Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Review Committee

Proceedings Formatting Team

Concurrent Pagecache

Peter Zijlstra
Red Hat
pzijlstr@redhat.com

Abstract

In this paper we present a concurrent pagecache for Linux, which is a continuation of the existing lockless pagecache work [5].

Currently the pagecache is protected by a reader/writer lock. The lockless pagecache work focuses on removing the reader lock, however, this paper presents a method to break the write side of the lock. Firstly, since the pagecache is centered around the radix tree, it is necessary to alter the radix tree in order to support concurrent modifications of the data structure. Secondly, we adapt the pagecache, by removing all non-radix tree consumers of the lock's protection, and extend the pageflag, introduced by the lockless pagecache, into a second per page lock. Then we can fully utilize the radix tree's new functionality to obtain a concurrent pagecache. Finally, we analyze the improvements in performance and scalability.

1 Introduction

In order to provide some background for this work, we will give a quick sketch of the Linux memory management. For a more in depth treatment see Mel Gorman's excellent book on the subject [2].

1.1 Pagecache

As we know, in Linux, the pagecache caches on-disk data in memory per file. So the Linux pagecache stores and retrieves disk pages based on the `(inode, offset)`-tuple. This per inode page-index is implemented with a radix tree [3].

The typical consumer of this functionality is the VFS, Virtual File-System. Some system-interfaces, such as `read(2)` and `write(2)`, as well as `mmap(2)`, operate on the pagecache. These instantiate pages for the pagecache when needed, after which, the newly allocated pages are inserted into the pagecache, and possibly filled with data read from disk.

1.1.1 Page Frames

Each physical page of memory is managed by a `struct page`. This structure contains the minimum required information to identify a piece of data, such as a pointer to the inode, and the offset therein. It also contains some management state, a reference counter for instance, to control the page's life-time, as well as various bits to indicate status, such as `PG_uptodate` and `PG_dirty`. It is these page structures which are indexed in the radix tree.

1.1.2 Page Reclaim

In a situation when a free page of memory is requested, and the free pages are exhausted, a used page needs to be reclaimed. However the pagecache memory can only be reclaimed when it is clean, that is, if the in-memory content corresponds to the on-disk version.

When the page is not clean, it is called dirty, and requires data to be written back to disk. The problem of finding all the dirty pages in a file is solved by *tags*, which is a unique addition to the Linux radix tree (see Section 2.1).

1.2 Motivation

The lockless pagecache work by Nick Piggin [5] shows that the SMP scalability of the pagecache is greatly improved by reducing the dependency on high-level locks. However, his work focuses on lookups.

While lookups are the most frequent operation performed, other operations on the pagecache can still form a significant amount of the total operations performed.

Therefore, it is necessary to investigate the other operations as well. They are:

- *insert* an item into the tree;
- *update* an existing slot to point at a new item;
- *remove* an item from the tree;
- *set a tag* on an existing item;
- *clear a tag* on an existing item.

2 Radix Tree

The radix tree deserves a thorough discussion, as it is the primary data structure of the pagecache.

The radix tree is a common dictionary-style data structure, also know as Patricia Trie or crit bit tree. The Linux kernel uses a version which operates on fixed-length input, namely an `unsigned long`. Each level represents a fixed number of bits of this input space (usually 6 bits per tree level - which gives a maximum tree height of $\lceil 64/6 \rceil = 11$ on 64-bit machines). See Figure 1 for a representation of a radix tree with nodes of order 2, mapping an 8-bit value.

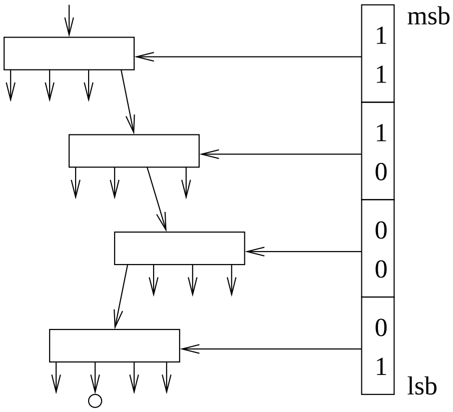

Figure 1: 8-bit radix tree

2.1 Tags

A unique feature of the Linux radix tree is its *tags*-extension. Each tag is basically a bitmap index on top of the radix tree. Tags can be used in conjunction with gang lookups to find all pages which have a given tag set within a given range.

The tags are maintained in per-node bitmaps, so that on each given level we can determine whether or not the next level has at least a single tag set. Figure 2 shows the same tree as before, but now with two tags (denoted by an open and closed bullet respectively).

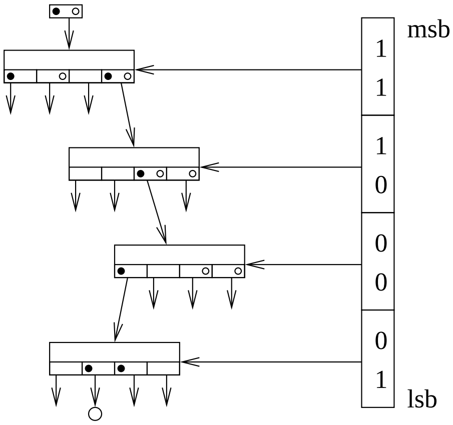

Figure 2: 8-bit radix tree with 2 bitmap indices

2.2 Concurrency Control

Traditionally, the radix tree is locked as a whole, and does not support concurrent operations. Lock contention of this high-level lock is where the scalability problems come from.

Linux currently uses a reader/writer lock, called `tree_lock`, to protect the tree. However, on large SMP machines it still does not scale properly due to cache line bouncing; the lock also fully serialises all modifications.

2.3 RCU Radix Tree

The RCU radix tree enables fully concurrent lookups (cf. [4]), which is done by exploiting the Read-Copy-Update technique [1].

RCU basically requires us to atomically flip pointers from one version of a (partial) data structure to a new version, while it will keep the old one around until the system passes through a quiescent state. At this point it is guaranteed that there are no more users of the old structure, and therefore it can be freed safely.

However, the radix tree structure is sufficiently static, so that often modifications are nothing but a single atomic change to the node. In this case we can just keep using the same node.

The radix-tree modifications still need to be serialised with respect to each other. The lookups, however, are no longer serialised with respect to modifications.

This allows us to replace the reader/writer lock with a regular one, and thus reduce the cache-line bouncing by not requiring an exclusive access to the cache line for the lookups.

2.4 Concurrent Radix Tree

With lookups fully concurrent, modifying operations become a limiting factor. The main idea is to 'break' the tree lock into many small locks.[1]

The obvious next candidate for locking would be the nodes.

When we study the operations in detail, we see that they fall into two categories:

- uni-directional;
- bi-directional.

The most simple of the two is the uni-directional operations; they perform only a single traversal of the tree: from the root to a leaf node. These include: *insert*, *update* and *set tag*.

The bi-directional operations are more complex, since they tend to go back up the tree after reaching the leaf node. These are the remaining operations: *remove* and *clear tag*.

2.4.1 Ladder Locking aka Lock-Coupling

This technique, which is frequently used in the database world, allows us to walk a node-locked tree in a single direction (bi-directional traffic would generate deadlocks).

If all modifiers alter the tree top-to-bottom, and hold a lock on the node which is being modified, then walking down is as simple as taking a lock on a child, while holding the node locked. We release the node lock as soon as the child is locked.

In this situation concurrency is possible, since another operation can start its descent as soon as the root node is unlocked. If their paths do not have another node in common, it might even finish before the operation which started earlier. The worst case, however, is pipelined operation, which is still a lot better than a fully serialised one.

2.4.2 Path Locking

Obviously, this model breaks down when we need to walk back up the tree again, for it will introduce cyclic lock dependencies. This implies that we cannot release the locks as we go down, which will seriously hinder concurrent modifications.

An inspection of the relevant operations shows that the upwards traversal has distinct termination conditions. If these conditions were reversed, so that we could positively identify the termination points during the downward traversal, then we could release all locks upwards of these points.

In the worst case, we will hold the root node lock, yet for non-degenerate trees the average case allows for good concurrency due to availability of termination points.

clear tag Clearing a tag in the radix tree involves walking down the tree, locating the item and clearing its tag. Then we go back up the tree clearing tags, as long as the child node has no tagged items.

Thus, the termination condition states:

> we terminate the upward traversal if we encounter a node, which still has one or more entries with the tag set after clearing one.

Changing this condition, in order to identify the termination points during downwards traversal, gives:

> the upwards traversal will terminate at nodes which have more tagged items than the one we are potentially clearing.

So, whenever we encounter such a node, it is clear that we will never pass it on our way back up the tree, therefore we can drop all the locks above it.

[1] Ideally we reduce the locks so far that we end up with single atomic operations. However tags and deletion (the back tracking operations) seem to make this impossible; this is still being researched.

remove Element removal is a little more involved: we need to remove all the tags for a designated item, as well as remove unused nodes. Its termination condition captures both of these aspects.

The following termination condition needs to be satisfied when walking back up the tree:

> the upward traversal is terminated when we encounter a node which is not empty, and none of the tags are unused.

The condition, identifying such point during the downward traversal, is given by:

> we terminate upwards traversal when a node that has more than two children is encountered, and for each tag it has more items than the ones we are potentially clearing.

Again, this condition identifies points that will never be crossed on the traversal back up the tree.

So, with these details worked out, we see that a node-locked tree can achieve adequate concurrency for most operations.

2.4.3 API

Concurrent modifications require multiple locking contexts and a way to track them.

The operation that has the richest semantics is `radix_tree_lookup_slot()`. It is used for speculative lookup, since that requires `rcu_dereference()` to be applied to the obtained slot. The update operation is performed using the same radix tree function, but by applying `rcu_assign_pointer()` to the resulting slot.

When used for update, the encompassing node should still be locked after return of `radix_tree_lookup_slot()`. Hence, clearly, we cannot hide the locking in the `radix_tree_*()` function calls.

Thus we need an API which elegantly captures both cases; lookup and modification.

We also prefer to retain as much of the old API as possible, in order to leave the other radix tree users undisturbed.

Finally, we would like a `CONFIG` option to disable the per-node locking for those environments where the increased memory footprint of the nodes is prohibitive.

We take the current pattern for lookups as an example:

```
struct page **slot, *page;

rcu_read_lock();
slot = radix_tree_lookup_slot(
        &mapping->page_tree, index);
page = rcu_dereference(*slot);
rcu_read_unlock();
```

Contrary to lookups, which only have global state (the RCU quiescent state), the modifications need to keep track of which locks are held. As explained before, this state must be external to the operations, thus we will need to instantiate a local context to track these locks.

```
struct page **slot;
DEFINE_RADIX_TREE_CONTEXT(ctx,
        &mapping->page_tree);

radix_tree_lock(&ctx);
slot = radix_tree_lookup_slot(
        ctx.tree, index);
rcu_assign_pointer(*slot, new_page);
radix_tree_unlock(&ctx);
```

As can be seen above, `radix_tree_lock()` operation locks the root node. By giving `ctx.tree` as the tree root instead of `&mapping->page_tree`, we pass the local context on, in order to track the held locks. This is done by using the lower bit of the pointer as a type field.

Then we adapt the modifying operations, in order to move the lock downwards:

```
void **radix_tree_lookup_slot(
        struct radix_tree *root,
        unsigned long index)
{
    ...
    RADIX_TREE_CONTEXT(context, root);
    ...
    do {
        ...
        /* move the lock down the tree */
        radix_ladder_lock(context, node);
        ...
    } while (height > 0);
    ...
}
```

The RADIX_TREE_CONTEXT() macro extracts the context and the actual root pointer.

Note that unmodified operations will be fully exclusive because they do not move the lock downwards.

This scheme captures all the requirements mentioned at the beginning of this section. The old API is retained by making the new parts fully optional; and by moving most of the locking specific functionality into a few macros and functions, it is possible to disable the fine grained locking at compile time using a CONFIG option.

3 Pagecache Synchronisation

The lockless pagecache paper [5] discusses the pagecache synchronisation in terms of guarantees provided by the read and write side of the pagecache lock. The read side provides the following guarantees (by excluding modifications):

- the *existence* guarantee;
- the *accuracy* guarantee.

The write side provides one additional guarantee (by being fully exclusive), namely:

- the *no-new-reference* guarantee.

The existence guarantee ensures that an object will exist during a given period. That is, the struct page found must remain valid. The read lock trivially guarantees this by excluding all modifications.

The accuracy guarantee adds to this by ensuring that not only will the object stay valid, but it will also stay in the pagecache. The existence only avoids deallocation, while the accuracy ensures that it keeps referring to the same (inode, offset)-tuple during the entire time. Once again, this guarantee is provided by the read lock by excluding all modifications.

The no-new-reference guarantee captures the fully exclusive state of the write lock. It excludes lookups from obtaining a reference. This is especially relevant for element removal.

3.1 Lockless Pagecache

The lockless pagecache focuses on providing the guarantees, introduced above, in view of the full concurrency of RCU lookups. That is, RCU lookups are not excluded by holding the tree lock.

3.1.1 Existence

The existence guarantee is trivially satisfied by observing that the page structures have a static relation with the actual pages to which they refer. Therefore they are never deallocated.

A free page still has an associated struct page, which is used by the page allocator to manage the free page. A free page's reference count is 0 by definition.

3.1.2 Accuracy

The accuracy guarantee is satisfied by using a *speculative-get* operation, which tries to get a reference on the page returned by the RCU lookup. If we did obtain a reference, we must verify that it is indeed the page requested. If either the speculative get, or the verification fails, e.g. the page was freed and possibly reused already, then we retry the whole sequence.

The important detail here is the *try-to-get-a-reference* operation, since we need to close a race with freeing pages, i.e. we need to avoid free pages from temporarily having a non-zero reference count. The reference count is modified by using atomic operations, and to close the race we need an atomic_inc_not_zero() operation, which will fail to increment when the counter is zero.

3.1.3 No New Reference

The no-new-reference guarantee is met by introducing a new page flag, PG_nonewrefs, which is used to synchronise lookups with modifying operations. That is, the *speculative get* should not return until this flag is clear. This allows atomic removal of elements which have a non-zero reference count (e.g. the pagecache itself might still have a reference).

3.1.4 Tree Lock

When we re-implement all lookup operations to take advantage of the *speculative get*, and re-implement the modifying operations to use PG_nonewrefs, then the read-side of the tree_lock will have no users left. Hence we can change it into a regular spinlock.

3.2 Concurrent Pagecache

The lockless pagecache leaves us with a single big lock serialising all modifications to the radix tree. However, with the adaptations to the radix tree, discussed in Section 2.4, the serialisation, required to meet the synchronisation guarantees of Section 3, is per page.

3.2.1 PG_nonewrefs vs. PG_locked

Since we need to set/clear PG_nonewrefs around most modifying operations, we might as well do it around all modifying operations, and change PG_nonewrefs into an exclusion primitive, which serialises each individual pagecache page modification.

We can't reuse PG_locked for this because, they have a different place in the locking hierarchy.

```
inode->i_mutex
  inode->i_alloc_sem
  mm->mmap_sem
    PG_locked
      mapping->i_mmap_lock
        anon_vma->lock
          mm->page_table_lock or pte_lock
            zone->lru_lock
            swap_lock
              mmlist_lock
              mapping->private_lock
              inode_lock
                sb_lock
                mapping->tree_lock
```

Figure 3: mm locking hierarchy

Figure 3 represents the locking hierarchy. As we can see, PG_locked is an upper level lock, whereas the tree_lock (now to be replaced by PG_nonewrefs) is at the bottom.

Also, PG_locked is a sleeping lock, whereas tree_lock must be a spinning lock.

3.2.2 Tree-Lock Users

Before we can fully remove the tree_lock, we need to make sure that there are no other users left.

A close scrutiny reveals that nr_pages is also serialised by the tree_lock. This counter needs to provide its own serialisation, for we take no lock covering the whole inode. Changing it to an atomic_long_t is the easiest way to achieve this.

Another unrelated user of the tree lock is architecture specific dcache flushing. However, since its use of the tree lock is a pure lock overload, it does not depend on any other uses of the lock. We preserve this usage and rename the lock to priv_lock.

3.2.3 No New Reference

The lockless pagecache sets and clears PG_nonewrefs around insertion operations, in order to avoid half inserted pages to be exposed to readers. However, the insertion could be done without PG_nonewrefs by properly ordering the operations.

On the other hand, the deletion fully relies on PG_nonewrefs. It is used to hold off the return of the *speculative get* until the page is fully removed. Then the accuracy check, after obtaining the speculative reference, will find that the page is not the one we requested, and will release the reference and re-try the operation. We cannot rely on atomic_inc_not_zero() failing in this case, because the pagecache itself still has a reference on the page.

By changing PG_nonewrefs into a bit-spinlock and using it around all modifying operations, thus serialising the pagecache on page level, we satisfy the requirements of both the lockless and the concurrent pagecache.

4 Performance

In order to benchmark the concurrent radix tree, a new kernel module is made. This kernel module exercises the modifying operations concurrently.

This module spawns a number of kernel threads, each of which applies a radix tree operation on a number of indices. Two range modes were tested: interleaved and sequential. The interleaved mode makes each thread iterate over the whole range, and pick only those elements which match *i mod nr_threads* = *nr_thread*. The sequential mode divides the full range into *nr_thread* separate sub ranges.

These two patterns should be able to highlight the impact of cache-line bouncing. The interleaved pattern has a maximal cache-line overlap, whereas the sequential pattern has a minimal cache-line overlap.

4.1 Results

Here we present results obtained by running the new kernel module, mentioned above, on a 2-way x86-64 machine, over a range of 16777216 items. The numbers represent the runtime (in seconds).

The interleaved mode gives:

operation	serial	concurrent	gain
insert	16.006	19.485	-22%
tag	14.989	15.538	-4%
untag	17.515	16.982	3%
remove	14.213	16.506	-16%

The sequential mode gives:

operation	serial	concurrent	gain
insert	15.768	14.792	6%
tag	15.110	14.581	4%
untag	18.138	15.027	17%
remove	14.607	16.250	-11%

As we see from the results, the lock induced cache-line bouncing is a real problem, even on small SMP systems. The locking overhead is not prohibitive however.

5 Optimistic Locking

Now that the pagecache is page-locked, and the basic concurrency control algorithms are in place, effort can be put into investigating more optimistic locking rules for the radix tree.

For example, for the insertion we can do an RCU lookup of the lowest possible matching node, then take its lock and verify that the node is still valid. After this, we continue the operation in a regular locked fashion. By doing this we would avoid locking the upper nodes in many cases, and thereby significantly reduce cache-line bouncing.

Something similar can be done for the item removal: find the lowest termination point during an RCU traversal, lock it and verify its validity. Then continue as a regular path-locked operation.

In each case, when the validation fails, the operation restarts as a fully locked operation.

Since these two examples cover both the ladder-locking model in Section 2.4.1, and the path-locking model in Section 2.4.2, they can be generalised to cover all other modifying operations.

5.1 Results

Rerunning the kernel module, in order to test the concurrent radix tree performance with this new optimistic locking model, yields much better results.

The interleaved mode gives:

operation	serial	optimistic	gain
insert	16.006	12.034	25%
tag	14.989	7.417	51%
untag	17.515	4.135	76%
remove	14.213	6.529	54%

The sequential mode gives:

operation	serial	optimistic	gain
insert	15.768	3.446	78%
tag	15.110	5.359	65%
untag	18.138	4.126	77%
remove	14.607	6.488	56%

These results are quite promising for larger SMP machines.

Now we see that, during the interleaved test, the threads slowly drifted apart, thus naturally avoiding cache-line bouncing.

6 Availability

This work resulted in a patch-set for the Linux kernel, and is available at:

`http://programming.kicks-ass.net/kernel-patches/concurrent-pagecache/`

References

[1] Wikipedia, *Read-Copy-Update* `http://en.wikipedia.org/wiki/RCU`

[2] M. Gorman, *Understanding the Linux Virtual Memory Manager*, 2004.

[3] Wikipedia, *Radix Tree*, `http://en.wikipedia.org/wiki/Radix_tree`

[4] N. Piggin, *RCU Radix Tree*, `http://www.kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.16-rc5/radix-intro.pdf`

[5] N. Piggin *A Lockless Pagecache in Linux - Introduction, Progress, Performance*, Proceedings of the Ottawa Linux Symposium 2006, pp. 241–254.