A Lockless Pagecache in Linux—Introduction, Progress, Performance

Nick Piggin
SUSE Labs, Novell Inc.
npiggin@suse.de

Abstract

Critical Linux pagecache operations can be made lockless to provide improvements in performance and scalability. I examine some existing pagecache synchronisation designs, then introduce my lockless pagecache for Linux. Performance and scalability of the implementation is analysed and compared with that of other schemes—this involves a comparison of benchmark results from a range of machines and workloads. Finally, I give a progress report on the present state of the work.

1 Introduction

The focus of this paper is to improve the multiprocessor scalability of the Linux pagecache without compromising other performance characteristics.

1.1 Pagecache

The pagecache is a transparent filesystem cache. The fundamental functionality required of the pagecache is to manage memory pages that hold inode[1] data, and which are stored and retrieved according to their *(inode, offset)* tuple.

[1] An inode essentially represents a file's contents.

Many modern UNIX-like operating systems, including Linux, have the concept of a pagecache, which obsoleted the buffer cache when it was introduced with *SVR4 UNIX*.

A common use-case for the pagecache is a page-sized and aligned read(2) system call; the Linux kernel performs the following operations:

1. System call entry into the VFS (kernel's filesystem subsystem).
2. VFS determines which inode is specified by the given file descriptor.
3. VFS calls into the memory manager to read the required *(inode, offset)*.
4. The memory manager queries the pagecache for the page. If the page does not exist, go to 5; if the page not valid, go to 7; otherwise go to 8.
5. memory manager allocates a new page, mark its contents invalid, and store this new page in the pagecache, representing the given *(inode, offset)*.
6. memory manager initiates a filesystem read to populate the page.
7. thread will now wait until completion of the read (which marks the page contents as valid).

8. memory manager will copy the required data to the VFS for the read call.

1.2 Linux pagecache history

1.2.1 Linux 2.4: Globally locked pagecache

Linux 2.4 uses a fixed sized global hash-chain data structure in order to store pagecache pages based on their *(inode, offset)* tuple. Pagecache pages are also present on per-inode lists of clean and dirty pages. Access to these lists and the hash table is synchronised by a single global spinlock.

This global spinlock is one of the largest scalability bottlenecks in the Linux 2.4 kernels for many workloads. On a workload such as dbench[2] [8], Juergen Doelle [1] demonstrated the poor scalability of this scheme, with almost no performance improvement when moving from 4 to 8 CPUs.

CPUs	throughput (normalised)
1	1.00
2	1.51
4	2.15
8	2.27

1.2.2 Linux 2.4: Molnar/Miller scalable pagecache

Ingo Molnar and David Miller [4] attempted to address the problem of the global pagecache lock with a synchronisation scheme which protected the hash table with an individual lock per hash-bucket, and protected per-inode lists (which contain clean/dirty pages) with a per-inode lock.

[2] dbench is a file server benchmark

This design is problematic because it introduces another layer of locking to the system, thus increasing the number of lock operations and the cache footprint of a typical path through the kernel. There is also complexity introduced in order to avoid lock ordering deadlocks.

The Molnar/Miller pagecache was never used in the Linux kernel, however it may have provided ideas which paved the way for the Velikov/Hellwig design.

1.2.3 Linux 2.6: Velikov/Hellwig radix-tree pagecache

Momchil Velikov and Christoph Hellwig designed a radix-tree based pagecache architecture, which is used by current Linux 2.6 kernels. Pagecache pages are stored in a variable height radix-tree, with one radix-tree per inode, and each tree is indexed by the page's offset within the inode. The per-inode page lists were retained for some time after its inclusion into the kernel. Andrew Morton subsequently modified this design to remove these lists: the radix-tree now maintains a hierarchy of 'tags' for each node, one of which indicates dirty pagecache, to speed up searches for dirty pages.

Each inode structure has a spinlock, `tree_lock`, which is used to synchronise concurrent access and modification of the radix-tree, and to control access to the pagecache in general.

1.2.4 Other operating systems

OpenSolaris uses a complex arrangement of hash tables and hashed locks in its pagecache implementation, which is in some ways similar to the Molnar/Miller scalable pagecache.

FreeBSD 6 uses a per-inode splay-tree and per-inode locking in its pagecache, in the same ba-

sic way as the Velikov/Hellwig radix-tree pagecache.

Most other free and open operating systems use either hashes or trees with lock based synchronisation, these are naturally suited to the application.

1.3 Linux memory management

An introduction to the relevant details of the Linux memory management implementation needs to be given, to provide the reader with background to understand the proposal for the lockless pagecache. These details are slightly simplified in places so as not to distract from the main concepts being introduced. For further reading, Mel Gorman [3] provides a thorough examination of memory management in Linux.

1.4 Memory, `struct page`

In Linux, every physical page frame that is to be used as RAM by the kernel is represented with a corresponding `struct page` structure. This structure contains fields `flags` for general flags, `_count` is a reference count, and various other data associated with the status and management of the page frame.

The `struct page` is the usual way to refer to a page, and the pagecache is no exception. It is the `struct page` representing a given pagecache page that is stored in the pagecache's radix-tree.

Figure 1 gives an idea of how the `struct page` relates to page-frames.[3]

[3] 'Two separate columns' is slightly inaccurate because actually the `mem_map` array of `struct page` is itself stored in physical memory frames, and it may not be implemented as a single contiguous array, however that is inconsequential to this discussion.

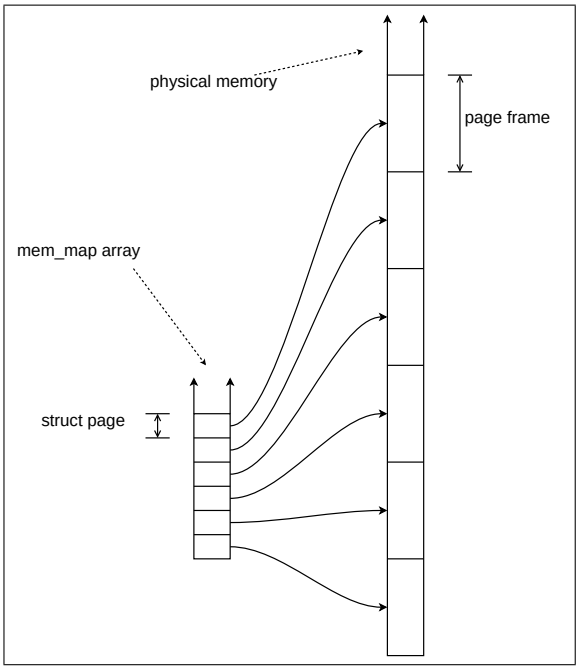

Figure 1: How `struct page` relates to physical memory pages

1.4.1 Page lifetimes, refcounting

`struct page` has a reference count, `_count`, which is 0 when the page is free, and is set to 1 when the page is allocated.

When some part of the kernel has finished with a page and would like to free it, `__free_pages` (shown in Figure 2) or a similar function would be called. This atomically decrements the refcount and if that caused it to become zero, the page is returned to the allocator. There is a `get_page` function, which increases the refcount of an allocated page.

```
 1: void __free_pages(struct page *page,
 2:         unsigned int order)
 3: {
 4:     if (put_page_testzero(page)) {
 5:         if (order == 0)
 6:             free_hot_page(page);
 7:         else
 8:             __free_pages_ok(page, order);
 9:     }
10: }
```

Figure 2: `__free_pages` function in Linux

1.4.2 Dirty pages

A pagecache page is considered dirty if its contents are more recent than the contents of the filesystem which it is caching. A pagecache page would become dirty if a program invokes the write system call to modify the data in an inode. If a pagecache page is not dirty then it is considered *clean*—it is storing a copy of file data that is *identical* to its corresponding data in the filesystem.

A clean page can be discarded from the pagecache, because if it is required in future it will be restored from the filesystem. Dirty pages can not be discarded from the pagecache because that would result in data loss as the contents of the disk are older than those in memory. A dirty page can be cleaned by directing the filesystem to write the contents of the page to its backing store.

1.4.3 Page reclaim

During the course of a system's operation, if memory becomes full, it will attempt to *reclaim* pagecache pages in order to satisfy requests for memory.

Clean pagecache pages are reclaimed by simply discarding them. It is important to ensure that the pages are clean and that they have no references to them before being reclaimed. A reference to the page indicates it is in use—that user may be in the process of dirtying the page even if it is now clean.

This detail becomes important later on, because Linux currently relies on the per-inode `tree_lock` to exclude read-side code when performing these tests.

2 Lockless pagecache in Linux

This section will propose a model for a lockless pagecache in Linux. By lockless, it is meant that pagecache lookup (read-side) operations will be performed without taking a lock. Insertion and removal of pages, and 'tag lookups' are still performed with the same locking—these operations are usually associated with less frequent operations such as IO, truncation, and page reclaim so are less important.

2.1 Lockless data structure

One fundamental protection provided by the `tree_lock` spinlock that is taken by pagecache lookup functions, is the protection of the pagecache data structure. Hence, one thing required for lockless pagecache is a lockless data structure.

Simple lockless data structures such as linked lists and hashes are already used in Linux. Lockless hash lookups are used in places such as the *pid hash* and *dcache hash*, however changing to a hash table would be a step back from the per-inode radix-tree structure in Linux 2.6.[4] What's more, fundamentally changing the nature of pagecache data structure is beyond the scope of this paper, which is to examine just pagecache *synchronisation* designs.

A lockless radix-tree using RCU has been developed [7] to be used as the lockless data structure. Lockless radix-tree lookups can return stale data, data that no longer exists in the radix-tree. It is up to the callers to deal with stale data.

[4] O(log(N)) vs O(N) lookup complexity is one reason.

2.2 Linux pagecache synchronisation introduction

With the ability to retrieve pagecache pages from the radix-tree without taking a lock, the problem of synchronising the pagecache itself still exists. In Linux, this pagecache synchronisation is performed using the same lock that is used for data structure synchronisation.

The following is a description of the pagecache synchronisation functions performed by `tree_lock` in Linux 2.6. When held for reading, the `inode`'s `tree_lock` in Linux 2.6 is used to provide the following pagecache synchronisation guarantees (by providing exclusion from writers):

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

When held for writing, `tree_lock` *additionally* provides a guarantee that no new references to the page is given (by also providing exclusion from readers):

- the *no new reference* guarantee.

2.2.1 Existence guarantee

> *Providing existence guarantees is likely the most difficult aspect of concurrency control. The traditional way of eliminating races between one thread trying to lock an object and another deallocating it, is to ensure that all references to an object are protected by their own lock* [2]

An existence guarantee provides the guarantee that an object will continue to exist and be valid for a given period, typically for the time that a sequence of operations are performed on that object.

Linux pagecache lookup functions require the guaranteed existence of a `struct page` in pagecache, from the time it is looked up via the radix-tree, until its reference count can be incremented.[5] This guarantee is provided by holding the `tree_lock` for reading.

A problem of existence

The notion of an existence guarantee can be difficult to understand at first; with traditional lock based synchronisation, existence is almost always implied at a fundamental level. Existence is best explained by examining the consequences of its absence.

```
 1: struct page *find_get_page(struct address_space *mapping,
 2:                            unsigned long offset)
 3: {
 4:     struct page *page;
 5:
 6:     read_lock_irq(&mapping->tree_lock);
 7:     page = radix_tree_lookup(&mapping->page_tree, offset);
 8:     if (page)
 9:         page_cache_get(page);
10:     read_unlock_irq(&mapping->tree_lock);
11:     return page;
12: }
```

Figure 3: `find_get_page`, a pagecache lookup function in Linux

Figure 3 shows a commonly used pagecache lookup function in Linux. At line 8, `page_cache_get` elevates the reference count of the `struct page`, which prevents the page from being freed. However if the `tree_lock` were not held during this operation, then after executing line 6 and before executing line 8, another CPU can concurrently remove the page from the pagecache and free it. When the original CPU does execute line 8, it would be incrementing the reference count of a `struct page` which has been freed and possibly allocated for some other use.

[5] The elevated refcount then guarantees existence.

2.2.2 Accuracy guarantee

After looking up a page in the pagecache radix-tree, the `tree_lock` held for reading provides the guarantee that the page will remain in the pagecache until the lock is released.

The accuracy guarantee is subtly different from the existence guarantee. The existence guarantee only provides that the page remains allocated, it may still be removed from the pagecache should its inode be truncated.

2.2.3 No new reference guarantee

The no new reference guarantee ensures that no pagecache lookup routines will be allowed to take a new reference to a particular page. This guarantee is provided by holding `tree_lock` for writing, thereby excluding those lookup functions, which all take the lock for reading.

This guarantee is important for page reclaim (and page migration). To reclaim a page, the memory manager needs to ensure nobody can take a new reference to the page before removing it from pagecache (see 1.4.3).

2.3 Providing guarantees without locking

Here, the fundamental concepts of the lockless pagecache synchronisation design are explained. That is, the methods that allow the removal of the per-inode `tree_lock` from some places where it is currently taken for *reading*. In order to show that correctness is maintained, it must be demonstrated that pagecache synchronisation requirements, described above in 2.2, can be provided by the lockless design.

2.3.1 Permanence of `struct page` (existence guarantee)

Taking a reference on a pagecache `struct page` without holding any locks relies on a key observation which alleviates the requirement of a strict existence guarantee. This is a central idea behind lockless pagecache: *a `struct page` itself is never actually allocated or freed, only its associated page frame is*. This is made clear when considering that free page frames retain their associated `struct page`, it is even used by the page allocator to manage the free page frame.[6]

2.3.2 Speculative pagecache references (accuracy guarantee)

With the necessity for an existence guarantee alleviated, it is possible to 'speculatively' elevate the `struct page's` reference count, then verify that the operation was performed on the correct page. If the page is no longer at the same position in the pagecache after the speculative reference, then it must have been replaced or deleted, so the speculative reference is dropped, and the whole operation retried.

There is an interesting corner case to consider, because it may not be obviously correct immediately. Suppose a particular pagecache page is removed from the pagecache and freed, but is then re-allocated and used as a pagecache page for exactly the same *(inode, offset)* as it has been previously. Now suppose that the speculative reference loads the address of the `struct page` when it is in the pagecache the first time around, but the reference count is actually incremented after the page has been freed and re-allocated. The check to see whether the page is still at the right place in the pagecache then

[6] One way the page allocator uses the `struct page` is to keep track of the page on 'free lists.'

finds that to indeed be the case, despite the page having been freed and reallocated.

This case turns out to be no problem, because it is equally possible that the initial address load had been slightly delayed and found the page *after* it had been reused. The important thing is just that the `struct page` that actually had its refcount increased is verified to be correct.

In one pagecache lookup function, `find_lock_page`, the accuracy requirement goes beyond increasing the refcount when the page is known to be in the pagecache. This is addressed in subsection 2.5.

2.3.3 Lookup synchronisation point (no new reference guarantee)

The 'no new reference' guarantee traditionally provided by holding the `tree_lock` for writing is no longer enforced due to the lookup side taking references without holding the lock for reading. This problem is overcome by introducing a new bit in the page's `flags` field. Code that requires the *no new reference guarantee* will set this bit. After a speculative reference is taken on a page, this bit will be checked and the operation retried if it was set.

Essentially the bit has become a synchronisation point and has taken over from the functionality provided by `tree_lock`. Importantly, it is not a *lock* that is taken by the read-side: it does not block writers, nor will it cause cacheline contention between multiple read-side lookups of the same page.

2.3.4 Guarantees in uniprocessor kernels

The Linux kernel offers a compilation configuration choice between uniprocessor (UP) or multiprocessor (SMP) kernels. The UP kernel option allows many optimisations in the resulting compiled code, in particular, spinlocks get optimised away because there is no need to prevent other processors from entering the critical section. It is important to note that it is still important to disable interrupts when providing critical sections with exclusion from interrupts.

A UP kernel already effectively has lockless pagecache lookup operations. The relatively complex mechanisms for providing pagecache synchronisation, described above, are not required for UP kernels. They are not required because all pagecache write-side operations are performed in process context and exclude interrupts while running. Read-side operations need only ensure that they are not interleaved with any other process context, which can be done so by having preemption disabled. Thus a special case can be made for UP kernels, which results in a lighter-weight lookup function.

2.3.5 Problems

There are subtle problems with this simplistic description of the mechanics of taking a speculative reference when relying on the permanence property of the `struct page`. They stem from the fact that the existence guarantee provided is not as strong as it could be. In particular, while the `struct page` itself does not get deallocated, it can be used in completely different ways depending on whether the page is allocated, and what part of the kernel has allocated the page.

Between the act of looking up the page and speculatively taking a reference on the page, it may have been removed from the pagecache, then freed, then allocated somewhere else. When taking the speculative reference, it is possible for the page to be in any state. The page may be free or re-allocated, perhaps for an entirely different purpose than pagecache.

Speculative references to free pages
One issue is free pages. Free pages have a refcount of zero and exist in the page allocator. Speculatively elevating the refcount of a free page poses a number of problems.

It can be difficult to tell if the page actually was free at that point (imagine a second speculative reference that had elevated the count from 0 to 1). When dropping the speculative reference it is essential that a free page is not freed *again*, when the count reaches zero.

Another problem is the possibility that the page might be allocated while a speculative reference has elevated the count, further complicating the task of determining the correct course of action to take when dropping a failed speculative reference.

All the problems associated with free pages are avoided by introducing the new atomic primitive `atomic_inc_not_zero`, to be used when taking speculative references. `atomic_inc_not_zero` increments the reference count only when it is not zero, and returns success or failure. This allows free pages to be detected and ignored.

Page refcounting uniformity
A second problem is one of 'page refcounting uniformity' throughout the kernel. By the time a speculative reference has been taken on a page, it may have been freed then allocated somewhere else (in which case `atomic_inc_not_zero` will succeed). This speculative reference must be dropped when it is discovered that the wrong page has been picked up. For this reason, it is important that the entire kernel treats the page's refcount in the same manner, and that dropping the last reference must free the page.

Page refcount instability
There is a third problem, introduced by the fact that any page taken from the allocator may have an unstable refcount. Before being allocated, the page may previously have been a pagecache page, and may have a speculative reference taken on it at any time.

To solve this problem, no part of the kernel should assume the refcount is stable, nor should non-atomic operations be used to manipulate the refcount. It can still be assumed that the refcount is be *greater than or equal to* the number of references that are *known* to be held at any point.

The *lookup synchronisation point* used to provide the 'no new reference' guarantee can be used, when necessary, to determine that the number of real references to a page is *less than or equal to* the refcount in the `struct page`.

2.3.6 Why RCU is not used for `struct page` existence guarantee

RCU is not used to provide existence guarantees for a pagecache page. While this would be possible, and would avoid many of the page reference counting problems encountered by relying on the permanence of `struct page` for existence, RCU has problems of its own.

RCU freeing would add an extra stage for pages to pass through before actually being freed. This stage would involve batching up pages into a list, and traversing the list again (after an RCU grace period) in order to actually free them. This scheme would have a number of problems:

- Visiting the page again will introduce overhead;
- within the grace period delay, the `struct page` could have been evicted from the CPU's, introducing cache misses when freeing the pages;

- the page allocator has per-CPU lists of free pages, which can be accessed lockless. Page allocator locks need only be taken when these lists overflow or underflow. The extra RCU stage will keep pages from reaching these per-CPU lists for some time. This will increase the incidence of underflow while the pages are being held, and of overflow when they are finally freed.

- the per-CPU lists attempt to keep track of pages which are likely to be cache-hot and those which are cache-cold, so they may be used appropriately. The extra RCU stage will reduce the effectiveness of these estimations.

- RCU can take some time to go through a quiescent state, this could be a problem in low memory conditions if pages aren't freed quickly enough.

Lockless pagecache does use RCU for the pagecache radix-tree nodes, however they are less affected by the above problems: they are much smaller than a page, and they are usually allocated and freed less often than pagecache pages.

2.3.7 page_cache_get_speculative

This subsection briefly introduces `page_cache_get_speculative`, which is is the core operation that implements pagecache synchronisation, according to the methods described above. Figure 4 is the actual C code for `page_cache_get_speculative`, with the simple uniprocessor implementation and some comments removed for clarity.

In lines 6-8, a pointer to the radix-tree's leaf-node slot is dereferenced, the function returns

```
 1: struct page *page_cache_get_speculative(struct page **pagep)
 2: {
 3:     struct page *page;
 4:
 5:  again:
 6:     page = rcu_dereference(*pagep);
 7:     if (unlikely(!page))
 8:         return NULL;
 9:
10:     if (unlikely(!get_page_unless_zero(page)))
11:         goto again; /* page has been freed */
12:
13:     while (unlikely(PageNoNewRefs(page)))
14:         cpu_relax();
15:
16:     smp_rmb();
17:
18:     if (unlikely(page != *pagep)) {
19:         /* page no longer at *pagep */
20:         put_page(page);
21:         goto again;
22:     }
23:
24:     return page;
25: }
```

Figure 4: `page_cache_get_speculative` function

NULL if the slot is empty, otherwise the slot contains a pointer to a `struct page`.

At line 10, the page's refcount is incremented if it was not previously 0; if it was, the operation is restarted.[7]

Line 13 busy-waits while the page's 'NoNewRefs' flag is set.[8]

When NoNewRefs is clear, lines 18–22 recheck that this page is present in pagecache.[9] If yes, then success and the page is returned; if no, the page's refcount is decremented (and will be freed if that caused it to reach 0), and the operation is restarted.

Note: `page_cache_get_speculative` relies on memory barriers to order memory operations correctly. Discussion of these barriers at this point would distract from the fundamental details of the operation, and as such will not be covered. The comments in the actual implementation explain all memory ordering in detail.

[7] this relies on the permanence of `struct page` and uniform page refcounting.

[8] The 'NoNewRefs' flag can be set to enforce the *no new references* guarantee.

[9] This recheck provides the accuracy guarantee.

2.4 Lockless pagecache operations

This section describes the re-implementation of Linux pagecache lookup functions, using the lockless radix-tree and the 'speculative get page' operation, without using locks.

```
 1: struct page *find_get_page(struct address_space *mapping,
 2:                            unsigned long offset)
 3: {
 4:     struct page **pagep;
 5:     struct page *page = NULL;
 6:
 7:     rcu_read_lock();
 8:     pagep = radix_tree_lookup_slot(&mapping->page_tree,
 9:                                    offset);
10:     if (pagep)
11:         page = page_cache_get_speculative(pagep);
12:     rcu_read_unlock();
13:     return page;
14: }
```

Figure 5: `Lockless find_get_page`

2.4.1 `find_get_page`

`find_get_page` has the following semantics, if the given pagecache coordinates (mapping,[10] offset):

- *always* contained the page, it must be returned;
- were always empty, NULL must be returned;
- ever contained a page, it may be returned;
- were ever empty, NULL may be returned.

If a page is to be returned, first its refcount is incremented *while it is in the pagecache*. `find_get_page` may return pages which are no longer in the pagecache, so there is no problem with the lockless radix-tree lookup returning stale data.

When a page has been found, `page_cache_get_speculative` can be used to increment its refcount and ensures the refcount was incremented while the page was in the pagecache. Figure 3 shows the locking version of the function, figure 5 is the lockless implementation.

2.5 `find_lock_page`

`find_lock_page` is similar to `find_get_page`, however it is also required to lock the page[11] while it is in pagecache.

The page lock actually pins a page in pagecache, unlike the refcount. This means that after taking the page lock, it is sufficient to subsequently recheck that the page indeed exists in the expected position in pagecache. In order to take the page lock, the page must be prevented from being freed concurrently. This existence guarantee is provided by first incrementing the page's refcount by calling the lockless `find_get_page`.

2.5.1 `find_get_pages`

The `find_get_pages` function finds up to a specified number of pages from a given offset in an inode, and elevates the refcount of each page found. The operation is performed completely under the `tree_lock`, which means that all returned pages were *all* in pagecache at the time each had their refcount incremented.

It is not possible to retain this atomicity without holding `tree_lock`. Instead of being replaced, a new function, `find_get_pages_nonatomic`, is introduced which provides only `find_get_page` semantics on a per page basis.

Truncation and invalidation
Truncation and invalidation are the main operations which use `find_get_pages` (in the form of `pagevec_lookup`). They are typically invoked on a range of pages in an inode,

[10] mapping basically represents an inode

[11] A page is locked by waiting for a 'lock' bit in its `flags` attribute to become clear, then setting it.

and pagevec_lookup is used to find these pages.

The truncate and invalidate operations themselves only operate on a single page at a time, so it is possible to use the lockless find_get_pages_nonatomic as their pagecache lookup function.

2.6 Lockless pagecache summary

This section described a design for lockless pagecache lookup operations in Linux, using a lockless RCU radix-tree for the pagecache data structure, and the page_cache_get_speculative operation to provide the required synchronisation without using a lock.

3 Performance results

In this section, the performance properties of the lockless pagecache will be analysed, and compared with the standard Linux 2.6 tree_lock based pagecache synchronisation.

3.1 Benchmarking methodology

The benchmarks presented here aim to give a fair representation of the basic performance and scalability behaviour of the lockless pagecache.

Benchmarks are run on several architectures where possible. It is important to show performance behaviour on a diverse range of hardware because low level details, especially memory coherency and consistency, atomic operations, can vary.

Benchmarks are run on uniprocessor and multiprocessor (UP, SMP, respectively) compiled kernels if relevant. UP compiled kernels can be optimised due to the fact that only a single processor will be running at once, so locking, atomic operations and memory consistency operations can differ significantly.

All benchmarks were run 10 times, and the error bars represent a 99.9% confidence interval.

3.1.1 Kernels tested

The base kernel tested was 2.6.16. The 'standard' kernel includes a number of preparatory patches [6] (which are now included in later kernels), because they might have an impact on performance. The 'lockless' kernel includes all preparatory patches, as well as the lockless pagecache patches [5].

3.2 find_get_page kernel level benchmarks

Benchmark machines
G5 - Apple G5 PowerMac. 2 CPUs (PPC970, 2.5GHz, 1MB L2). 4GB RAM.
P4 - Intel Pentium 4. 2 CPUs (Nocona Xeon, 3.4GHz, 1MB L2, HyperThreading). 4GB RAM.

find_get_page is a fundamental pagecache lookup function in Linux, which is made lockless with the lockless pagecache. The following tests were performed by timing loops which ran in kernel mode for the duration of the test (plus a single system call—fadvise—used to initiate the test). All find_get_page tests are performed on just a single file.

3.2.1 find_get_page single threaded benchmarks

Single threaded performance on SMP compiled kernels was tested from by looking up a sin-

gle page 1,000,000 times (Figure 8), and by looking up each page of a cached 1GB file in turn (Figure 9). In the former test, the working set should completely fit in the cache of all CPUs; in the latter case, each `struct page` being operated upon will not be in CPU cache. Uniprocessor (UP) kernels are also tested in single threaded benchmarks. Figures 6 and 7 show the results of the same two tests on UP kernels.

3.2.2 `find_get_page` multi threaded benchmarks

Multi threaded performance was tested by having two CPUs running `find_get_page` 1,000,000 times concurrently, first on the same page (Figure 11), then on different pages of the same file (Figure 10).

These microbenchmarks show that small system performance of various architectures and configurations has not suffered as a result of the lockless pagecache implementation; in fact, usually the opposite.

3.3 IO and reclaim benchmark

Page reclaim is an important operation for the kernel, as it is part of almost any workload that is filesystem IO intensive, and where the working set does not fit completely into RAM. Some examples may include desktop systems, web and file servers, compile/build servers, and some databases.

It is important to benchmark low level performance of page reclaim and IO together, because the lockless pagecache implementation changes both.

Figure 12 shows the results of reading 16GB per thread from a large file. The system only has 2GB of memory available for pagecache, so most of the pagecache must be reclaimed in the course of the test. In the single threaded case, kswapd, the asynchronous reclaim daemon, was restricted to the same CPU as the reading thread. The file is sparse, so reading from it is not limited by the speed of the system's block devices.

This benchmark together with the `find_get_page` one demonstrate that single threaded performance has not suffered, and even been improved, with the lockless pagecache.

Figure 12: IO and page reclaim, SMP kernel, two threads

3.4 Pagefault benchmark

Pagefaults of memory mapped files are one of the most basic of operations initiated from userspace, that require a pagecache lookup. The following benchmark involves a number of processes mapping 256MB chunks of the same file (which is resident in pagecache), and touching each page (causing a pagefault), then unmapping the chunk; this sequence is repeated 64 times. The total throughput (amount of pages faulted per second) is measured by the time taken for all threads to complete

This benchmark was run on a dual core AMD Opteron system, with 8GB RAM and 16 cores (8 sockets), Figure 13 illustrates the results.

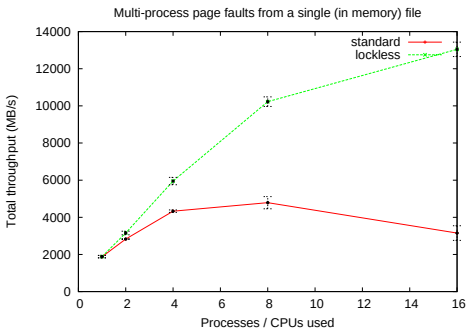

Figure 13: Pagefault scalability

The pagefault benchmark gives an idea of the potential scalability improvement provided by the lockless pagecache.

3.5 Data structure size

The lockless pagecache imposes a small impact on the size of the radix-tree node data structure as a result of using RCU for delayed deallocation. The impact is roughly a 5% increase in the size of the node. This is undesirable, however a radix-tree node itself takes much less than 1% of the memory it can store in pagecache pages, so the small size increase is not a major problem.

4 Conclusion

The lockless pagecache design has good potential. The design is not overly complex, and the implementation has so far proven to be robust. Initial benchmarks have shown that performance is improved in many areas, and the improvement in scalability of basic operations is significant. Further investigation of performance in 'real-world' benchmarks is warranted.

References

[1] Juergen Doelle. Re: [patch] align vm locks, new spinlock patch. [Viewed December 29, 2005], September 2001.

[2] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, pages 87–100, New Orleans, LA, February 1999. Preprint Available: `http://www.research.ibm.com/K42/osdi-preprint.ps` [Viewed Dec 29, 2005].

[3] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. 2004.

[4] Ingo Mollnar and David Miller. Scalable pagecache, February 2002. [Viewed December 29, 2005].

[5] Nick Piggin. Lockless pagecache patches for Linux 2.6.16. `http://www.kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.16/2.6.16-lockless.gz`.

[6] Nick Piggin. Preparatory patches for Linux 2.6.16. `http://www.kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.16/2.6.16-prep.gz`.

[7] Nick Piggin. Rcu radix-tree. Draft chapter available `http://www.kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.16-rc5/radix-intro.pdf`.

[8] Andrew Tridgell. dbench. `http://samba.org/ftp/tridge/dbench/`.

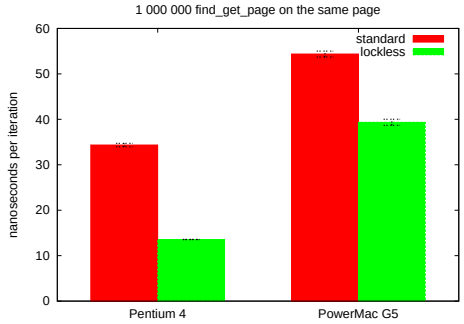

Figure 6: `find_get_page` UP kernel, cache hot

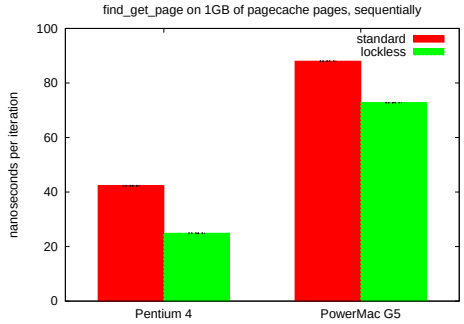

Figure 7: `find_get_page` UP kernel, cache cold

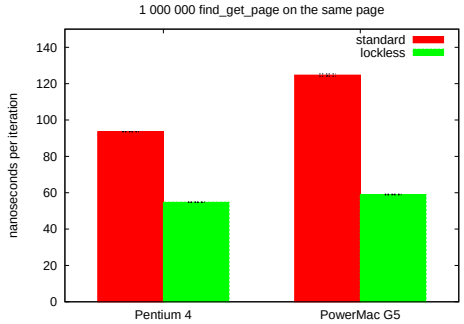

Figure 8: `find_get_page` SMP kernel, cache hot

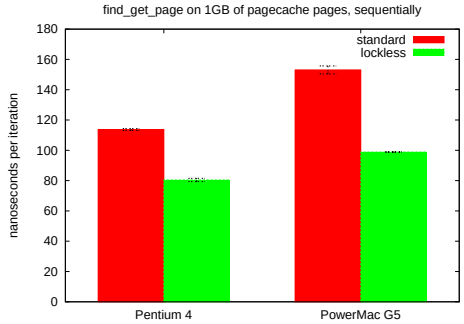

Figure 9: `find_get_page` SMP kernel, cache cold

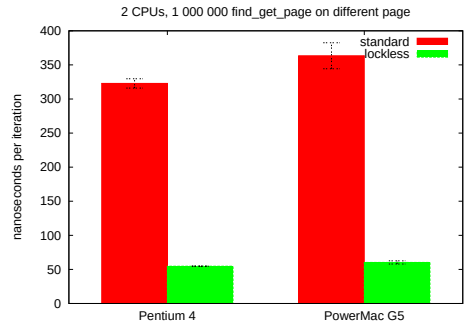

Figure 10: `find_get_page` SMP kernel, two threads, different pages

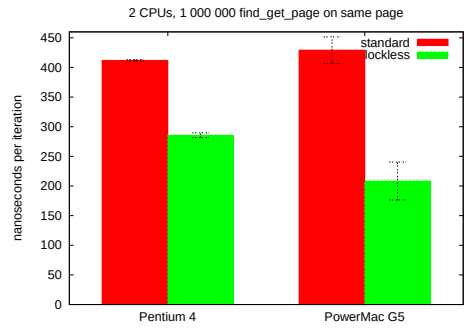

Figure 11: `find_get_page` SMP kernel, two threads, same page

Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin