, from left to right and then from right to left, alternating each time it runs. When it goes from left to right in the tree ordering, it shoves everything as far to the left as it will go, and when it goes from right to left it shoves everything as far to the right as it will go. (Left means small in key or in block number:-)). In the absence of FS activity the effect of this over time is to sort by tree order (defragment), and to pack with perfect efficiency.

Reiser4.1 will modify the repacker to insert controlled "air holes", as it is well known that insertion efficiency is harmed by overly tight packing.

I hypothesize that it is more efficient to periodically run a repacker that systematically repacks using large IOs than to perform lots of 1 block reads of neighboring nodes of the modification points so as to preserve a balancing invariant in the face of poorly localized modifications to the tree.

What Is Best Done Just Before Flushing To Disk vs. At Tree Insertion Time vs. In Nightly Repacker

Painting it in broad strokes, Reiser4 is designed to massage and performance optimize changes to data in three stages:

- at the time the system call that invokes Reiser4 executes (this may put the data into the tree, the page cache, the dentry cache, etc.)
- at the time the data is flushed from a cache (usually to disk)
- at the time the repacker runs

Different kinds of information massaging belong in each of these stages:

- Object plugins modify the data in ways appropriate to do as the data is being changed.
- Flush plugins modify the data in ways appropriate to do as the data is flushed to disk.
- The repacker modifies the data in ways appropriate to do only at long time intervals.

I see the flushing stage as the time most optimal for:

- allocating blocks to nodes, including deciding whether to overwrite or relocate
- squeezing nodes together for tight packing before they go to disk
- encrypting data

A Hint of Things Left For Other Papers

At the time of writing we implement all VFS operations atomically, and the infrastructure for atomic operations is fully complete. The interface for specifying multiple FS operations that are atomic is still being tested and debugged, we are going to let our mailing list comment before we fix it in stone, and space in this paper is limited, so we have left it for a future, hopefully interesting, paper.

Expediency often tempts security researchers into adding architecturally inelegant extensions to filesystems. Reiser4's objective was to systematically reviewed all of the infrastructure lackings that tempt security researchers into implementing inelegance, and systematically approached remedying them. It creates a complete plugin infrastructure that allows you to construct custom made file objects. It implements inheritance, constraints, encryption at flush time, and compression at flush time. We implement all of the features needed to effectively implement security attributes as just flavors of files. Space does not permit discussing these features, or the philosophy behind why security attributes should be just particular flavors of files, in this paper, but please watch our website at www.namesys.com.

Conclusion

Atomic filesystems keep your data more secure. Historically filesystems have not been made atomic in large part because of the performance cost. One component of this performance cost is the need to write twice, and Reiser4 avoids this. Reiser4 introduces a set of performance improvements that dwarf what additional cost of atomicity remains after the write twice penalty is eliminated, and the result is that you are going to have both greater data security and faster performance when you use it. Application writers are going to be able to avoid a variety of security holes by tapping into the atomic operation functionality.

References

A summary of some influences upon Reiser4 performance design:

- Its avoidance of seeks for writes is influenced by LFS and WAFL.
- Plans for a repacker in Reiser4.1 resemble the use of a cleaner in LFS.
- Its flush plugins are a generalization of delayed block allocation used in XFS.
- Performance problems, benefits (larger blocks), and space savings, due to tails in ReiserFS V3 (not Reiser4) resemble performance problems (these are not well documented in the literature unfortunately), benefits (they do document in the FFS paper cited below the benefit of being able to increase block size as a result of not fearing as much its cost in space usage), and space savings, due to fragments in FFS.
- The use of balanced trees for aggregating multiple small objects into one node resembles how they are used in databases.

Citations:

- *[Gray93]*

  Jim Gray and Andreas Reuter. "Transaction Processing: Concepts and Techniques". Morgan Kaufmann Publishers, Inc., 1993. Old but good textbook on transactions. Available at http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-190-2 [http://www.mkp.com/books_catalog/catalog.asp?ISBN
  55860-190-2]

- *[Hitz94]*

  D. Hitz, J. Lau and M. Malcolm. "File system design for an NFS file server appliance". Proceedings of the 1994 USENIX Winter Technical Conference, pp. 235-246, San Francisco, CA, January 1994 Available at http://citeseer.nj.nec.com/hitz95file.html

- *[TR3001]*

  D. Hitz. "A Storage Networking Appliance". Tech. Rep TR3001, Network Appliance, Inc., 1995 Available at http://www.netapp.com/tech_library/3001.html

- *[TR3002]*

  D. Hitz, J. Lau and M. Malcolm. "File system design for an NFS file server appliance". Tech. Rep. TR3002, Network Appliance, Inc., 1995 Available at http://www.netapp.com/tech_library/3002.html

- *[Kusick84]*

  M. McKusick, W. Joy, S. Leffler, R. Fabry. "A Fast File System for UNIX". ACM Transactions on Computer Systems, Vol. 2, No. 3, pp. 181-197, August 1984 Available at http://citeseer.nj.nec.com/mckusick84fast.html

- *[Ousterh89]*

  J. Ousterhout and F. Douglis. "Beating the I/O Bottleneck: A Case for Log-Structured File Systems". ACM Operating System Reviews, Vol. 23, No. 1, pp.11-28, January 1989 Available at http://citeseer.nj.nec.com/ousterhout88beating.html

- *[Seltzer95]*

  M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains and V. Padmanabhan. "File System Logging versus Clustering: A Performance Comparison". Proceedings of the 1995 USENIX Technical Conference, pp. 249-264, New Orleans, LA, January 1995 Available at http://citeseer.nj.nec.com/seltzer95file.html

- *[Seltzer95Supp]*

  M. Seltzer. "LFS and FFS Supplementary Information". 1995 http://www.eecs.harvard.edu/~margo/usenix.195/

- *[Ousterh93Crit]*

  J. Ousterhout. "A Critique of Seltzer's 1993 USENIX Paper" http://www.eecs.harvard.edu/~margo/usenix.195/ouster_critique1.h

- *[Ousterh95Crit]*

  J. Ousterhout. "A Critique of Seltzer's LFS Measurements" http://www.eecs.harvard.edu/~margo/usenix.195/ouster_critique2.htm

- *[Rosenblum92]*

  M. Rosenblum and J. Ousterhout. "The design and implementation of a log-structured file system". ACM Transactions on Computer Systems, Vol. 10, No. 1, pp. 26-52, February 1992 Available at http://citeseer.nj.nec.com/rosenblum91design.html

- *[SwD96]*

  A. Sweeny, D. Doucette, W. Hu, C. Anderson, M. Nishimoto and G. Peck. "Scalability in the XFS File System". Proceedings of the 1996 USENIX Technical Conference, pp. 1-14, San Diego, CA, January 1996 Available at http://citeseer.nj.nec.com/sweeney96scalability.html