Examining Linux 2.6 Page-Cache Performance

Sonny Rao, Dominique Heger, Steven Pratt
IBM Corporation
{sonnyrao, dheger, slpratt}@us.ibm.com

Abstract

Given the current trends towards ubiquitous 64-bit server/desktop computing with large amounts of cheap system memory, the performance and structure of the Linux® page-cache will undoubtedly become more important in the future. An empirical and analytical examination of performance will be valuable in guiding future development.

The current 2.6 radix-tree based design represents a huge leap forward from the old global hash-table design, but there may be some issues with the current radix-tree structure itself.

The main goal is to understand performance of the current implementation, examine performance with respect to other potential data-structures, and look at ways to improve concurrency.

1 The Radix-Tree based Page Cache in Linux 2.6

The Linux 2.6 page cache is basically a collection of pages that normally belong to files. The pages are kept in memory for performance reasons. As on other UNIX® operating systems, the page cache may take up the majority of the available memory. Whenever a thread reads from or writes to a file, takes a page fault, or is paged out, the page cache becomes involved. Hence, the performance of the page cache has a rather dramatic impact on the performance of the system. As a particular page is referenced, the page cache has to be able to locate the page, or has to determine that the page is not in the cache, in as efficient and effective way as possible with a focus on minimal memory overhead.

1.1 Evolution of the Page Cache

In older versions the Linux kernel utilized a global hash-table based approach to maintain the pages in the cache. The hash based approach had some performance issues:

1. A hash key is normally not unique; hence the system has to resolve collisions. A hash chain had to be built to hold entries (each entry used up 8 bytes per referenced page).
2. A single global lock governed the page cache; causing scalability issues on SMP based systems.

The radix-tree based page cache solution addresses the issues discussed above.

Technically, the Linux 2.6 system consists of many smaller page cache subsystems, or more

specifically, one for each open file in the system.

Segregating page caches has a few advantages:

First, each page cache can have its own lock, avoiding the global page cache lock that was necessary in older versions.

Second, search operations work on a smaller address space, and complete more quickly.

Third, as there is one page cache per open file, the only index required to look up a specific page is the offset within the file.

In the radix-tree, the 32-bit or 64-bit file offset is divided into subsets whose size is based on the value of `MAP_SHIFT` as defined in lib/radix.c. The current implementation uses a `MAP_SHIFT` of six for 6-bit indices. The highest-order sub-field (or set) is used to branch into a 64-entry table in the root of the radix-tree. An entry in that sub-table serves as a pointer to the next node in the tree. The next lower sub-field (from the index) is used to index that particular node, yielding a third abstraction. Eventually, the algorithm will reach the bottom of the tree and obtain the actual page pointer or finds an empty entry, signifiying that the page is not present. Table 2 shows maximum file offset and number of pages versus tree height for the shift value of six.

There is some precedent for using a value other than six for the `MAP_SHIFT`. Originally, seven was used for the `MAP_SHIFT` when the structure was first introduced [7]. Larger values mean smaller trees in terms of height and the possiblity of shorter search times. This possibility comes at the expense of bigger nodes in the slab cache, which means that there is more potential for wasted entries.

Shift	through-put	delta	profile	delta
6	4.61	N/A	13.21	N/A
8	4.745	(+3)%	12.09	(–8.7)%
10	4.705	(+2)%	12.40	(–6.14)%
12	4.695	(+1.8)%	12.31	(–6.72)%
4	4.683	(+1.6)%	17.22	(+30.4)%

Table 1. Sequential read throughput and percent of profile ticks for `radix_tree_lookup` results for different values of `MAP_SHIFT`. The units for throughput are GB/sec, and the profile column represents time spent in `radix_tree_lookup`. These values represent the average of four runs.

height	maximum pages	maximum file offset
0	0	0
1	64	256 KB
2	4096	16 MB
3	262144	1 GB
4	16777216	64 GB
5	1073741823	4 TB
6	4294967296	16 TB

Table 2. Max number of pages by radix-tree height with a 32-bit key and `MAP_SHIFT` of 6, file offset assumes 4k pages

One optimization criterion was to ensure that the radix-tree would only be as deep as necessary. In the case where the system operates on small files (smaller than 65 pages), only one level of abstraction (one node) will be used. In other words, only the least significant sub-field of the offset is being utilized. This property of the current implementation allows the normally detrimental effects of a large key on a radix-tree to be minimized. The only potential downside is in the case of a sparse file where nodes located at relatively large offsets will force a higher tree depth than might otherwise be necessary.

1.2 Newer Features in 2.6

One of the newer features incorporated in Linux revolves around 'tagging' dirty pages in the radix-tree. In other words, a dirty page is only flagged in the radix-tree, and not moved to a separate list as in the pre 2.6.6 design. Along the same lines, pages that are being written back to disk are flagged as well. A new set of radix-tree functions was implemented to locate these pages as necessary. Searching an entire tree structure for these pages is not as efficient as just traversing trough a dedicated list, but based on the feedback from the Linux community, the performance delta is not considered a big issue. There is some concern in the Linux community that with very large files the 2.6 lock-per-file based approach will be as bad as the global lock based 2.4 implementation. The tagging of these pages in the new design required a lot of changes to the page cache and the VM subsystems, respectively. One implication of the changes is that the dirty pages are now always written in file offset order out to disk. As the Linux community reports, this may cause some performance issues involving parallel write() operations on large SMP systems. The tagging of all these pages in the radix-tree contributes to the complexity of switching from a radix-tree based approach to another data structure (if needed). Based on the current implementation, improving the radix-tree seems more feasible than a complete re-design and should therefore be explored first. The `MAP_SHIFT` parameters in the radix code reveal some potential for performance work.

There is a scalability issue when dealing with only a small amount of very large files and a workload that consists of many concurrent read operations on the files. The single lock governing the radix-tree will basically eliminate any potential scalability on SMP systems while exposed to such a workload. Scalability of course is achieved when the workload consists of n worker threads reading from n separate files, hence the locking is distributed over the set of files being accessed. Table 3 shows the severity of the locking problems of the current spinlock design vs the rwlock design and shows that even the rwlock implementation spends a good deal of time overall CPU time in locking functions.

Table 3 shows throughput on an IBM p650 8-CPU POWER4+ server with 16GB of RAM and two 7GB files fully cached with differing numbers of threads attempting to sequentially read the files. Throughput is in GB/sec and the profile columns show the percentage of time from the profile spent in locking functions.

Threads	Spinlock	Profile	Rwlock	Profile
1	1.11	0.10%	1.04	0.80%
4	2.26	12.4%	2.47	4.33%
8	2.01	54.1%	2.82	9.75%
12	2.20	51.6%	2.98	9.86%
14	2.31	49.3%	3.03	9.74%
16	2.34	48.9%	3.13	9.52%

Table 3. Read throughput and time spent in lock functions for spinlock and rwlock kernels.

There has been some ongoing debate over whether a rwlock solution would be more acceptable, however as of this writing it has been held out of mainline due to specific concerns over the performance of the rwlock solution on Pentium-4 machines [9, 10]. Although the cost of locking is substantial on all architectures, this architecture seems to exhibit particularly high latency on the unlock operation. This also seems to indicate that the radix-nodes tend to be cached and that search times are small [8].

2 Alternative Data-Structures

Given the unique nature of the radix implementation in the Linux kernel, comparative analysis

of the radix-tree with alternative data-structures should provide insights into its strengths and weaknesses. In general, for the application of page-cache lookup, speed should be paramount since in the case of a cache hit, the entire read or write operation should occur at memory speed. Inserts, on the other hand, will typically be followed by disk I/O, and that I/O should become the limiting factor for the operation rather than the cache update. Deletes are initiated from a truncate operation or by the page-scanner when the system is under memory pressure. This case of memory pressure is performance critical since the VM wants to release the pages selected as soon as possible, and updates to the caching structures represent pure overhead. Operations such as "tagging" pages as dirty are also interesting because they involve both a lookup and a modification to the state of the data structure. However this operation is specific to the Linux 2.6 radix-tree implementation and is not available on all data-structures. In some cases it may be possible to graft these additional pieces of state information onto other standard data-structures, but it is not practical in all cases.

Given these qualities, it seemed appropriate to test the Linux kernel implementation of radix-trees against a number of other data-structures each with slightly different design trade-offs.

2.1 Extendible-Hashing

One idea suggested was that of extendible-hashing, which is a technique developed for optimizing lookup operations in database systems [6]. Among other interesting properties, extendible hashing guarantees that data can be accessed in just two "page-faults" in database terminology, which translates to two pointer dereferences for our purposes. As the name suggests, it is capable of extending itself as the amount of data stored increases, and it can do this without costly re-hashing of the entire data-set. Conversely, the extendible hash-table can be implemented to contract itself as elements are removed. Naturally, these characteristics are not free and represent a trade-off for the fixed number of memory dereferences in the lookup path.

The extendible hash-table typically is implemented using two structures: buckets, which contain the pointers to the data, and a directory, which contains the pointers to the buckets. The directory is just a large array with a power-of-two size. The logarithm of the current size is called the directory depth.

Elements are inserted by computing a hash key and taking the n most-significant bits of that key, where n is equal to the directory depth. Using this value to index into the directory yields a pointer to the bucket where the new element will reside. Different strategies exist for placing an element into a bucket. Depending on the size of the bucket, the object's hash value can be used again to place the item, or if the bucket is fairly small, a simple linear insert can be effective.

Each pointer in the directory is not necessarily unique, and there can be mulitple pointers to a certain bucket. For this reason, the buckets keep a local-depth value, which can be used to compute the number of pointers to it in the directory. When a bucket becomes full, it must be split into two separate buckets in an operation called a bucket-split. After the bucket-split, each new bucket will get half of the old pointers in the directory, and the local depth of the buckets will increase by one. If the bucket has a local depth equal to the directory depth, then the directory must be first doubled in size before the bucket can be split. In this case, there is only one pointer in the directory to this particular bucket before the directory doubling operation, and afterwards there are two pointers and the bucket-split can proceed. When a

bucket-split occurs, the elements in the original bucket are redistributed into the new buckets in such a way that their hash-keys will lead to the correct bucket from the directory. In this way, the extendible hash-table avoids having to ever globally re-hash and instead limits redistribution to bucket-splits while retaining the original hash function.

One additional characteristic of the extendible-hashing is its ability to handle random sequences of keys equally as well as sequential sequences. Though many typical applications will primarily use sequential I/O patterns, some applications might find this characteristic beneficial.

2.2 Threaded Red-Black Tree

Threaded red-black trees are a twist on the notion of a traditional red-black tree, which try to optimize for sequential access sequences by using normally NULL leaf pointers as "threads" which keep track of nodes with neighboring keys [12]. So, if one already has a reference to a particular leaf node, access to the previous node (in terms of key order) only requires accessing that node's left "thread."

The regular red-black tree properties still apply [1,2], but since almost all child pointers are used in some way, additional state information must be kept in the nodes to differentiate children from threads. Luckily, red-black trees, such as the one in the Linux kernel, already use an extra word per node to keep track of color. This extra word can be overloaded to keep track of thread information as well with no additional space cost.

Since one cannot simply test for NULL during lookups, one must also alter any open-coded lookup sequences to be thread-aware, which is to say such code must examine the state information in the node. Ideally, this should not be a significant cost because the flags should typically have reasonable spatial locality with the other pointers in the node and would be kept in the same cache-line.

As with regular red-black trees, performance of inserts and deletes is traded off to keep the tree balanced and keep average lookup times down. In the case of the threaded version this is even more true as thread information must be kept consistent through rebalancing operations.

The implementation tested was similar to the Linux kernel's present red-black tree implementation which assumes the node contents are embedded into another object and passes off responsibility for memory allocation and implementing lookups onto the tree's user.

2.3 Treap

A treap is the basic data structure (BST) underlying randomized search trees [3]. The name itself refers to the synthesis of a tree and a heap structure. More specifically, a treap represents a set of items where each item has associated with it a key and a priority. In general a priority is randomly assigned to a given key by the implementation. A treap represents a special case of a binary search tree, in which the node set is arranged in order (with respect to the keys) as well as in heap fashion with regards to the priority. The procedure for lookup in a treap is the same as for a normal binary search-tree and the node priorities are simply ignored. In a treap, the access time is proportional to the depth of an element in the tree. An insert of a new item basically consists of a two step process. The first step consists of utilizing the item's key to attach to the treap at the appropriate leaf position, and second to use the priority of the new element to rotate the new entry up in the structure until the item locates the parent node that has a larger priority. Interestingly, it can be shown in the general case that

an insert will only cause two rotations, which means updates are much less costly then in the case of a strictly balanced tree such as an AVL tree or red-black tree.

The implementation tested used a simple polynomial hash function on the key to generate the priority. This approach was used instead of the kernel's random number generator to keep the implementation as self-contained as possible.

Again, the implementation tested follows the Linux kernel's convention of assuming the user must allocate the nodes and open-code the lookup sequences.

2.4 Linux Radix-Tree

The Linux implementation of the radix-tree is highly optimized and customized for use in the kernel and differs signifcantly from what is commonly referred to as a radix-tree [1,4,5] It avoids paying the memory cost of explictly keeping keys, child-pointers, and separate data-pointers on each object but instead uses implicit ordering along with node height to determine the meaning of these pointers. For example, if the tree has a global height of three, then the pointers on the first two levels only point to child nodes and the lowest level uses its pointers for data objects. Data pointers only exist at the lowest level.

By aggressively conserving memory and reducing the tree's overall size, the radix-tree has an extremely small cache footprint which is vital to its success at larger tree sizes.

The main disadvantage of using implicit ordering in the implementation is that a highly sparse file will force the use of more tree-levels across the entire tree for all offsets. The current implementation uses a `MAP_SHIFT` of six which means sixty-four pointers per node, and in the worst case all but one of those pointers is wasted from the root all the way down to the leaf. The height is directly related to the offset of the last object inserted into the tree.

The kernel implementation also supports tagging which means each node not only consists of an array of pointers but a set of bit-fields for each pointer which can be used somewhat arbitrarily by the subsystem utilizing the tree. In the case of the page-cache, these tags are used to keep track of whether a page is dirty or undergoing writeback.

The meaning of these tags is clear at the leaf nodes, but at higher levels, tags are used to refer to the state of any objects in or below the child node at the corresponding offset.

For example, given a three level radix-tree, and the page at offset one is dirty, then the dirty-tag for bit one on the leaf node is set and the tags for bit zero are set in the two nodes above. This way, gang-lookups searching for tagged nodes can be optimized to skip over subtrees without any tagged descendants.

2.5 Analysis

In all three operations tested, there was no significant difference between the data structures until roughly 128K elements where the differences begin and are highlighted by the remaining data points.

The extendible-hashing results were initially very surprising as it seems to perform much worse than the tree structures at high object counts. After analyzing performance counter information, it was determined that the extremely poor spatial and temporal locality of the the hash directory and buckets were causing TLB and similar translation cache misses and thus large amounts of time were spent doing

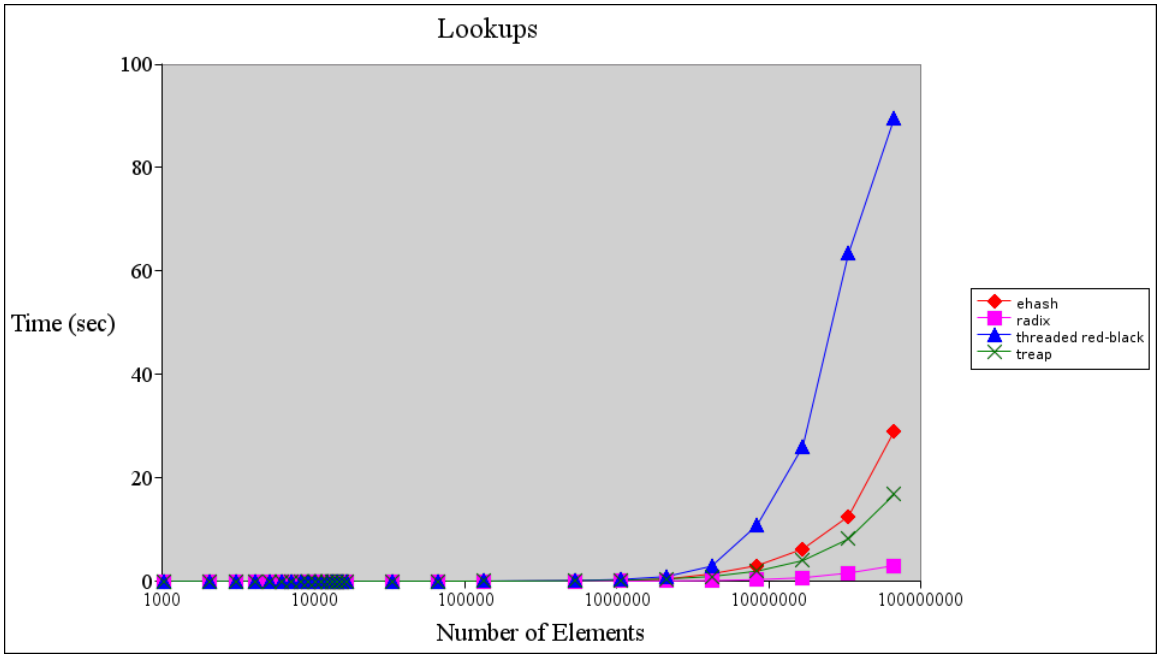

Figure 1: Sequential Lookup Performance

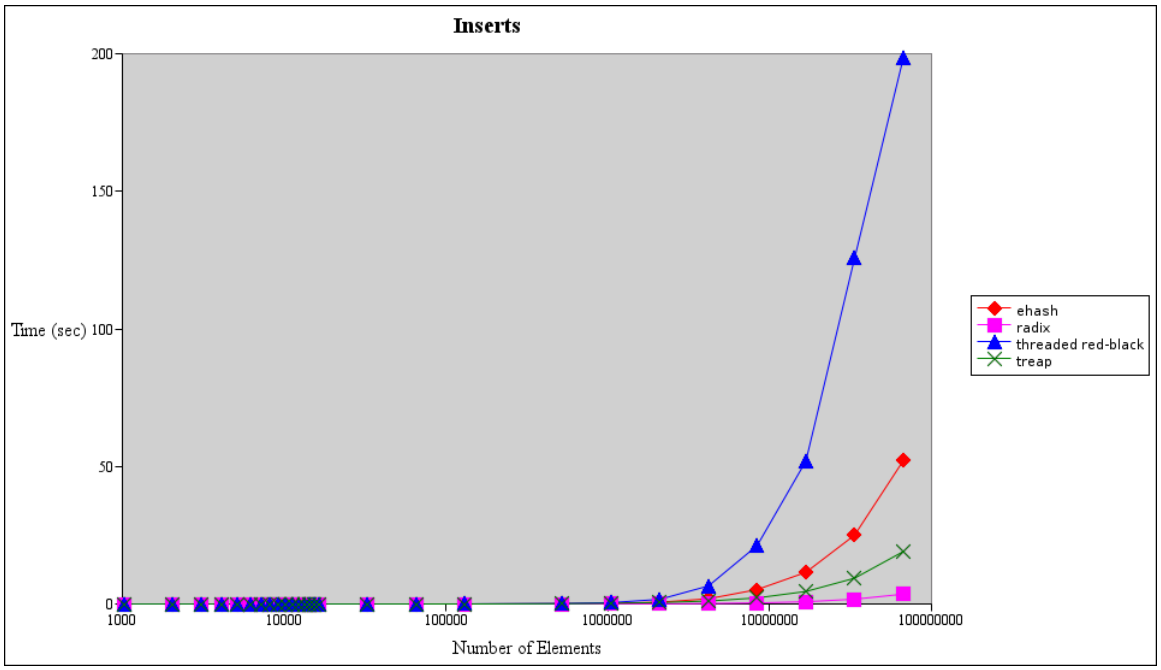

Figure 2: Sequential Insert Performance

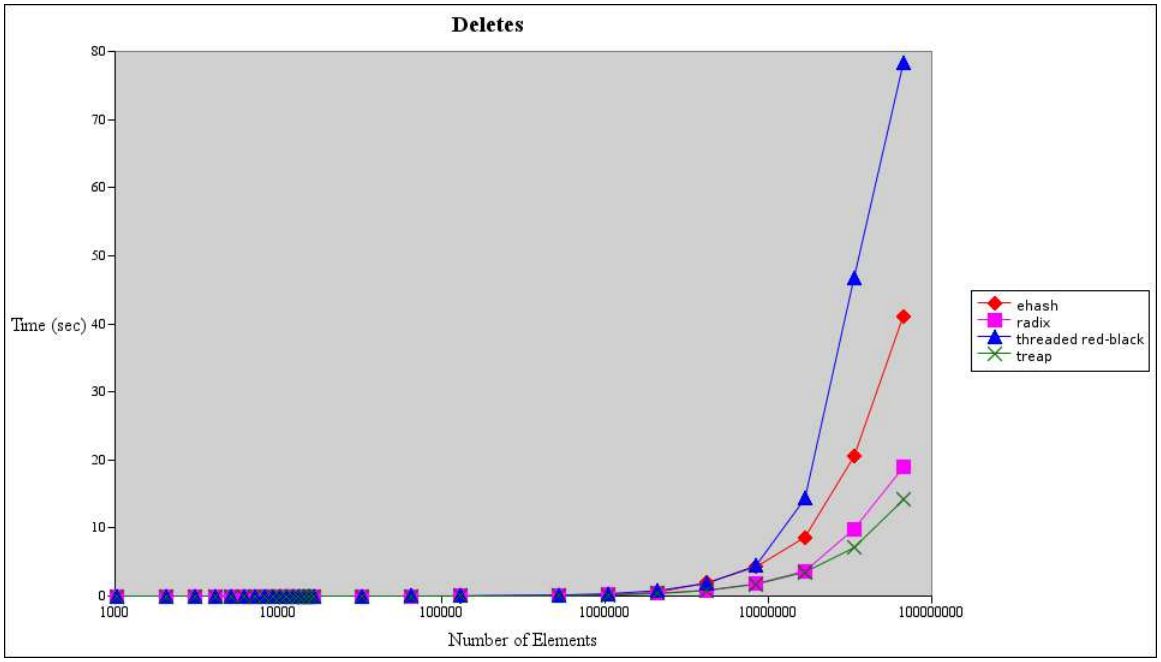

Figure 3: Sequential Delete Performance

page-table walk operations. Also, the poor spatial locality caused a great deal of data-cache misses which compounded the problems. On the other hand, for the tree structures, the sequential nature of the test yielded significant benefits to their cache interactions.

The two binary-tree structures offer mixed performance with generally worse performance on lookups and inserts with only the treap narrowly beating the radix-tree in deletes. The threaded red-black tree also seems to do worse than expected in lookups which will require some further analysis.

The radix-tree scales extremely well into the very large numbers of pages because the tree itself fits into processor caches much better than the alternative designs. In the case of the delete operations the radix-tree still does well, but is beaten in some cases by the treap. Most likely, this is because of the extensive updates which must occur to the tagging information up the tree which typically would not have good spatial or temporal locality with respect to the initial lookup.

This result has also been observed under a 'real' data-base workload where the `radix_tree_delete` call shows up higher in kernel profiles than the `radix_tree_lookup` operations, which was initially rather confusing, as it was expected that most of the time in the radix-tree code would be spent doing lookup operations. Table 4 shows this effect, where `radix_tree_delete` shows up as the third highest kernel function and `radix_tree_lookup` is number ten. Overall, this particular database query is heavily I/O bound, as `dedicated_idle` represents time spent waiting on I/O to complete, and the rest of the functions indicate memory pressure (`shrink_list`, `shrink_cache`, `refill_inactive_zone`, and `radix_tree_delete`) and other filesystem activity (`find_get_block`).

DB Workload: Top 10 Kernel Functions

dedicated_idle
__copy_tofrom_user
radix_tree_delete
_spin_lock_irq
__find_get_block
shrink_list
refill_inactive_zone
__might_sleep
shrink_cache
radix_tree_lookup

Table 4: Kernel functions reported by OProfile from a standard commercial database benchmark which simulates a business decision support workload. The tests were run on IBM OpenPower 720 4-CPU machine running on Ext3 with 92% of time spent in the kernel for this query. Other querys in the workload showed similar results where in all cases `radix_tree_delete` was ordered higher than `radix_tree_lookup`.

2.6 Continuing Work

In the interests of time, all of these results were collected in userspace. As time permits, the tests can be re-done using kernel-space implementations to keep user-space biases to a minium and to avoid any bias due to the memory allocator.

These tests also represent best-case cache-behavior, because actual data pages were not being moved through the memory sub-system. Again, these structures should be re-examined in the future with a mixed workload with sub-optimal caching behaviors.

3 Going Forward with Improvements to the Page-Cache

As far as improving the radix-tree, there does not appear to be any reason to outright replace the current implementation, however performance could probably be improved for the class of workloads desiring concurrent access to the radix-tree structure by improving the locking behaviors for the radix-tree. As an example, a database system using large files for storing tables and using the page-cache could run into this issue.

3.1 A Lockless Design

Ultimately, it would be beneficial to implement a fully lockless design (for readers) using a Read-Copy-Update (RCU) approach [11]. This would allow the tree to better scale with many concurrent readers, and should not cause any difference in performance for a writers. This could cause a number of issues and race-conditions where readers seeing “stale” data could cause problems, and these issues must be

fully explored and understood before an implementation can be attempted.

Of the data-structures mentioned above, the radix-tree and the extendible hash-table would be the best structures suited for a lockless design, while the binary-tree structures are somewhat more difficult to modify for RCU.

In the case of the extendible hash-table, there are two cases to consider: bucket-splits and directory-expansion. In the case of bucket-splits, two new buckets are typically allocated to replace the original, so the original could be left in place for other readers, while the writing thread copied the data from the original bucket to the new ones and then updated the pointers in the directory. The race between readers looking at the directory and seeing the original bucket and seeing one of the new buckets is not problematic, since in either case, the appropriate data will be in whichever bucket is seen. The release of the memory for the old bucket would simply have to wait until all processors had reached a quiescent state. In the case of the directory expansion (or contraction) a similar technique would apply, where the writing thread works to update the new directory while leaving the old one in place. Then it can update the pointer to the directory after it finishes and use a deferred release for the old directory.

For the radix-tree, the main update case is radix-tree extension, where a new offset is inserted which requires an increase in the height of the tree. Luckily, the radix-tree is fairly simple and does not require complex restructuring in this case, but instead merely adds new levels ontop of the exisiting tree. So, in this case the writer thread creates these new nodes and sets them up while letting concurrent readers see the pre-exisiting tree, then when all of the new radix-nodes are set up, the height of the tree can be incremented and a new root installed. There is one problem with doing this today, the radix-tree root object currently consists of three fields including the height and a pointer to the root. For the RCU design to work, it must be able to atomically update a single field for the readers to look at, however both the height and the root pointer require updates. The solution to this is to add another level of indirection and simply keep that information in a separate dynamic object.

3.2 An Evolutionary Improvement

An alternative approach using gang-lookups, which is more evolutionary with respect to the current locking design, was suggested by Suparna Bhattacharya[1].

The current locking design works one page at a time where the radix-tree lock is acquired and released for each page locked. This is one reason why the rwlock approach may not be faster, since it uses an atomic operation both on acquisition and release whereas a spin-lock only uses one atomic operation on a successful lock acquisition. Her suggestion was to instead use a gang-lookup and lock each page requested one after the other before releasing the tree-lock. This approach would drastically reduce the number of costly atomic operations. This would come at the cost of increased lock hold times for the tree, but this could be mitigated somewhat by going back to the rwlock approach. Further, in this case the rwlock becomes a more effective solution since the number of unlock operations is drastically reduced.

method	spin-lock	rwlock
page by page	2n	3n
gang-lookup	n + 1	n + 2

[1] This idea was suggested in a private email to the authors, where she is working on converting the write-path to do something similar

Table 5. Table showing number of atomic operations required to lock n pages for the different locking strategies.

4 Summary

Overall, the performance of the current Linux 2.6 radix-tree is quite good as compared to the other data-structures chosen. Probably the area which is most ripe for improvement is the locking strategy for the radix-tree. A few different alternatives have been suggested, and hopefully by using these or other approaches, page-cache performance can be improved so that it even better than it is today.

5 Legal

6 References

[1] Cormen, T., *Algorithms* , Second Edition, MIT Press, 2001.

[2] Wirth, N., *Algorithms + Data Structures = Programs*, Prentice-Hall.

[3] Seidel, R., Aragon, C., *Randomized Search Trees*, Algorithmica 16, 1996.

[4] Andersson, A., Nielsson, S., *A New Efficient Radix Sort*, FOCS, 1994.

[5] Weiss, M., *Data Structures and C Programs* Addison-Wesley, 1997.

[6] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. *Extendible hashing—a fast access method for dynamic files*, September 1979, ACM Transactions on Database Systems, 4(3):315–344.

[7] Hellwig, C. *[PATCH] Radix-tree pagecache for 2.5*, January 2001, `http://www.ussg.iu.edu/hypermail/linux/kernel/0201.3/1234.html`

[8] Morton, A. *2.5.67-mm1*, April 2003, `http://www.uwsg.iu.edu/hypermail/linux/kernel/0304.1/0049.html`.

[9] Morton, A. *Re: 67-mjb2 vs 68-mjb1 (sdet degredation)*, April 2003, `http://www.cs.helsinki.fi/linux/linux-kernel/2003-16/0426.html`.

[10] Morton, A. *Re: [PATCH] Fixing address space lock contention in 2.6.11*, March 2005, `http://www.ussg.iu.edu/hypermail/linux/kernel/0503.0/1098.html`.

[11] McKenney, P., Appavoo, J., et al. *Read-Copy Update*, July 2001, Ottawa Linux Symposium.

[12] Pfaff, B. *GNU Libavl 2.0.2 Documentation* `http://www.stanford.edu/~blp/avl/libavl.html/`.

Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*