

A lockless pagecache in Linux

Nick Piggin

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Software Engineering at
The Department of Computer Science
Australian National University

November 2006

© Nick Piggin

Except where otherwise indicated, this thesis is my own original work.

Nick Piggin
3 November 2006

Acknowledgements

I would like to like to thank a number of people and organisations, without them I could not have undertaken this research.

My girlfriend Lydia, parents, family, and friends for support. My supervisor Eric McCreath for advice and his time and effort.

SuSE Labs, Novell Inc., my employer, for supporting my research. In no particular order, AMD, IBM, Intel, NEC, and SGI, for help with hardware resources I used.

The Linux kernel community and free and open source software developers everywhere, for creating the great software they share with the world.

Abstract

One problem facing the Linux kernel when running on large systems is the multiprocessor scalability of the pagecache under various workloads. This thesis proposes a method for lockless synchronisation of the pagecache to improve scalability of critical operations. The approach taken to solve this problem was to determine the guarantees provided by the existing synchronisation scheme, then develop a lockless method which provides the same guarantees. An empirical study was conducted to compare the performance both approaches. The results demonstrate that some pagecache intensive workloads have throughput improved by orders of magnitude on large systems, while being as good or better on single CPU systems. This work also provides further avenues for improving pagecache scalability.

Contents

Acknowledgements	iii
Abstract	v
1 Introduction	1
1.1 Purpose	2
1.2 Contribution	2
1.3 Organisation	2
1.4 Definitions	3
2 Background	5
2.1 Pagecache history	5
2.1.1 Buffercache	5
2.1.2 Pagecache	6
2.2 Linux pagecache	7
2.2.1 Linux pagecache use-case	7
2.2.2 Linux pagecache history	7
2.3 Multiprocessor systems	9
2.3.1 Lock based synchronisation primitives	9
2.3.2 Cache coherency	10
2.3.3 Memory consistency	10
2.4 Motivation for a lockless pagecache	11
2.4.1 Memory wall	11
2.4.2 Critical section efficiency	12
2.4.3 Write-side scalability	15
2.5 Related work	15
2.5.1 Hashed locks	15
2.6 Read-Copy Update (RCU)	18
2.6.1 The problem	18
2.6.2 The solution	18
2.6.3 Implementation of RCU	19
2.6.4 Using RCU in Linux	19
2.7 Linux memory management details	21
2.7.1 Memory, struct page	21
2.7.2 Pagecache query operations	24
2.7.3 Pagecache synchronisation	26

3	Scalability of cache coherency	29
3.1	Introduction	29
3.2	Elements of workload involving memory sharing	30
3.2.1	Queueing model	30
3.2.2	Ahmdal's Law	33
3.2.3	Simulator	34
3.2.4	Comparison	34
3.3	Application of the model	35
3.3.1	Predictions	35
3.3.2	Real systems	36
4	Lockless radix-tree	39
4.1	Radix-trees	39
4.2	Lookups with concurrent modification	40
4.2.1	Slot modifications	41
4.2.2	Height modifications	42
4.3	Lockless radix-tree lookup semantics	46
4.4	Implementation details	47
4.4.1	Radix-tree tags	47
4.4.2	Gang lookups	47
4.4.3	Child count attribute	48
4.5	Summary	48
5	A lockless pagecache in Linux	49
5.1	Lockless pagecache outline	49
5.1.1	Permanence of struct page (existence guarantee)	49
5.1.2	Speculative pagecache references (accuracy guarantee)	50
5.1.3	Lookup synchronisation point (no new reference guarantee)	51
5.1.4	Providing guarantees in uniprocessor kernels	51
5.1.5	Problems	51
5.2	Lockless pagecache operations	54
5.2.1	find_get_page	54
5.2.2	find_lock_page	57
5.2.3	find_get_pages	57
6	Performance results	59
6.1	Benchmarking methodology	59
6.1.1	Kernels tested	59
6.1.2	Machines tested	59
6.2	Pagecache lookup scalability benchmark	60
6.3	find_get_page kernel level benchmarks	62
6.3.1	find_get_page single threaded benchmarks	62
6.3.2	find_get_page multi threaded benchmarks	64
6.4	IO and reclaim benchmark	64

7 Conclusion	65
7.1 Summary	65
7.2 Future work	66
A Engineering report	67
A.1 Project management	67
A.1.1 Constraints	67
A.1.2 Process activities	68
A.1.3 Process artifacts	70
A.2 Project lifecycle	71
A.2.1 Process model	71
A.2.2 Process activities	71
A.3 Configuration management	75
A.3.1 Revision control	75
A.4 Conclusion	75
Bibliography	77

List of Figures

2.1	Memory consistency example	11
2.2	Critical section efficiency	13
2.3	find_get_page pagecache lookup function in Linux	13
2.4	find_get_page critical section efficiency with one thread	14
2.5	find_get_page critical section efficiency with two threads	14
2.6	Linux struct page definition (simplified)	21
2.7	How struct page relates to physical memory pages	22
2.8	__free_pages function in Linux	23
2.9	Linux find_get_page function	24
2.10	Linux find_lock_page function	25
2.11	Linux find_get_pages function	26
3.1	Comparison of throughput for several methods of prediction	35
3.2	Predicted scalability with a constant service time	36
3.3	Predicted scalability with a logarithmic service time	37
4.1	Radix-tree of height 3	40
4.2	Population or clearing of a slot	41
4.3	Increasing the height of the radix-tree	43
4.4	Decreasing the height of the radix-tree	44
4.5	Node height invariant under increasing tree height	45
4.6	Node height invariant under decreasing tree height	46
5.1	Lockless find_get_page for SMP	55
5.2	Lockless find_lock_page	58
6.1	Pagefault scalability, standard kernel	61
6.2	Pagefault scalability, lockless vs standard kernel	61
6.3	find_get_page on UP kernel, cache hot	62
6.4	find_get_page on UP kernel, cache cold	62
6.5	find_get_page on SMP kernel, cache hot	63
6.6	find_get_page on SMP kernel, cache cold	63
6.7	find_get_page on SMP kernel, two threads, different pages	63
6.8	find_get_page on SMP kernel, two threads, same page	63
6.9	Page reclaim, SMP kernel	64

List of Tables

2.1	dbench scalability from 1 to 8 processors on Linux 2.4	8
4.1	Lockless radix-tree lookup versus non-leaf node insertion and deletion	42
4.2	Lockless radix-tree lookup versus leaf node insertion and deletion	42
4.3	Lockless radix-tree lookup versus tree height increase	43
4.4	Lockless radix-tree lookup versus tree height decrease	45
5.1	Steps involved in <code>find_get_page</code> and discarding a page	57

Introduction

Caching is an important part of the design of computing systems. This has been true from the very first electronic computers, and will continue to be for the foreseeable future. A cache stores a copy of frequently used data in a location where it can be accessed more quickly than from its original location.

Disk or file caches are very common in modern computer systems. They are usually implemented and managed by the operating system, which uses main memory to hold a copy of parts of the file or disk. As disks are very slow in comparison to RAM, virtually all disk operations in modern operating systems are performed via a cache.

Because disk or file caches are used so frequently, it is important for an implementation to achieve high performance in a wide range of usage patterns. In Linux, this includes the ability to perform well on multiprocessor systems.

Multiprocessor systems have existed for decades, but they have recently been gaining in importance as they are becoming cheaper and more common. The number of processing cores on a single chip is increasing as the gains in single core performance are diminishing. Today, chips with as many as 8 cores exist, and this number is likely to increase in future.

Unique performance considerations exist for multiprocessor systems. Each processor must prevent the intermediate steps of the algorithm being executed from being made visible to other processors, as that would cause problems. Access to data is usually protected by locks, which temporarily stop a processor until the lock is released, preventing intermediate data from being seen. Synchronising processors with locks can be a performance problem if a significant amount of time is spent waiting for the lock.

In Linux, a multiprocessor operating system, all processors must be allowed to access the file or disk cache. These accesses must be synchronised so the integrity of the cache is maintained. This is a bottleneck if a lock is used to synchronise the cache. This bottleneck is the topic of concern of this thesis.

1.1 Purpose

This thesis presents a new method for synchronising access to the Linux pagecache (a file cache) on multiprocessor systems without using locking in common operations. This method aims to provide increased performance, in particular on multiprocessor systems with large numbers of CPUs.

1.2 Contribution

The major contributions made in this work are a lockless method for radix-tree synchronisation, using Read-Copy Update (RCU); and a lockless method for Linux pagecache synchronisation on multiprocessor systems.

Secondarily, a queueing model was developed for the theoretical analysis of multiprocessor scalability in the presence of a contended resource, and an empirical study of the performance advantages of the lockless pagecache is presented.

1.3 Organisation

Chapter 2 provides the necessary background required to understand subsequent introduction of the work being presented, including: a history and description of the pagecache in Linux, and related work in other systems; an overview of multiprocessor systems and requirements of software that runs on them; an introduction to the Read-Copy Update (RCU) method used to implement the ‘lockless radix-tree’; and details specific to the development of the solution are introduced, these include Linux memory management details and pagecache synchronisation requirements.

In **Chapter 3**, a queueing model is developed to analyse multiprocessor scalability, and demonstrate the scalability problem inherent in any synchronisation scheme involving locks. Note that the reader may skip this chapter without missing any information required to understand subsequent chapters.

Chapter 4 presents the lockless radix-tree. This data structure is used by the lockless pagecache.

The lockless method for pagecache synchronisation is presented in **Chapter 5**.

An empirical study of the performance of the lockless pagecache is undertaken in **Chapter 6**.

Chapter 7 summarises the findings presented in this thesis, and suggests areas of future work.

1.4 Definitions

This section introduces some definitions used frequently throughout the thesis. Commonly used terminology, not found in this section, follows accepted definitions.

Linux The Linux kernel. Linux has had several pagecache data structures and synchronisation schemes in its history. Unless otherwise indicated, ‘Linux’ refers to Linux 2.6.17.

Inode is a data structure on a filesystem that stores information about a file or directory, and its contents. For this paper, it can be assumed that an `inode` has a one-to-one correspondence with a file. Linux has an in-memory data structure to represent an `inode`, which is used when an operation is to be performed on the file. Unless otherwise indicated, `inode` refers to this in-memory structure rather than the actual `inode` on disk.

Scalability The performance behaviour of a system in response to varying resources or problem size. This term is used to describe the performance of the *software* in response to varying the number of processors in a multiprocessor system.

Lock A lock is used to prevent one thread of execution from interfering with the intermediate results of another. If a lock has been taken and not yet released, an attempt by another thread to acquire the lock halts the thread at that point until the lock is released.

Concurrent operations are those which are executed at the same time on different processors.

Atomic operation which is seen as having executed completely, or not at all, from the point of view of concurrent processors.

Read-side When performing operations on a data structure, read-side operations are those which do not alter the data structure. Typically a lookup operation is a read-side operation.

Write-side / update-side Write-side and/or update-side operations are those which alter a data structure. Typically item insertion and removal are write-side operations.

Lockless The term lockless is used in this paper to indicate that a specific operation can be carried out without the need for any locks to control access to a critical section (ie. locking is not required for multiprocessor synchronisation).

Single threaded workloads are those where only a single thread of execution is running, or only a single thread is running in the code path of interest.

UP, SMP The Linux kernel offers UP (uniprocessor) and SMP (multiprocessor) compile options which changes some operations. For example, the UP kernel can optimise away spinlocks, because it is known that no other CPU will be holding them.

Background

2.1 Pagecache history

2.1.1 Buffercache

The design of the original UNIX operating system includes a disk caching layer that is today known as a *buffercache*. Dennis Ritchie and Ken Thompson [Ritchie and Thompson 1978] describe the high level architecture of the buffercache in *The UNIX Time-Sharing System*.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a read call the data are available; conversely, after a write the user's workspace may be reused. In fact, the system maintains a rather complicated buffering mechanism that reduces greatly the number of I/O operations required to access a file. Suppose a write call is made specifying transmission of a single byte. The system will search its buffers to see whether the affected disk block currently resides in main memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written. The return from the write call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

Maurice Bach [Bach 1986] examines the buffercache in further detail in *The design of the UNIX Operating System*. The high level function of the buffercache is a cache of block device (eg. disk) blocks. The buffercache is conceptually located between the block device interface and block device drivers. For example, if a filesystem requests the contents of a specific block from a block device, the buffercache will be queried for this block before the disk driver is instructed to read the data from disk.

The buffercache will not be examined further in this paper. Linux does have something equivalent to a buffercache for block device access, however this is implemented completely within Linux's pagecache infrastructure¹ and thus it requires no special consideration when examining pagecache synchronisation issues.

2.1.2 Pagecache

Many modern UNIX operating systems have the concept of a pagecache, which transparently caches contents of *files*, rather than the contents of block devices like the buffercache. The pagecache obsoleted the buffercache in UNIX operating systems when it was introduced in *SunOS 4* [Moran et al. 1987] (and later picked up by *SVR4 UNIX*). Linux and Solaris are examples of current UNIX operating systems which have a pagecache.

A pagecache uses physical memory pages to hold file data, which are stored and retrieved according to their corresponding *(inode, offset)* tuple, where *inode* represents a unique file in the system, and *offset* is the offset within that file. [Moran et al. 1987] explain, in *Virtual Memory Architecture in SunOS*, the role of their VM system (virtual memory system, or memory management system) as a cache manager, a *logically* addressed cache which is the pagecache as it is known today:

The VM system is a particularly effective cache manager, and maintains a high degree of sharing over multiple uses of a given page of an object. As such, it has subsumed the functions of older data structures, in particular the text table and disk block data buffer cache (the "buffer cache"). The VM system has replaced the old fixed-size buffer cache with a logical cache that uses all of the system's pageable physical memory.

The main advantage of a pagecache over a buffercache is that cache lookups do not need to go through the filesystem in order to access the cached data of a file². This is advantageous because it does not require the computational overhead of going through the filesystem, and it is able to cache the contents of filesystems that are not backed by a block device. One example of a filesystem without a backing block device is NFS (Network File System).

In Linux, the pagecache must satisfy a few basic performance characteristics. It must be fast to query, it must be small in its memory footprint, must scale well with many files, many pages, and it must scale well with concurrent access from multiple CPUs (multiprocessor scalability).

The focus of this paper is the improvement of the multiprocessor scalability of the pagecache in Linux.

¹Such an implementation is often referred to as a unified buffercache.

²The filesystem makes the translation from *(inode, offset)* to the *(device, block number)* required by the buffercache.

2.2 Linux pagecache

2.2.1 Linux pagecache use-case

A common use-case for the pagecache is a page-sized and aligned read(2) system call³, which copies the contents of a file into the user supplied buffer. In executing such an operation, the Linux kernel roughly performs the following operations:

1. System call entry into the VFS (the 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. Memory manager queries the pagecache for the page. If the page does not exist, go to 5; if the page not valid, go to 6; otherwise go to 10.
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 calls the VFS, to bring the page uptodate.
7. VFS calls the filesystem, which initiates a block device read of the page.
8. The process now blocks until completion of the read.
9. Upon completion of the read (typically from interrupt context), the page contents will be marked valid and the process woken.
10. Memory manager will copy the required data to the VFS for the read call.

2.2.2 Linux pagecache history

Following is a history of Linux pagecache architectures, including the present Linux 2.6.17 pagecache architecture. The descriptions focus on the synchronisation methods and multiprocessor scalability characteristics of the designs.

2.2.2.1 Linux 2.2: Single threaded kernel

Linux 2.2 has a global hash data structure to store pagecache pages. This data structure is protected from multiprocessor access with a lock shared by the entire kernel. This lock, called the 'Big Kernel Lock' (BKL), enforces the rule that only a single thread of execution may be active in the kernel at any given time.

This scheme is very simple because it generally requires no further thought about multiprocessor synchronisation issues (outside low level details like TLB flushing or context switching).

³If the read is larger than a page, or not page-aligned, it may require retrieval of 2 or more pagecache pages, however this is not much more complicated than the single page read.

The problem with a single threaded kernel is that it scales very poorly to multiple CPUs when running a workload that has some amount of kernel activity.

2.2.2.2 Linux 2.4: Single threaded pagecache

Linux 2.4 uses a fixed sized global hash-chain data structure in order to store pagecache pages. The pages are hashed according to (inode, offset) tuple. Pagecache pages are also present on per-inode lists of clean and dirty pages. Multiprocessor access to the hash table and lists 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. The lock must be taken on every read, write and truncate system call, and for page write-out and page reclaim. For a workload such as ‘dbench’⁴ (which benchmarks the filesystem component of a Samba ‘netbench’ file server benchmark execution), this lock causes diminishing returns in performance to set at 2 CPUs, and there is almost no performance improvement when moving from 4 to 8 CPUs.

Juergen Doelle [Doelle 2001] demonstrated the poor scalability of the global pagecache lock when running dbench. The results are shown in Table 2.1.

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

Table 2.1: dbench scalability from 1 to 8 processors on Linux 2.4

2.2.2.3 Linux 2.6: Per-inode threaded 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. There is one radix-tree per inode, and each tree is indexed by the page’s offset within the inode. The per-inode clean and dirty page lists were retained for some time, but have now been replaced with a hierarchy of tags (bits) in the radix tree to signal the existence of dirty pages.

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

⁴dbench is available at <http://samba.org/ftp/tridge/dbench/>

This spinlock was initially an exclusive lock, but was changed to a reader- writer spinlock in order to improve scalability. Looking up a pagecache page based on the (inode, offset) requires only a read-lock, thus multiple concurrent readers can query the pagecache of a particular inode.

The scalability of this design has proven to be acceptable for most workloads, because most workloads are spread over multiple files. However, when working with few files, contention increases because there are fewer locks. In the degenerate case, one file, this design is effectively equivalent to a globally locked pagecache.

2.3 Multiprocessor systems

A multiprocessor system is generally regarded as a system with multiple CPUs that all share coherent access to the same global memory. In Flynn's Taxonomy [Flynn 1972], this is classified as a multiple instruction, multiple data (MIMD) system, in which access to memory is coherent.

2.3.1 Lock based synchronisation primitives

When multiple CPUs are manipulating the same complex data structures in memory, some form of synchronisation is usually required, so that one CPU does not see partial results of another's computations, and that data is not corrupted. Locks are a common and simple way to solve this problem, and simply work by disallowing CPUs from entering *critical sections* while another is within one.

The following types of locks are used in Linux:

spinlock A spinlock prevents a CPU from entering critical sections by causing it to spin, continually polling the lock in a tight loop until it is released, at which time the CPU will atomically mark the lock as taken.

mutex A mutex prevents a CPU from entering critical sections by causing it to stop the currently executing task and putting it to sleep until it is woken when the lock is released. In the meantime, the scheduler is able to run other tasks on this CPU.

reader-writer Both spinlocks and mutexes can take the form of a reader-writer (also known as a shared-exclusive) lock. This type of lock can be taken for reading or taken for writing and provides the optimisation that multiple readers are allowed in the critical section concurrently. If the workload is very read heavy, which is commonly the case, allowing multiple readers in the critical section can be a good optimisation.

2.3.2 Cache coherency

It is important in our definition of multiprocessor systems to specify that memory access is *coherent*; this is the generally accepted definition today, and these are the only type of multiprocessor systems that Linux will run on. However in the past this has not always been the case, and it is still possible to find exotic systems where multiple processors can share memory which is not coherent.

The cache coherence problem arises when each CPU in the system has a private cache of memory. Such a system is said to be coherent only if a mechanism exists to ensure all copies of the memory remain consistent when that memory location is modified [Archibald and Baer 1986; Agarwal et al. 1998].

In practical terms, cache coherence ensures mutual exclusion algorithms are obeyed by all processors, prevents memory updates from being lost or overwritten by old values, and prevents processors from operating on old (previously overwritten) values.

All common methods used for maintaining cache coherency rely on causing writes to a memory location by one CPU to invalidate or update that memory location in all other caches containing that location. This is usually implemented in hardware, transparently to software. Importantly, these require communication to signal the other CPUs, which can be very slow.

2.3.3 Memory consistency

The memory consistency, or memory ordering, model of a multiprocessor system is the specification of how a sequence of memory operations generated by one processor interacts with those of other processors.

The traditional model for multiprocessor programming is sequential consistency as defined by [Lamport 1979]. Sequential consistency guarantees that the result of any parallel execution of n processors yields the same outcome as if the operations of all processors had been executed in some sequential order, and that the memory operations of a program appear in the same order that they are specified [Mosberger 1993].

Figure 2.1 is an example of two sequences of instructions. After being executed in parallel on two processors, this will never result in $A = 1$ and $B = 0$ if the processors are sequentially consistent. It may result in $A = 1$ and $B = 1$, or $A = 0$ and $B = 1$, or $A = 0$ and $B = 0$.

This form of consistency has found to restrict performance, hence modern CPUs use more relaxed (or weak) models, which may reorder loads and stores. In the above example, under a weaker memory ordering model, CPU1 may satisfy the load of x *before* the load of y is satisfied. This could result in $A = 1$ and $B = 0$. Such weakly ordered CPUs include explicit instructions (called ‘barriers’) to force sequential consistency at the point where the barrier is executed.

1: CPU0	CPU1
2:	
3: /* x, y start at 0 */	
4: x = 1;	A = y;
5: y = 1;	B = x;

Figure 2.1: Memory consistency example

It should be noted that relaxed consistency systems can be made to appear sequentially consistent by inserting barriers between all memory accesses⁵, so there is no loss of expressiveness in such a system. Inserting the correct barriers is simply an implementation detail.

By adding the required barriers to ‘lock’ and ‘unlock’ operations, a weakly ordered CPU can be made to appear sequentially consistent when using lock based critical sections for synchronisation and access to shared data. This requires no extra thought from the programmer, provided their locking is correct. However, weak memory consistency models are an issue when implementing lockless synchronisation algorithms. There are no implicit barriers in locks to provide sequential consistency, so any barriers required have to be issued explicitly.

Any machine independent algorithms in the Linux kernel (which includes the pagecache) must assume that the system has a weak memory consistency, and so must use the provided barrier functions if a specific memory ordering is required. Howells [Howells 2006] has compiled a comprehensive documentation of memory consistency issues with respect to Linux.

2.4 Motivation for a lockless pagecache

The motivation for this work is to improve the scalability of Linux’s pagecache lookup functions. There is an industry-wide push toward multi-core CPUs. Linux is being deployed on increasingly large multiprocessor systems⁶, and is running more diverse workloads. Trends in computer hardware will continue to make the reduction of locking overheads more important.

2.4.1 Memory wall

Wulf and McKee [Wulf and McKee 1995] coined the term in 1995; the memory wall comes about because the exponential growth in the speed of microprocessors far outstrips the increase of the speed of their connections to memory.

CPUs have continued to employ larger and more sophisticated cache hierarchies to alleviate the growing disparity between CPU and memory, and they have been very successful in avoiding

⁵Note that memory barriers are not free, and performing a barrier between each memory access would eliminate any performance advantages of a weakly ordered memory subsystem.

⁶SGI Altix systems can run with up to 1024 CPUs, with the hardware capability for more.

the memory wall. However caches are completely ineffective when sharing memory (locks and data) *between* CPUs in a multiprocessor system. The reason is that modified data cannot be cached on more than one CPU at a time.

The fundamental problem is that going ‘off chip’ takes a long time⁷, and sharing data between two CPUs necessitates moving data between the physical chips. McKenney [McKenney 2005] shows a modern PowerPC CPU is stalled for up to 1 000 instructions when accessing data that was in a remote CPU’s cache.

The result of this trend is that invalidating and reading cachelines as a result of locking continues to become more costly relative to the execution of regular instructions.

2.4.2 Critical section efficiency

Lock contention has traditionally been the major scalability problem in Linux. That is, locks would be held over a large number of instructions, and other CPUs wishing to enter a critical section protected by that same lock would have to wait a long time for it to be released. The typical solution to lock contention was to introduce more locks and make each one protect a smaller critical section. The result is that any particular lock is less likely to be held at a given time, and if it is held then it will be released sooner.

Increasingly fine-grained locking worked well while the cost of critical sections were much larger than the cost of taking and releasing the locks. However operating systems designers continued to introduce more locks, and the *memory wall* meant that hardware continued to become relatively slower at locking. At some point, the cost of locking becomes comparable to or even larger than the cost of the critical section itself. At this point, introducing more locks no longer improves scalability and can actually slow things down.

McKenney defines ‘critical section efficiency’ in order to quantify and examine this phenomenon, and has found that efficiency has generally been getting worse. Figure 2.2 defines and illustrates critical section efficiency.

2.4.2.1 Linux pagecache critical section efficiency

The primary Linux pagecache lookup method is the function `find_get_page` (which is a pagecache lookup function, introduced in 2.7.2). This function is shown in Figure 2.3 with its locking statements highlighted. Importantly, note that a lock is acquired, then page lookup operations are performed, then the lock is released. This function will be examined more thoroughly later.

Critical section efficiency for `find_get_page` can be estimated by comparing the time to execute this function with the time to execute the same sequence of operations excluding the

⁷In the duration of a single clock cycle on a 3GHz processor, light travels approximately 10cm.

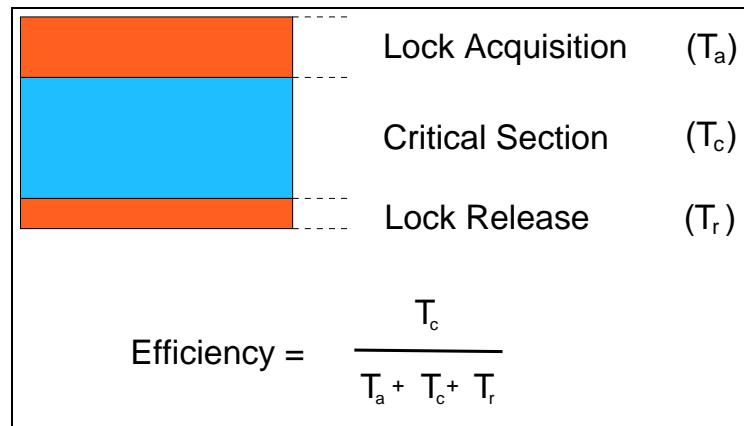


Figure 2.2: Critical section efficiency

```

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 2.3: find_get_page pagecache lookup function in Linux

locking statements. Note that it is not possible to measure the acquire and release costs separately, however that is not important for the purposes of simply finding the critical section efficiency.

Single threaded efficiency was measured by running each function in turn on a single CPU, 1 000 000 times for the same page of a file. The system tested was the P4 Xeon described in Chapter 6. Figure 2.4 shows the results.

A critical section efficiency of 0.36 indicates that only 36% of the time is spent performing the actual work of looking up the page and taking a reference on it, while the rest of the time is taken by locking and unlocking.

Now find_get_page is run concurrently on two CPUs on different pages of the same file in order to examine the effect on critical section efficiency. The results in Figure 2.5 show that efficiency is 0.11 – less than $\frac{1}{3}$ the efficiency of the single threaded case.

The increased cost of locking when running on two CPUs is due to contention for access to

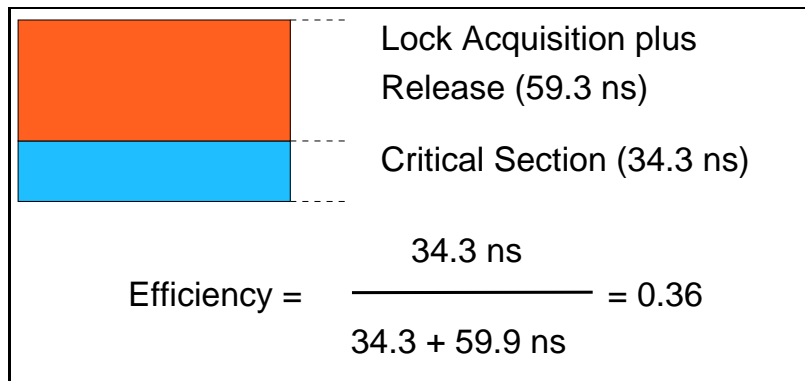


Figure 2.4: find_get_page critical section efficiency with one thread

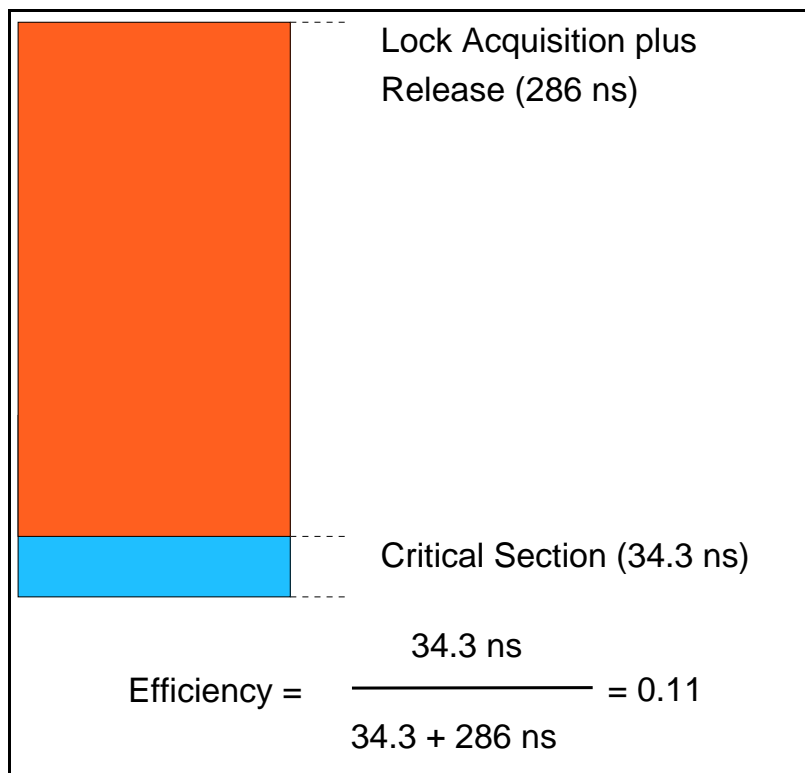


Figure 2.5: find_get_page critical section efficiency with two threads

the cacheline containing the lock. Although the lock is only being taken for read, each time it is taken or released the CPU must perform a write to memory. This write invalidates the cacheline corresponding to that memory location in any other CPU's cache. When the other CPU needs to read this memory value in order to perform a lock operation, it will take an expensive cache miss. As more CPUs contend the same cacheline, efficiency will continue to drop off, resulting in poor scalability.

2.4.3 Write-side scalability

This paper does not attempt to tackle the problem of improving Linux pagecache write-side scalability. The write-side includes operations such as adding and removing pages from the cache. These operations must still take the per-inode lock in the lockless pagecache method.

There are two reasons why this paper ignores the performance of write-side operations. Firstly, file caching is based on the principle that lookup operations on cached data (which involves read-side) will occur more frequently than inserting or removing data from the cache (which involves write-side). Secondly, improvement to the write-side is simply outside the scope of this paper. Possibilities for increasing write-side scalability are suggested as further work in the conclusion, Chapter 7.

2.5 Related work

In this section, an alternative synchronisation scheme for improving scalability of pagecache operations on a single inode is described and critically evaluated.

2.5.1 Hashed locks

Hashed locks are a common technique to improve scalability, although they come at a cost of cacheline footprint and will thus impact single threaded performance. Hashed pagecache locking typically requires a new lock (and thus a new cacheline) be taken for every page being read by a process.

Hashed locks are implemented in OpenSolaris, and the ‘Molnar/Miller scalable pagecache for Linux’.

2.5.1.1 Molnar/Miller scalable pagecache

Ingo Molnar and David Miller [Molnar and Miller 2002] 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.

In the Molnar/Miller pagecache, locks correspond with pagecache hash table buckets, and thus are randomly distributed according to the (inode, offset) tuple. This design uses a per-inode lock to protect per-inode lists of pagecache pages (for dirty/clean pages).

The 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. While the lookup path (arguably the most important, required for read and write syscalls) only needed to take the hash-bucket lock, other important operations (page-out, truncate, population of cache) also had to take both hash-bucket and inode locks, further impacting single-threaded performance. There is also complexity introduced in order to avoid lock ordering deadlocks.

The scalable pagecache was never adopted for the Linux kernel.

2.5.1.2 OpenSolaris hashed locks

OpenSolaris uses a global hash table to lookup pagecache pages. This hash table is traversed locklessly (in `page_lookup_create`) when looking up pagecache pages, although there are a number of situations in which a hash of locks protecting the hash table must be taken in order to synchronise access to the data structure (eg. when the page is not found during the lockless traversal). A hash of locks (`pse_mutex`), which is keyed by page address, is used to provide pagecache synchronisation, and is taken after a page is found with the lookup hash.

2.5.1.3 Implementation difficulties

Both the OpenSolaris and the Molnar/Miller pagecache synchronisation design systems use multiple layers of locks. By two layers of locks, what is meant is that some paths must take more than one path in order to access or modify the pagecache. For example, the Molnar/Miller pagecache must take one of the hashed locks in order to insert a page into the pagecache hash, and it must also take a per-inode lock in order to insert the page onto the per-inode page lists.

Introducing more layers of locking increases complexity, and is also a backward step in terms of single threaded performance. For these reasons, the hashed lock approach has been rejected by Linux developers and can not be seriously considered as a replacement for the current Linux pagecache design.

2.5.1.4 Hashed lock benefits

Lock hashing provides scalability improvements for locks that are likely to be *contended*: assuming a perfectly distributed hash, the chance of taking a lock already held by one CPU is reduced by a factor H (the number of entries in the hash) on a 2-CPU system⁸.

⁸The effect is not easy to determine on larger systems, because more than one lock may be held at one time. However, the magnitude of reduction in lock contention should be similar (to the factor H).

2.5.1.5 Hashed lock problems

One of the problems associated with hashed locks, in comparison to per-inode locks is that when a workload is distributed over multiple files, hashed locks will continue to be shared between processors, while per-inode locks will not.

Another issue is that lock hashing does not appear to reduce the amount of cacheline bouncing of the locks. In situations where contention for locks is low (which should be often the case in the Linux pagecache lookup routines, where the critical sections are small), lock hashing only offers an improvement in situations where the same lock is likely to be taken by one CPU several times in quick succession.

Take the scenario of a system which is accessing a locked objects from N CPUs, and not often accessing the same object or small set of objects multiple times in a row from any one CPU. Assume synchronisation can be provided by hashed locks, and that a good hash function will provide an essentially random pattern into the hash of locks.

Each of the cachelines in the hash of locks will be valid in at most the cache of one of the CPUs that has most recently taken or released the lock. Assuming equally busy CPUs and a random hash function, a proportion of $\frac{1}{N}$ of the total cachelines will be valid on each CPU.

Given the random hash function, there will be no correlation between the next lock hash access and the valid cachelines on any CPU, thus the chance of hitting a valid cacheline will be $\frac{1}{N}$, and the chance of an invalid cacheline is $\frac{N-1}{N}$. This represents the chance that the next lock operation will miss the cache.

Compared with a single lock, where the lock's cacheline will be valid on 1 CPU, thus the next CPU to access the lock will be $\frac{1}{N}$ chance of being the CPU where the cacheline is valid (assuming random lock access patterns among all CPUs). Thus the chance of an invalid cacheline is $\frac{N-1}{N}$. This shows that the hashed locks may not be any better than a single lock in cases where lock efficiency is low. In fact, it may be worse due to the extra cacheline footprint taken by the N locks and extra complexity in the hashing.

At extremely high levels of contention, the number of CPUs currently waiting for access to the cacheline will come into play. Suppose N CPUs attempt to take a read lock at the same time: one will be granted access to the cacheline and within time t , the second will then be granted access to the cacheline in $2t$, and the N th within Nt . By this time, the first CPU may be trying to acquire the lock again.

In the high contention scenario, hashed locks *will* alleviate cacheline contention. In the case of N CPUs attempting to take a random hashed lock, the average number of CPUs contending any lock (for H hashed locks) will be $\frac{N}{H}$, whereas the number is N for a single lock.

2.6 Read-Copy Update (RCU)

Read-Copy Update, also known as RCU, is a multiprocessor synchronisation technique invented by McKenney and Slingwine [McKenney and Slingwine 1998]. This section gives a background introduction to RCU and the problem it is designed to solve.

RCU is a framework and methodology for developing lockless algorithms and data structures. Before RCU, there were simply no good methods for implementing dynamic shared data structures without locking. McKenney and Slingwine [McKenney and Slingwine 1998] found primitive and ad-hoc implementations, but they had many problems and could not be used in a general purpose operating system (for example, one method was to never delete data elements, great for a short lived compiler application, but would eventually run out of memory if used in an operating system).

2.6.1 The problem

The major problem was the lack of a high performing and general way to provide *existence guarantees*. Existence guarantees ensure that part of the data structure is not deallocated while another thread of execution is examining or operating on it. Gamsa et al. [Gamsa et al. 1999] describe the difficulty of providing existence guarantees without locking:

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

2.6.2 The solution

The fundamental and novel idea behind RCU is a method of providing an existence guarantee without the need for locking. The existence guarantee is provided by delaying the deallocation of data, until after it is known that no other thread of execution can have a reference to the data. Determining when no references to the data exist relies on the *execution history* of each thread.

*This paper describes a novel approach in which updating CPUs and threads refer to a **summary of thread activity** in order to determine when update operations may be safely carried out. [McKenney and Slingwine 1998]*

Quiescent states and grace periods

RCU requires that each thread must periodically go through *quiescent* states. In a quiescent state, it must be guaranteed that the thread no longer has references to any data based on the re-

sults of previous computations. When all threads have gone through at least one quiescent state since an object was queued for delayed freeing, it is certain that no threads have a reference to the object.

A period in which all threads go through at least one quiescent state is known as a *grace period*⁹.

From the time an object is queued for delayed freeing, it may be deallocated after a quiescent period has subsequently passed. It now becomes possible to traverse the shared dynamic data structure and examine items without taking a lock.

Amenable data structures and algorithms

Those usage patterns most amenable to RCU are those which are:

- *read-intensive*, so the list of objects waiting to be deferred does not get too large;
- where stale data can be tolerated or suppressed;
- where there are frequently occurring quiescent states.

2.6.3 Implementation of RCU

McKenney and Slingwine [McKenney and Slingwine 1998] suggest quiescent states for various applications, and describe several implementations for finding quiescent periods. The important result is that quiescent states can be accounted for much less frequently than locking operations, and such accounting can be managed within operations that are already very expensive. Finding quiescent periods to deallocate waiting objects is more expensive, but it likewise is performed relatively infrequently in usage patterns that are suited for RCU.

In Linux, natural quiescent states are when a CPU is executing in user mode, performs a context switch, or becomes idle. While executing in kernel mode, with kernel preemption disabled, there is a direct mapping from thread to CPU so the implementations track quiescent states for CPUs rather than threads. Other quiescent states include context switches and CPU idle events.

2.6.4 Using RCU in Linux

In most cases, especially with a non-trivial data structure, RCU cannot be simply ‘dropped in’ to make a data structure suitable for a lockless read-side. The usual difficulty is to ensure that

⁹In the seminal RCU paper a grace period was called a quiescent period.

the update-side will transform the data structure from one valid state to the next, atomically, from the point of view of the read-side.

McKenney [McKenney 2006a] goes into great detail explaining the implementation and usage of RCU in Linux, but this quick introduction is sufficient to understand the methods presented in this thesis.

Read-side

In Linux, a reader that wishes to gain the existence guarantee provided by RCU can simply call `rcu_read_lock()`, and finish with `rcu_read_unlock()`. These are not actually locks in the traditional sense, they simply prevent preemption of this process (this does not involve atomic operations or cacheline contention from other CPUs). This prevents context switches in the RCU critical section, which would otherwise signal a quiescent period. This introduces the restriction that the process may not sleep in the critical section, as that would also result in a context switch.

Write-side

The write-side must still take care of the problem of multiple writers updating the data structure. Typically this is done with a lock. The writers must also structure their modifications so they result in atomic transformations from one valid state to the next, from the point of view of the reader.

Insertion of an object into a data structure typically involves initialising all the fields correctly first, then finally linking it into the data structure with a store to a pointer (which is atomic on architectures that Linux supports).

There is a slight complication with memory ordering in such an operation, and that is because some stores to the data structure may become visible *after* the store to the pointer on some architectures. Also, the read-side may be able to load parts of the data before the pointer becomes visible in rare situations. These issues are hidden behind two macros: `rcu_assign_pointer`, used by the write-side when performing a store to a pointer that makes the data structure visible; and `rcu_dereference`, to be used by the read-side when traversing pointers assigned in this manner.

When *deleting* an object, it will be atomically unlinked from the data structure by a store to a pointer. Then the object may be queued for delayed freeing after a grace period has passed. This can be achieved by calling `synchronize_rcu()`, which is a synchronous interface; or by registering a callback with `call_rcu()`, which is called after a grace period has passed.

The object cannot be deleted immediately after being unlinked from the data structure, because a concurrent reader may have loaded the pointer and has not yet finished examining the data. After a grace period, there will be no readers with a reference to the unlinked object.

The read-side must be able to cope with stale data being returned. It may encounter data items that have been deleted and queued for freeing. Handling this correctly is specific to the data structure and its usage.

2.7 Linux memory management details

An introduction to the relevant details of the Linux memory management implementation needs to be given in order to provide the reader with enough background to understand the proposal for a lockless pagecache. Unnecessary details are slightly simplified in places, so as not to distract from the main concepts being introduced. For further reading, Mel Gorman [Gorman 2004] provides a thorough examination of memory management in Linux.

2.7.1 Memory, `struct page`

In Linux, every physical page frame (RAM) that is used by the kernel is represented with a corresponding `struct page` structure. This structure contains the fields: `mapping` and `index` correspond to the pagecache (`inode`, `offset`) if the page is allocated for pagecache; `flags` are general flags bits; `_count` is a reference count; and various other data associated with the status and management of the page frame. The `struct page` definition is given in Figure 2.6, it is slightly simplified, and comments are changed to just describe the fields relevant to this work.

```

1: /*
2:  * Each physical page in the system has a struct page
3:  * associated with it to keep track of whatever it is
4:  * we are using the page for at the moment.
5:  */
6: struct page {
7:     unsigned long flags;
8:     atomic_t _count;           /* usage count */
9:     atomic_t _mapcount;
10:    unsigned long private;
11:    struct address_space *mapping; /* pagecache inode's mapping */
12:    pgoff_t index;             /* offset within the inode */
13:    struct list_head lru;
14:    void *virtual;
15: };

```

Figure 2.6: Linux `struct page` definition (simplified)

The `struct page` is the usual way to refer to a page, and the pagecache is no exception: it is a pointer to the `struct page` representing a page that is stored in the pagecache radix-tree.

Figure 2.7 gives an idea of how the `struct page` relates to page frames¹⁰.

¹⁰Two separate columns is slightly inaccurate because the `mem_map` array of `struct page` is itself stored in

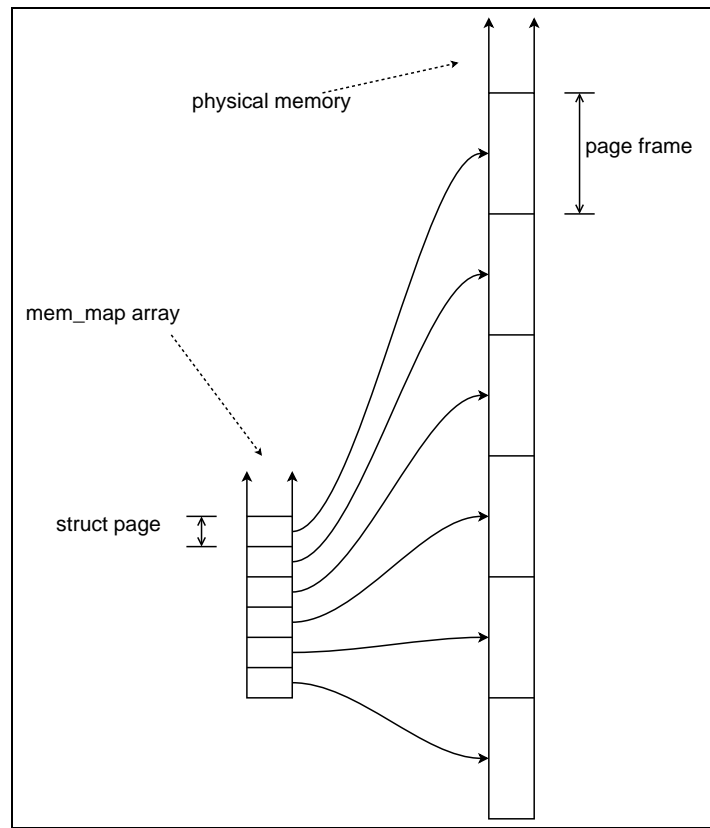


Figure 2.7: How `struct page` relates to physical memory pages

2.7.1.1 Page lifetimes, reference counting

All pages have a reference count (`_count`), in their `struct page`. This reference count is 0 when the pages are free. When the page is allocated, `_count` is set to 1 before it is made available.

When a part of the kernel that has a reference to the page is done with it, `_free_pages` (shown in Figure 2.8) or a similar function is called. This does not directly free the page, but atomically decrements the refcount and tests whether that has caused it to become zero. If it has become zero, the page will then be freed and returned to the page allocator, otherwise no further action will happen – the non-zero refcount indicates that there is another user of this page, so it is still in use. There is a `get_page` function, which increases the refcount of an allocated page if another reference to the page is made.

physical memory frames, and it may not be implemented as a single contiguous array, however these details are inconsequential here.

```
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 2.8: `__free_pages` function in Linux

2.7.1.2 Dirty pages

A pagecache page is considered dirty if its contents are more recent than the contents of the filesystem it is caching. A pagecache page becomes dirty if a program invokes the write system call to modify the data in a file. If a pagecache page is not dirty then it is *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 is cleaned by writing the contents of the page to appropriate location of the underlying block device.

2.7.1.3 Page reclaim, migration

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 are unused (have no references to them, except the pagecache reference) before being reclaimed. If the page has a reference taken on it from somewhere, that user may subsequently dirty the page even if it is now clean, and that data would be lost.

Page migration is a feature in Linux where pagecache data is moved from one page to another¹¹. This operation involves discarding the page currently in pagecache, then copying its contents to another page, then inserting that page into the pagecache.

¹¹Page migration is primarily used to optimise placement of memory on Non-Uniform Memory Architecture (NUMA) systems.

2.7.2 Pagecache query operations

This subsection introduces the pagecache query (lookup) operations relevant to the lockless pagecache. They are `find_get_page`, `find_lock_page`, and `find_get_pages`.

2.7.2.1 `find_get_page`

The `find_get_page` operation performs a lookup of a pagecache page at the given (`inode`, `offset`) tuple (the `mapping` parameter corresponds to the `inode`). If no page exists, `NULL` is returned, otherwise the page has its reference count elevated, and is returned to the caller.

The current implementation of `find_get_page` is given in Figure 2.9. `find_get_page` has specific semantics for what it can return. If the pagecache location given by (`mapping`, `offset`)

- *always* contained a particular page, `find_get_page` *must* return it;
- was *always* empty, `find_get_page` *must* return `NULL`;
- *ever* contained a page, `find_get_page` *may* return it;
- was *ever* empty, `find_get_page` *may* return `NULL`.

If a page is selected to be returned in accordance with the above semantics, `find_get_page` is required to ensure that the page's refcount is elevated *while it is in the pagecache*.

```

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 2.9: Linux `find_get_page` function

2.7.2.2 `find_lock_page`

`find_lock_page` (in Figure 2.10) is similar to `find_get_page`, however it also requires that the returned page is locked¹², and pinned at the given location in pagecache.

¹²A page is locked by waiting for a 'lock' bit in its `flags` attribute to become clear, then setting it.

Locking a page pins it in the pagecache (by excluding concurrent processors from removing the page), unlike holding a reference to it (which only prevents the page from being deallocated after being removed from pagecache). So the usual pattern for getting a locked page at a particular pagecache location is to first get a reference to the page to provide an existence guarantee¹³, then taking the page lock, then verifying that the page has not moved. If it has, the operation is retried.

Note that `find_lock_page` relies on `tree_lock` to hold the page in pagecache while attempting to take the page lock as an optimisation, on line 11. If this fails, it falls back to waiting for the lock and rechecking that the page is in the pagecache.

```

1: struct page *find_lock_page(struct address_space *mapping,
2:                             unsigned long offset)
3: {
4:     struct page *page;
5:
6:     read_lock_irq(&mapping->tree_lock);
7: repeat:
8:     page = radix_tree_lookup(&mapping->page_tree, offset);
9:     if (page) {
10:        page_cache_get(page);
11:        if (TestSetPageLocked(page)) {
12:            read_unlock_irq(&mapping->tree_lock);
13:            __lock_page(page);
14:            read_lock_irq(&mapping->tree_lock);
15:
16:            /* Has the page been truncated? */
17:            if (unlikely(page->mapping != mapping ||
18:                        page->index != offset)) {
19:                unlock_page(page);
20:                page_cache_release(page);
21:                goto repeat;
22:            }
23:        }
24:    }
25:    read_unlock_irq(&mapping->tree_lock);
26:    return page;
27: }
28:

```

Figure 2.10: Linux `find_lock_page` function

2.7.2.3 `find_get_pages`

`find_get_pages` is like `find_lock_page` but it operates on a range of pagecache, conceptually performing a `find_get_page` on each page that exists in the range. `find_get_pages` is shown in Figure 2.11. The `radix_tree_gang_lookup` function returns a range of pages in an array.

¹³Section 2.6 describes existence guarantees.

`find_get_pages` is not *exactly* the same as multiple calls to `find_get_page`, because it holds `tree_lock` for the duration of the calls, all the pages returned existed in the pagecache at the same time, at some point. Multiple calls to `find_get_page` would only ensure that each page existed in the pagecache at some point. However this functionality is not used by any caller, since `find_get_pages` was introduced as an optimisation for multiple calls to `find_get_page`.

```

1: unsigned int find_get_pages(struct address_space *mapping,
2:                             pgoff_t start, unsigned int nr_pages,
3:                             struct page **pages)
4: {
5:     unsigned int i;
6:     unsigned int ret;
7:
8:     read_lock_irq(&mapping->tree_lock);
9:     ret = radix_tree_gang_lookup(&mapping->page_tree,
10:                                (void **)pages, start, nr_pages);
11:     for (i = 0; i < ret; i++)
12:         page_cache_get(pages[i]);
13:     read_unlock_irq(&mapping->tree_lock);
14:
15:     return ret;
16: }
17:

```

Figure 2.11: Linux `find_get_pages` function

2.7.3 Pagecache synchronisation

Pagecache synchronisation in Linux requires controlling access and modification to both the pagecache data structure, and the pagecache pages themselves. The actual synchronisation requirements will be stated now.

2.7.3.1 Data structure synchronisation

One protection provided by `tree_lock` is the protection of the pagecache data structure (the radix-tree). In the current Linux synchronisation design:

- Pagecache read-side operations (which do not modify the data structure) hold the corresponding inode's `tree_lock` for read, and so exclude concurrent modifications to the radix-tree associated with that inode.
- Pagecache write-side operations (which do modify the data structure) hold the `tree_lock` for write, and so also exclude concurrent readers.

To perform pagecache lookup operations without holding `tree_lock`, a data structure able to cope with lockless readers is required. 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 perinode radix-tree structure in Linux 2.6¹⁴. 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* improvements.

A lockless radix-tree was developed to be used as the data structure for the lockless pagecache method. The lockless radix-tree is presented in Chapter 4.

2.7.3.2 Page synchronisation

With the ability to retrieve pagecache pages from the data structure without taking a lock, the problem of synchronising access to pagecache pages themselves still exists. Access to pagecache pages is presently synchronised with the `tree_lock` of the inode to which the page belongs.

When `tree_lock` is held for read, it provides the following guarantees (by providing exclusion from writers):

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

When held for write, `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.

2.7.3.3 Existence guarantee

An existence guarantee is 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. After the `refcount` is incremented, the elevated reference provides an existence guarantee for the caller that receives a pointer to the `struct page`.

¹⁴A fixed-size, changed hash has $O(N)$ worst case computational complexity for lookup vs $O(\log(n))$ for a radix-tree, and a hash table is much more difficult to size correctly for all workloads.

This guarantee is provided by holding the `tree_lock` for read. The radix-tree itself holds a reference to the page, and by excluding writers it is guaranteed that the page will not be removed from the radix-tree. If the page remains in the radix-tree, then its `refcount` will be *at least* 1, so it will not be freed.

The concept of an existence guarantee can be difficult to understand at first; with traditional lock based synchronisation, existence is almost always provided by having concurrent modifications excluded, so little thought needs to be given to it. Existence can be better understood by examining the consequences of its absence.

`find_get_page`, shown in Figure 2.9, elevates the reference count of the `struct page` using `page_cache_get` on line 9. This prevents the page from being freed before that reference is dropped. However if the `tree_lock` were not held during this operation, then after executing line 7 but before executing line 9, another CPU might remove the page from the pagecache, causing it to be freed. Then, the execution of line 9 would increment the reference count of a `struct page` which has been freed and possibly even allocated for some other use.

2.7.3.4 Accuracy guarantee

After looking up a page in the pagecache radix-tree, `find_get_page` requires that it remain in the pagecache until after the `refcount` has been incremented. In other words, the page must be actually in the pagecache when this operation is performed. `find_lock_page` also requires that the page is locked while it is in the pagecache.

The accuracy guarantee is different from the existence guarantee. The existence guarantee only provides that the `struct page` exists and is valid memory.

`find_get_page` currently holds the `tree_lock` over the entire operation, which prevents the page from being removed from pagecache, and thus provides the accuracy guarantee.

2.7.3.5 No new reference guarantee

The no new reference guarantee ensures that no pagecache lookup routines will take a new reference to a particular page. This guarantee is required in order to discard clean, unused pages for reclaim and migration (see 2.7.1.3). To discard a clean, unused page, the memory manager needs to ensure nobody can take a new reference to the page before it is removed from pagecache.

The no new reference guarantee is presently enforced by holding `tree_lock` for write. `find_get_page` obeys because it will not be granted the `tree_lock` for read until it is no longer locked for write.

Scalability of cache coherency

This chapter contains a theoretical analysis of the scalability impact of resource contention in a workload. The reader may skip this chapter without missing concepts used in the description of the lockless radix-tree and pagecache.

3.1 Introduction

Modern processors contain *cache memory*, which provides a copy of system RAM that can be accessed very quickly. Cache is managed as a set of fixed sized lines (cachelines) of contiguous memory. The size of each cacheline is larger than the word size of the machine for efficiency reasons (64 bytes is a typical cacheline size).

Cache memory poses a coherency problem for multiprocessor systems. Because they work on cached copies of memory, they must be prevented from working on stale data that has already been modified by another processor. *Cache coherency* protocols are used to solve this problem. These protocols determine whether a processor is allowed to access memory at a particular time, and how processors can request access to a particular memory location. Cache coherency protocols manage memory in cacheline sized chunks as well.

A particular memory location may exist in a read-only (shared) state in the caches of multiple processors at once, or it may exist in a read-write (exclusive) state in the cache of at most one processor. If a processor needs write access to this memory, they must cause any other copies of this location to be invalidated. If a processor needs read access to the memory, it must cause the read-write cached copy to be invalidated, if one exists.

Invalidating cachelines is slow because it requires off-chip communication. When one or more processors frequently writes to a cacheline shared by others (read, written, or both), the cacheline must be moved between caches frequently. This is known as cacheline bouncing or cacheline ping-pong, and is one of the biggest scalability problems for multiprocessor software.

3.2 Elements of workload involving memory sharing

Assume that a particular cacheline is a single shared resource (which it is). When this cacheline is being accessed in a read-write manner, the cacheline must be requested in the exclusive state and only one processor may access the resource at a time. Therefore when multiple processors attempt to access the cacheline, they must enter some kind of queue to wait for the resource. When a processor reaches the head of this queue, it is serviced, then leaves the queue.

We consider a single cacheline, so there is a single queue. It requires a constant time, S , to service a processor once it reaches the head of the queue.

The processor will access this cacheline at an ideal fixed frequency ν , if the service time to access this cacheline was instantaneous. Hence, the processor is performing other useful work for time ν^{-1} *between* accesses to the cacheline. This can be thought of as one unit of work.

There is a ‘population’ of N processors accessing the cacheline in this same pattern. When $N > 1$, there may be contention for the cacheline, causing each processor to perform work at less than the ideal rate – additional time is being spent waiting in the queue for access to the cacheline.

Thus, the average time that each processor spends waiting in the queue will give an indication of the performance characteristics of this model. *Queueing theory* provides a way of estimating these characteristics.

3.2.1 Queueing model

3.2.1.1 Little’s Theorem

Little’s theorem [Little 1961] states

$$N_q = \lambda T \tag{3.1}$$

where N_q denotes the number of entities in the system (queued or being serviced), λ is the arrival rate of entities into the system, and T is the average time that each entity is in the system.

Little’s Theorem was developed to model telephone exchange queueing, and most queueing theory deals with situations where the number of entities queued are insignificant compared to the total population of potential entities. In such a situation, the queued entities don’t have a significant impact on the population that is not queued, and so the insignificant influence to the arrival rate is ignored.

However, in the shared cacheline model, a large proportion of the total population of processors can be waiting in the queue. This significantly reduces the number of processors not caught

in the queue, which results in a significant reduction in the arrival rate. This should not be ignored.

3.2.1.2 Model for a shared cacheline

To apply Little's Theorem to the problem of a shared cacheline, the arrival rate must take into account the number of processors waiting in the queue. Assuming the service time was infinitely fast, the frequency of arrivals would be the ideal frequency for single processor, multiplied by the total number of processors. But if some processors are waiting in the queue, they won't contribute to arrivals. So let the frequency of arrivals be the frequency of arrival of a single processor multiplied by the number of processors which are *not* queued in the system.

$$\lambda = v(N - N_q) \quad (3.2)$$

The average time that each processor is in the system (queued or being serviced) is the time required to service a single processor multiplied by the number of processors in the queue (which represents the average time waiting in the queue), plus the time to service a single processor (the time to service this processor).

$$T = S(N_q + 1) \quad (3.3)$$

Then λ and T can be substituted into Little's Theorem

$$\begin{aligned} N &= v(N - N_q)S(N_q + 1) \\ &= vS(NN_q - N_q^2 + N - N_q) \end{aligned} \quad (3.4)$$

Simplifying further

$$\begin{aligned} \frac{N_q}{vS} - NN_q - N_q + N_q^2 + N &= 0 \\ N_q^2 + N_q\left(\frac{1}{vS} - N + 1\right) + N &= 0 \end{aligned} \quad (3.5)$$

Solving for N_q yields two equations:

$$\begin{aligned}
 N_q &= \frac{-\left(\frac{1}{vS} - N + 1\right) \pm \sqrt{\left(\frac{1}{vS} - N + 1\right)^2 + 4N}}{2} \\
 &= \frac{\left(N - \frac{1}{vS} - 1\right) \pm \sqrt{\left(N - \frac{1}{vS} - 1\right)^2 + 4N}}{2}
 \end{aligned} \tag{3.6}$$

$N \geq 1$ is given, because there must be at least one processor in the system. Then it can be shown that when $N > 0$, one of the equations is always negative and so can be discarded (a negative number of entities queued is nonsense):

$$\begin{aligned}
 \left(N - \frac{1}{vS} - 1\right) &= \sqrt{\left(N - \frac{1}{vS} - 1\right)^2} \\
 \left(N - \frac{1}{vS} - 1\right) &< \sqrt{\left(N - \frac{1}{vS} - 1\right)^2 + 4N}
 \end{aligned} \tag{3.7}$$

so

$$\begin{aligned}
 N_q &= \frac{\left(N - \frac{1}{vS} - 1\right) - \sqrt{\left(N - \frac{1}{vS} - 1\right)^2 + 4N}}{2} \\
 2N_q &= \left(N - \frac{1}{vS} - 1\right) - \sqrt{\left(N - \frac{1}{vS} - 1\right)^2 + 4N} \\
 2N_q &< 0 \\
 N_q &< 0
 \end{aligned} \tag{3.8}$$

So there is a unique solution for N_q . Then T can be re-written with N_q substituted away.

$$T = \left(\frac{\left(N - \frac{1}{vS} - 1\right) + \sqrt{\left(N - \frac{1}{vS} - 1\right)^2 + 4N}}{2} + 1 \right) * S \tag{3.9}$$

The time taken for a processor to perform one unit of work is $T + v^{-1}$, the time spent in the system, plus the time spent performing useful work. Then throughput (work / time) for a single processor is $\frac{1}{T+v^{-1}}$. So total throughput for all N processors is $\frac{N}{T+v^{-1}}$

Expanding T , and taking the limit as the number of processors approaches infinity gives the

following limit.

$$\lim_{N \rightarrow \infty} \frac{N}{T + v^{-1}} = \frac{1}{S} \quad (3.10)$$

This equation shows that the total throughput in a multiprocessor system has a fundamental upper bound as the number of processors increases. This is an important result because it shows that performance cannot be increased to an arbitrary level by increasing the number of processors in a workload with a shared exclusive cacheline.

For comparison, the ideal total throughput (ie. assuming service time is instantaneous) is $\frac{N}{v^{-1}}$.

3.2.2 Ahmdal's Law

The above limit is the same as the limit predicted by *Amdahl's Law* when applied to an analogous problem. This provides some degree of confidence in the derivation of the model.

Amdahl's Law (when applied to parallel programming) states that, for a particular problem, the work can be broken into a serial plus a parallel portion. In one work unit, the serial portion is a proportion of the total work $0 \leq K \leq 1$, and the parallel portion is the remaining $1 - K$. Then the work unit can be completed in a smaller time by increasing the number of processors, N :

$$speedup\ factor = \frac{1}{K + \frac{1-K}{N}} \quad (3.11)$$

Then the maximum possible speedup factor is:

$$\lim_{N \rightarrow \infty} \frac{1}{K + \frac{1-K}{N}} = \frac{1}{K} \quad (3.12)$$

To apply Amdahl's Law to the cacheline contention problem, suppose that each processor performs one work unit. All processors perform the parallel part of the work; then all perform the cacheline transfer, one after the other. Thus the workload can be split into a strictly parallel portion, and a strictly serial portion. Then the serial portion of work is S , and the parallel portion is v^{-1} . Thus,

$$K = \frac{S}{S + v^{-1}} \quad (3.13)$$

Then predicted throughput is the ideal throughput of a single processor, multiplied by the

speedup factor.

$$\frac{1}{S + v^{-1}} * \frac{1}{K + \frac{1-K}{N}} \quad (3.14)$$

Then the limit as N approaches infinity is the same as the limit found for the queueing model!

$$\frac{1}{S + v^{-1}} * \frac{S}{S + v^{-1}} = \frac{1}{S} \quad (3.15)$$

This does not give quite the same results as the queue model (as shown below). The reason is that the queue model attempts to describe the situation where a number of processors are performing the parallel part of the work and a number are queued for access to the cacheline. That is, access to the cacheline can be performed in *parallel* with other processors performing useful work, something that this Amdahl's Law model does not allow.

However the main result is that the limit is the same, which is intuitive because as the number of processors approaches infinity, the time spent performing the parallel part of the problem becomes insignificant in either model.

3.2.3 Simulator

One downside of the queueing model is that it is modelling a discrete system with a probabilistic equation, so it will not be entirely accurate.

A discrete system can be modelled using a simple simulator, in which a number of processors are represented in various states (queued, servicing, running). Such a model is not entirely accurate either, because there are complex perturbations in a real workload running on a real system. The behaviour predicted by the simulator shows a very sharp point where scaling improvements stop. This is because all processors become perfectly distributed in terms of their relative position in the workload 'pipeline'.

In reality, behaviour is likely to be somewhere between that predicted by the queueing model, and that predicted by the simulator.

3.2.4 Comparison

Given the same input parameters, arrival frequency of 100 000, service latency of 500ns, Figure 3.1 compares the performance behaviour predicted by each of the 3 methods.

Importantly, all 3 methods appear to be converging to a similar upper bound, which gives confidence in the predicted limit.

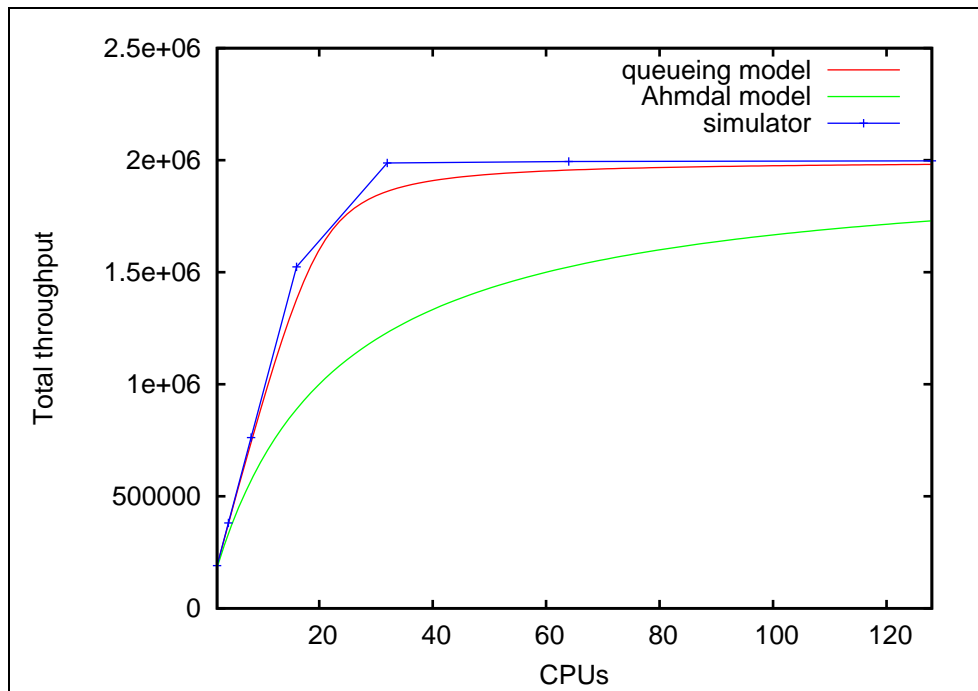


Figure 3.1: Comparison of throughput for several methods of prediction

Also, the comparison shows that the queuing model agrees reasonably closely with the results obtained from simulation. The Ahmdal model is further away, which is to be expected because that model is not entirely accurate, as noted above.

3.3 Application of the model

3.3.1 Predictions

Using data gathered from the Pentium 4 described in Chapter 6, the average time for a processor to take, and release the lock is roughly 200ns. Using this for the service time, and with a workload performing 1 310 072 pagecache lookups per second (512MB/s with 4K sized pages), the queuing model yields the performance prediction shown in Figure 3.2.

This prediction shows performance increasing quite well up to 32 processors, but then dropping dramatically after that, and the increase almost stops by 64 processors.

And actually, the situation is worse than this. The service time estimate taken from the 2 processor Pentium 4 is unrealistic for a larger system, simply because of the physical constraints. The larger a system is, in general, the longer it will take to transfer a cacheline from one processor to another. SGI Altix multiprocessor systems have a tree topology [Woodacre et al. 2003], which suggests that latency increases with some logarithmic factor of the number of

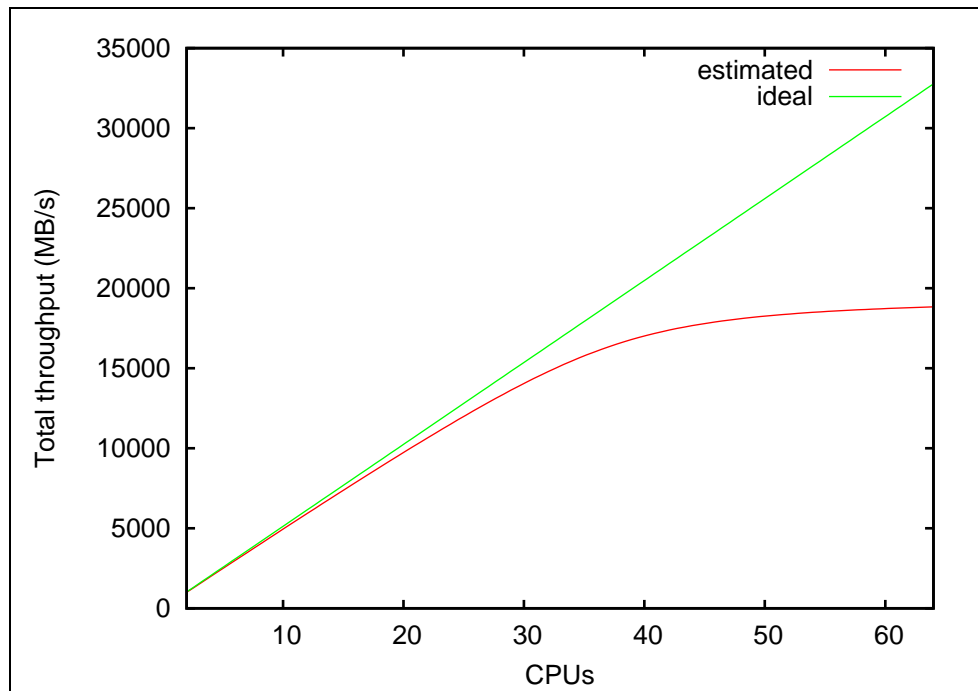


Figure 3.2: Predicted scalability with a constant service time

processors. AMD Opteron systems follow a similar logarithmic factor. Introducing such a factor into the equation yields Figure 3.3. This shows that performance can actually *decrease* as more processors are added.

3.3.2 Real systems

The application of this model is difficult to apply to real systems, because it is very hard to know what the average time will be to transfer a cacheline. Real systems have complex interconnects with many operations occurring on the interconnect, and complex cache coherency protocols. All these things influence the time required for a cacheline to be transferred.

The important point conveyed by this model is that scalability is fundamentally limited when there is any type of cacheline contention in a multiprocessor system. Not only is scalability limited, but total performance has an upper bound regardless of the number of processors in the system. This means that in a workload involves shared cachelines, then a given level of performance may not be achievable simply by adding more processors to the system.

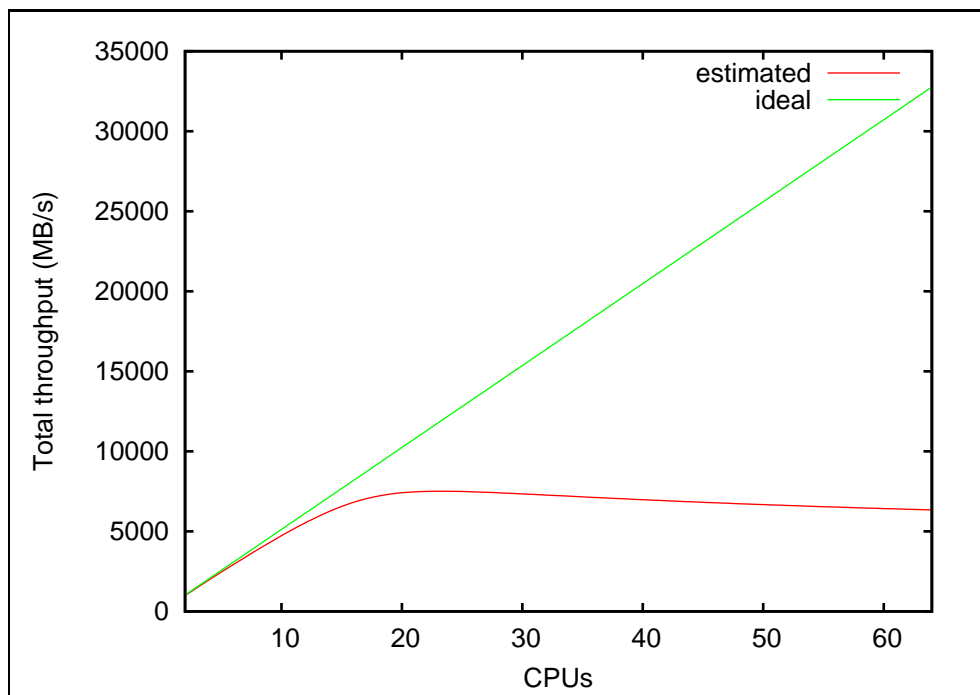


Figure 3.3: Predicted scalability with a logarithmic service time

Lockless radix-tree

In short, RCU seems to be a case of “hey, that’s cool”, but it’s a solution in search of a problem so severe that it is worth it. – Linus Torvalds¹

One of the requirements for a lockless pagecache is a lockless data structure (identified in Section 2.7.3.1). A lockless radix-tree was developed to satisfy this requirement, and is introduced in this chapter.

The radix-tree is made lockless by having modifications appear atomic with respect to the read-side with careful ordering of operations, and by providing existence guarantees to the read-side using RCU.

4.1 Radix-trees

For ease of explanation, a simpler radix-tree structure will be described than exists in Linux; in particular, ‘tags’ and ‘gang lookups’ will be ignored for this and the following section. The lockless radix-tree concepts will then be reconciled with the complexities of the Linux radix-tree in Section 4.4.

A radix-tree is a tree of nodes, where the leaf nodes are the actual data items. Each node has a fixed number (must be a power of 2) of slots. These slots are pointers to child nodes (or NULL if no child node exists). Also there is a pointer to the root node which is a ‘handle’ to the radix-tree.

The height of a radix-tree is defined as the number of traversals from the root node required to reach a leaf node.

¹Linux Torvalds is the creator of Linux. This quote suggests that he didn’t see value in RCU for Linux at the time. However it was included in the kernel a year later, and is now used extensively (<http://www.rdrop.com/users/paulmck/rclock/linuxusage.html>).

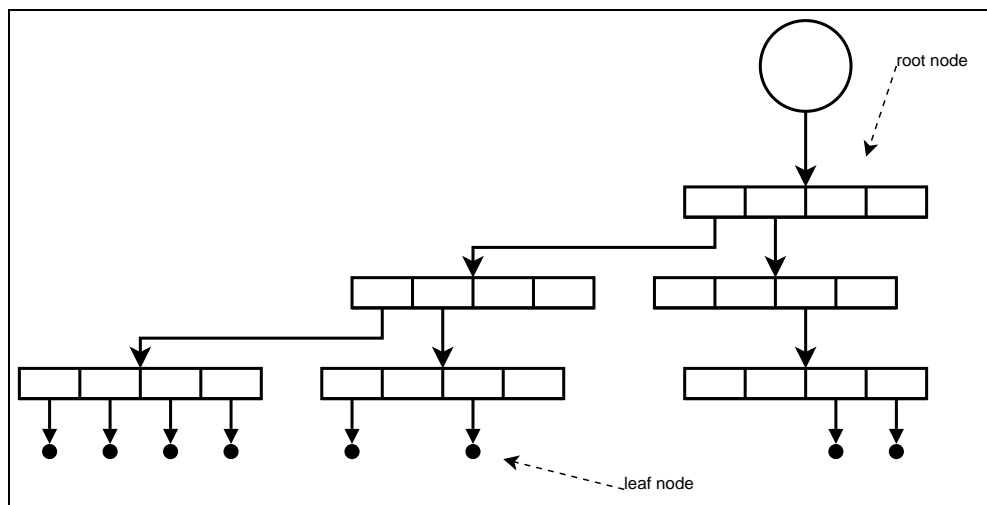


Figure 4.1: Radix-tree of height 3

Figure 4.1 illustrates the structure of a radix tree with 4 slots per node, and a root node that is 3 traversals away from the leaf nodes. Slots which do not have arrows coming down from them are empty (pointer contains NULL).

The radix-tree is of variable height, depending on the range of keys stored. The radix-tree lookup code must know the height of the tree it is traversing, so that it may determine whether a particular slot is a pointer to a leaf node or to a lower radix-tree node. The Linux radix-tree has an associated `height` attribute to store this information.

4.2 Lookups with concurrent modification

This section describes how radix-tree lookup operations can work in the presence of concurrent modifications to the tree.

Insertion, removal, and swapping of data items are the two high level modifications that can be performed on the radix-tree. Both operations can be broken down into a small number of basic, low level operations. These low level operations can be made atomic, with respect to a concurrent lookup, by careful ordering of instructions. It can then be shown that a concurrent lookup produces acceptable results regardless of the interleaving of operations.

The basic, low level modification operations are as follows:

1. Populate an empty slot with a pointer to a node or leaf-node.
2. Replace populated slot with a different leaf-node.
3. Clear a populated slot (make it empty).

4. Increase the height of the tree by 1.
5. Decrease the height of the tree by 1.

High level operations are implemented with combinations of low level operations. Insertion of a new item involves a combination of low level operations 1 and 4. Deletion of an item involves a combination of operations 2 and 5. Swapping an existing item for a new one involves operation 2.

If all the low level operations are shown to be safe in the presence of concurrent lookups, then the high level operations may run in the presence of concurrent lookups. The preservation of the semantics of the high level operations can then be argued by examining the consequences of possible interleavings of low level operations.

4.2.1 Slot modifications

4.2.1.1 Populate empty slot, clear populated slot

Operations to populate and clear radix-tree slots are trivially atomic with respect to lookup code due to the fact that storing a value to a pointer, and loading a value from a pointer is atomic in Linux. A concurrent lookup will only find either an empty slot or the valid pointer.

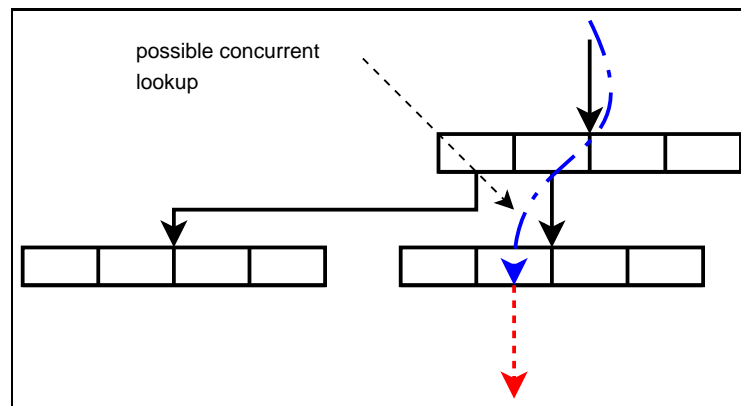


Figure 4.2: Population or clearing of a slot

Figure 4.2 illustrates a slot population or clearing operation in red, and a concurrent lookup operation in blue. Both the population and clearing operations are a special case of the general operation storing a value to a pointer. The lookup operation eventually loads this pointer, in which case it will find *either* the old or the new value.

When inserting a node (either leaf or non-leaf), the data in the new node must be first initialised; then a memory ordering instruction must be issued; then its address can be stored in the slot. This ordering ensures that a concurrent lookup will not find uninitialised data

in the new node. The correct memory ordering instruction is included in the RCU primitive `rcu_assign_pointer`, and was explained further in Section 2.6.4.

When removing a non-leaf node (clearing a populated slot), it must remain valid and allocated for as long as it is possible that a concurrent reader may still have a reference to the old pointer. This existence guarantee ensures that the concurrent reader will not operate on the node's memory after it has been freed and used by something else. This guarantee is provided for non-leaf nodes by RCU delayed freeing.

Note that non-leaf nodes are only ever inserted or removed when they have no children (are empty).

There are several types of interleavings of operations to be considered for correctness. Tables 4.1 and 4.2 provide a matrix of possibilities for non-leaf nodes and leaf nodes, respectively. The column headings of these tables describe the write-side operation, and the row headings represent the possible situations that a concurrent lookup may encounter. The intersection cell describes the subsequent behaviour in the given situation.

	non-leaf node being inserted	non-leaf node being deleted
lookup finds node	continues at node	continues at empty node, lookup fails
lookup finds NULL	lookup fails	lookup fails

Table 4.1: Lockless radix-tree lookup versus non-leaf node insertion and deletion

	leaf node being inserted	leaf node being deleted
lookup finds node	lookup succeeds	lookup succeeds
lookup finds NULL	lookup fails	lookup fails

Table 4.2: Lockless radix-tree lookup versus leaf node insertion and deletion

In the presence of node insertion or removal operations it is possible for a concurrent lookup to have an outcome that depends on the exact interleaving of operations, unlike a locked lookup that only has one. This is not a fatal deviation from the desired behaviour, but it does result in more relaxed semantics of the radix-tree lookup (described in 4.3).

4.2.2 Height modifications

4.2.2.1 Increase tree height

When increasing the height of the tree, a new root node is added and the previous root node becomes its left-most child. The critical part of this operation is switching the root node pointer from the old to the new node. This is done by an atomic pointer store after the new root node has been initialised, so a concurrent lookup traversing the pointer to the root node will find either the old or the new node both of which are valid.

Similarly to inserting a new node, this operation requires a memory barrier issued between initialising the root node and making it visible with the pointer store. Again, this is performed by `rcu_assign_pointer`.

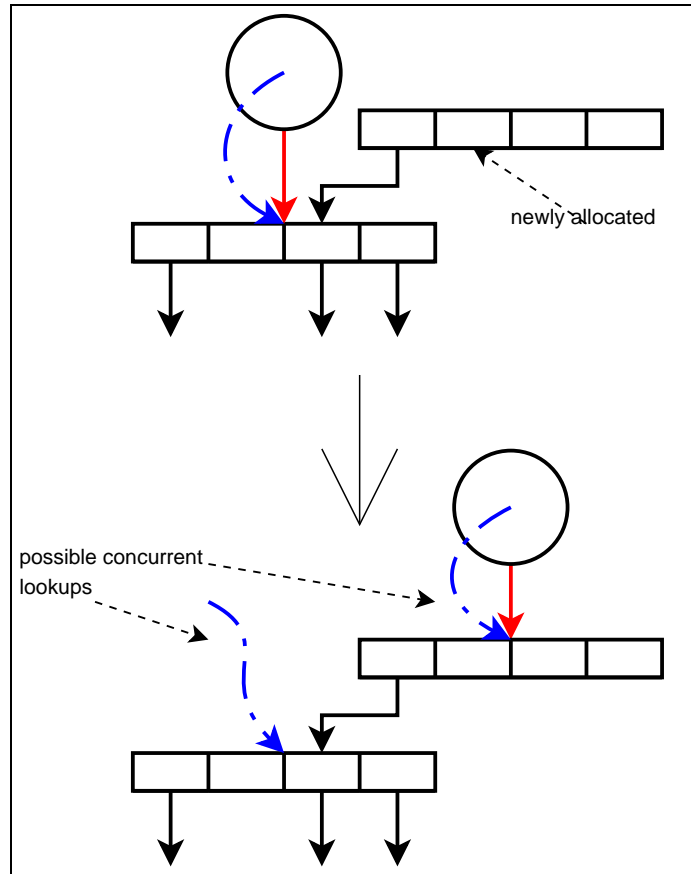


Figure 4.3: Increasing the height of the radix-tree

Figure 4.3 shows the process of increasing the radix-tree height. There are several combinations of lookup types and interleavings to consider; Table 4.3 illustrates that in each case, behaviour is unchanged regardless of whether the concurrent lookup finds the new or the old root. Column headings describe the key requested by the lookup, row headings describe the possibilities encountered by a lookup operation and a concurrent tree height increase.

	lookup key within old root	lookup key not within old root
lookup finds old root	continues at old root	old root out of range, lookup fails
lookup finds new root	leftmost slot of new root taken, continues at old root	if key out of range of new root, lookup fails; else all slots but leftmost empty, lookup fails

Table 4.3: Lockless radix-tree lookup versus tree height increase

4.2.2.2 Decrease tree height

Decreasing the height of the tree is similar to the increasing operation, in reverse. The root node is empty except for its left-most child, which becomes the new root. The old root is freed with the delayed RCU mechanism.

Figure 4.4 illustrates why the existence guarantee provided by RCU is required: a concurrent lookup may still be operating on the old root node *after* the root pointer has switched over; if the old root were immediately freed, the concurrent lookup may be operating on data that has been used for something else.

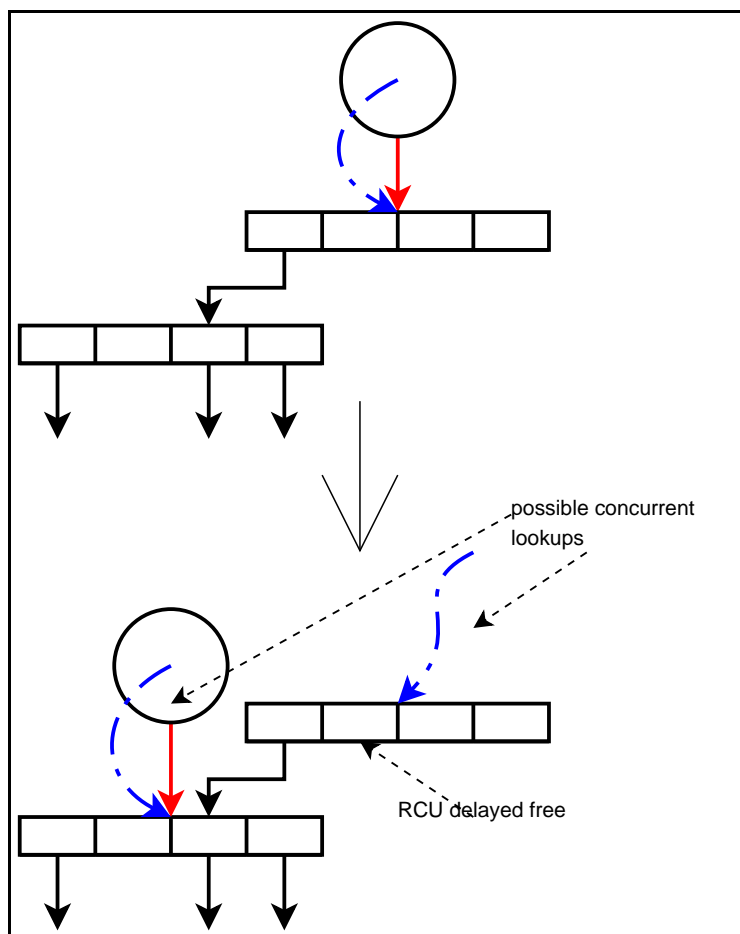


Figure 4.4: Decreasing the height of the radix-tree

Table 4.4 follows the same pattern as the previous table, for a height decrease rather than increase. The interleavings of concurrent lookups are symmetric to those with the height increasing operation, and behaviour can likewise be shown to be unchanged regardless of whether a lookup finds the old or the new root.

	lookup key within new root	lookup key not within new root
lookup finds new root	continues at new root	new root out of range, lookup fails
lookup finds old root	leftmost slot of old root taken, continues at new root	if key out of range of old root, lookup fails; else all slots but leftmost empty, lookup fails

Table 4.4: Lockless radix-tree lookup versus tree height decrease

4.2.2.3 Height attribute

The tree’s height attribute cannot be relied upon by lockless lookups, because a concurrent write-side operation may change the actual height of the tree, or the value of the height attribute at any time.

The solution to this problem relies on the observation that changes to the tree height are only performed by inserting a new root node or removing the existing root node, leaving the height of any sub-tree the same. Thus the height of a radix-tree node can be defined as the height of the sub-tree rooted at that node, and this height is invariant for the lifetime of the node. Figures 4.5 and 4.6 illustrate the node height invariant under modifications to tree height.

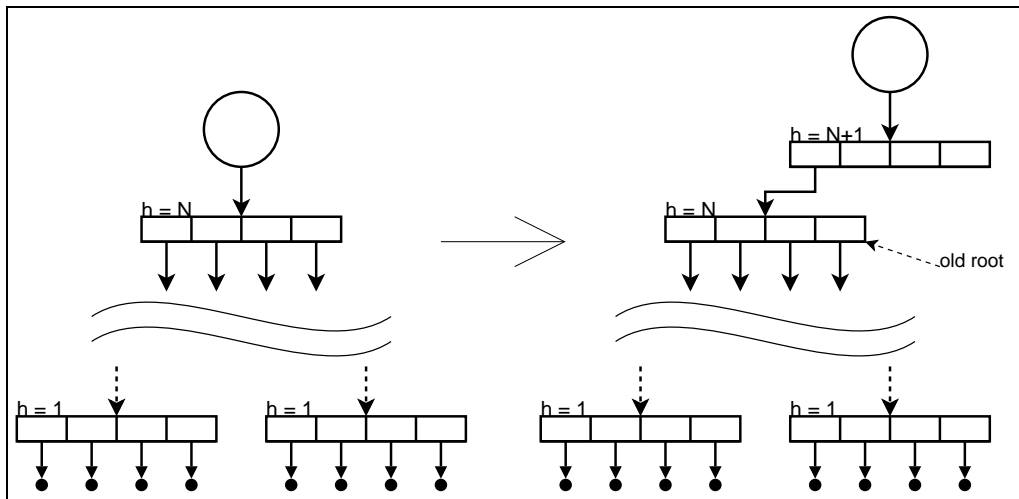


Figure 4.5: Node height invariant under increasing tree height

The node height attribute is set before the node is linked into the tree, and remains unchanged. Having a per-node height attribute allows a lockless lookup to determine the number of traversals required until a leaf node is reached, even when the tree height is concurrently being modified.

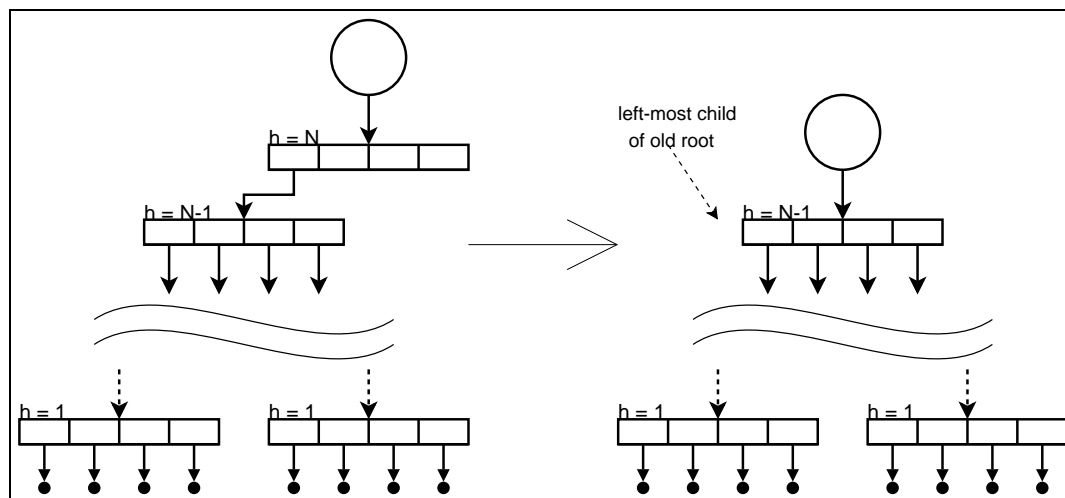


Figure 4.6: Node height invariant under decreasing tree height

4.3 Lockless radix-tree lookup semantics

While it has been shown that modification operations are safe to run in the presence of concurrent lookups, that does not guarantee that the semantics of lookups are unchanged. In fact there is a small difference, radix-tree lookups performed with locking have the following semantics:

Within the critical section, if the offset

- contained a particular item, it must be returned;
- was empty, NULL must be returned.

Lockless radix-tree lookups have more relaxed semantics due to the fact that RCU synchronisation allows the read-side to see stale data. When doing a lockless lookup of a particular offset, if the offset

- always contained a particular item, it must be returned;
- was always empty, NULL must be returned;
- ever contained an item, it may be returned;
- was ever empty, NULL may be returned.

It is important that when a lookup returns a data item, a concurrent deletion may have removed that item from the radix-tree before the caller had even examined the item. Thus it is important for the users of the radix-tree to ensure that any required *existence* guarantee is properly met for their data items (the use of RCU in the radix-tree only provides this guarantee for the non-leaf nodes).

4.4 Implementation details

This section describes the complicating factors involved in the Linux pagecache radix-tree, such as tags and gang lookups.

4.4.1 Radix-tree tags

One detail glossed over in the description of the radix-tree, and design of the lockless lookup are so-called radix-tree tags. In the Linux radix-tree, each slot has a corresponding set of tags which are implemented as a bitmap. These tags are part of the radix-tree node structure, and are set and cleared under lock.

‘Tagged lookups’ are lookups which return radix-tree entries which have a specific tag set, or may query which tags are set for a given entry. Tagged lookups may be performed in parallel, but they require exclusion from operations which set or clear tags.

The lockless radix-tree does not attempt to perform lockless tag lookups, and so requires that tag operations are still performed with the traditional synchronisation. Most tag operations used on the pagecache radix-tree are associated with relatively infrequent operations such as IO, where scalability is not so important.

There is no additional concurrency introduced between tagged lookups and non-tagged lookups by making the non-tagged lookups lockless, because both operated under a read lock. This meant that both operations could execute concurrently.

Additional concurrency *is* introduced between lockless lookups and tag setting and clearing operations (which were traditionally excluded from one another). However, lockless (non-tagged) lookups do not affect tag operations in any way, nor do any tag operations change the structure of the tree or modify any data which is used by untagged lookups. Hence neither operation effects the other in any way, thus any interleaving of the two operations is safe.

4.4.2 Gang lookups

The radix-tree has facilities to perform gang lookups that return, at most, the next N items starting from a given offset. It is possible to perform gang lookups without taking any locks for the same reasons that a single lookup is able to be lockless.

While the locked gang lookup guarantees that all items returned are present in the tree and that they are the only items present over the given range for the duration of the lock, the lockless gang lookup may return items that no longer exist and miss items that are now present. The semantics of the lockless gang lookup are the same as those for the lockless lookup applied to each entry in the range of the gang lookup, individually.

In Chapter 5, it will be shown how the lockless pagecache can cope with these slightly relaxed semantics.

4.4.3 Child count attribute

As well as height, child slots, and tag bits, the radix-tree non-leaf node contains a count attribute. The count attribute contains the number of children present in its slots².

The child count requires no extra consideration when moving to a lockless lookup because the it is only ever read or modified by write-side operations. These operations continue to maintain the same synchronisation with respect to one another.

4.5 Summary

This chapter described the theory and implementation of a simplified lockless radix-tree. Each possible interleaving of lookup versus modification was examined and shown to be safe. It was shown that the semantics of the lockless lookup are more relaxed than the traditional synchronised lookup, so callers must be able to tolerate this.

Then it was shown that this simplified radix-tree model is easily extended to accommodate the features of the Linux kernel radix-tree, such as tags and gang lookups. The lockless radix-tree is one of the components of the lockless pagecache.

²The child count is used to shrink the radix tree when entries are deleted.

A lockless pagecache in Linux

having a totally, absolutely lockless pagecache in the read and fault path is like the Holy Grail of OS design – Ingo Molnar¹

This chapter proposes a method for a lockless pagecache in Linux. By lockless, it is meant that pagecache lookup operations will be performed without taking a lock. Insertion and removal of pages, and ‘tag lookups’ are performed with the existing locking – these operations are associated with less frequent operations such as IO, file truncation, and page reclaim, so are less important.

5.1 Lockless pagecache outline

This section outlines the important concepts of the lockless pagecache method. The method involves the employment of a lockless synchronisation scheme for the basic pagecache lookup function `find_get_page`, then using `find_get_page` to implement more complex lookup operations `find_lock_page` and `find_get_pages`. This section outlines the ideas behind the lockless synchronisation scheme used in `find_get_page`.

In the lockless pagecache method, `find_get_page` is implemented without taking the per-inode `tree_lock`, where it is currently taken for *reading*. Hence, in order to show that correctness is maintained, it must be demonstrated that pagecache synchronisation requirements, described earlier in 2.7.3, are fulfilled by the lockless method.

5.1.1 Permanence of `struct page` (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

¹Ingo Molnar is a long-time Linux kernel developer who designed and implemented the scalable ‘O(1)’ scheduler, among other things. This quote is from a private communication with Ingo, with his permission to publicise it.

lock an object and another deallocating it, is to ensure that all references to an object are protected by their own lock [Gamsa et al. 1999]

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: *a `struct page` itself is never actually allocated or freed, only its associated page frame is.*

This can be made clearer by thinking about the example of free pages. Free pages are managed in lists in the page allocator, these lists are implemented as linked lists of the associated `struct page` structures. The actual page frame itself is free but it still exists, and its meta-data, the `struct page`, continues to maintain information about its state.

5.1.2 Speculative pagecache references (accuracy guarantee)

With the necessity for an existence guarantee alleviated, it is then possible to operate on the `struct page` data structure without the possibility that it may be concurrently freed.

`find_get_page` is called to take a reference on the page at a given location (`inode, offset`) in pagecache. The accuracy guarantee ensures that the given page has its reference count incremented while it is in the pagecache. It can be provided by ‘speculatively’ elevating the `struct page` reference count, and then verifying that the page is still in the expected location in the pagecache.

If the page is no longer at the same location in the pagecache after the speculative reference was taken, then it must have been removed from the location at some point, so this speculative reference is dropped and the operation retried.

An interesting corner case to consider is the case where a particular page is removed from the pagecache and freed, then re-allocated as a pagecache page for exactly the same location as it had been deleted from. It is not immediately clear that this case is correct – might this cause a concurrent reader to incorrectly ‘verify’ that the page is the correct one?

Suppose that the page is found by a `find_get_page` when it was in the pagecache the first time around. Then if a speculative reference is taken on the page at some point *after* the page has been deleted and reallocated, the pagecache will subsequently be found to be in the correct position in the pagecache, as if it had always been there. This case turns out to be no problem, because it is fully possible that the initial page lookup had taken slightly longer, and that the page is initially found after it had been added to the pagecache the second time. What is important is that the page which has had its reference count incremented is in the pagecache at the required location. Returning this page does not violate the semantics of `find_get_page` (see Section 2.7.2.1), because it existed in the pagecache at some point, hence it may be returned.

5.1.3 Lookup synchronisation point (no new reference guarantee)

The ‘no new reference’ guarantee is traditionally provided when holding the `tree_lock` for writing, which stabilises the page reference count by preventing `find_get_page` and similar lookup functions from running concurrently. Without holding `tree_lock` for reading, `find_get_page` is no longer prevented from running.

This problem is overcome by introducing a new bit in the `struct page`’s `flags` field. Sites that require the no new reference guarantee will set this bit. After a speculative reference is taken on a page, the page will not be checked for accuracy (as prescribed by the accuracy guarantee) until this bit becomes clear.

Essentially the bit has become a synchronisation point and has taken over from the functionality provided by `tree_lock`. This quasi lock allows a write-side operation to halt readers at a particular point, however readers do not have a critical section that blocks writers. Most importantly, it is not a *lock* that is taken by the read-side, it does not cause cacheline contention between multiple lookups of the same page (because it does not write to the cacheline) nor does it use expensive atomic locking instructions.

5.1.4 Providing guarantees in uniprocessor kernels

The Linux kernel offers a compilation configuration choice of uniprocessor (UP) or multiprocessor (SMP) capable 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 (there are no other processors). However interrupts must still be disabled to guard critical sections that may require exclusion from interrupts.

So a UP kernel already effectively has lockless pagecache lookup operations. The relatively complex mechanisms for providing pagecache synchronisation, described above, are not required on UP kernels. They are not required because all pagecache write-side operations which interact with lockless lookups are performed in process context and exclude interrupts. Lockless lookups 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 simply increments the page’s reference count.

5.1.5 Problems

The above outline for providing guarantees without locking is slightly simplistic in order to explain the important concepts first. There are some complexities which must now be examined; they come about because the lockless existence guarantee of `struct page` is not as strong as that which can be provided by using locks.

While the `struct page` itself is not 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. Aside from pagecache, a page may be free, used as a page table page, a buffer for a network packet, storage for an internal kernel data structure, or other things. Many of these users treat the `struct page` slightly differently, sometimes overloading fields for different purposes.

By allowing `struct page` ‘existence’ to span over these lifetimes, the previously distinct states of a page become blurred. 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. So when incrementing the `_count` field to for the speculative reference, it is possible for the page to be in any state.

Fortunately, the only attribute of a `struct page` which might be queried or modified outside of the normal lifetime of the page is `_count`, the reference count.

5.1.5.1 Free pages

Free pages have a `refcount` of zero and are managed by the page allocator. Speculatively elevating the `refcount` of a free page poses a number of problems:

- When dropping the speculative reference it is essential that a free page is not freed into the page allocator *again*, when the count reaches zero. However it is hard to tell if the page actually was free when its reference count was incremented (imagine a second speculative reference that had elevated the count from 0 to 1).
- 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 if it is not zero, and otherwise returns failure. This allows free pages to be detected and ignored entirely (if the page is free, it is definitely not in the desired pagecache location).

5.1.5.2 Page reference counting uniformity

A second problem is one of ‘page reference counting uniformity’ throughout the kernel. By the time a speculative reference has been taken on a page, it may have been removed from the pagecache, 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 page is not in the right location in pagecache. However before the speculative reference is dropped, the last *real* reference to the page may have been dropped, thus dropping the speculative reference will cause the reference count to transition to zero (which must free the page). So it is required that all kernel users treat the page's refcount in the same manner, and that dropping the last reference must free the page in the same manner throughout the kernel. This allows the incorrect speculative reference to be correctly dropped, and the page freed if necessary.

5.1.5.3 Page reference count instability

There is a third problem, again due to the fact that speculative references can be taken on a page when it is in any state. Now all pages returned from the page allocator may have an unstable refcount. Any page returned may have been a pagecache page, so a speculative reference might be taken on it at any time.

For this to not cause problems, no part of the kernel may assume the refcount is stable, nor should non-atomic operations be used to manipulate the refcount. It can be assumed that the refcount (which includes speculative references) is *greater than or equal to* the number of *real* references that are held at that time.

The *lookup synchronisation point* used to provide the 'no new reference' guarantee can be used, when necessary, that no more *real* references will be returned by the lockless pagecache lookup.

5.1.5.4 Why RCU is not used for the existence guarantee

RCU is not used to provide existence guarantees for a pagecache page. While this would be possible, and would avoid the difficulties caused by the weaker existence guarantee, RCU has problems of its own.

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

- The extra work required to visit the page again will introduce overhead.
- If the list grows to a significant size, each `struct page` can be evicted from the CPU's cache before being visited, introducing cache misses on each page visited.
- The page allocator has per-CPU lists of free pages, which can be accessed locklessly. Page allocator locks need only be taken when these lists overflow or underflow. The RCU queued stage before reaching these per-CPU lists will increase the incidence of underflow while the pages are being held for delayed freeing, and of overflow when they are finally released.

- 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 RCU queued 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. Low memory problems have been a problem with RCU in the past.

RCU *is* used for delayed freeing of the nodes of the lockless radix-tree, however that are less affected by the above problems. A radix-tree node is much smaller than a page, and are usually allocated and freed significantly less often and they are usually allocated and freed significantly less often than pagecache pages (one node holds up to 64 pagecache pages).

5.2 Lockless pagecache operations

The methods outlined above will now be made concrete by demonstrating how they provide the required semantics. The most important operation is `find_get_page`, and the others are built on top of that.

5.2.1 `find_get_page`

The lockless `find_get_page` is the fundamental lockless operation upon which the others are built. Figure 5.1 gives the C code for the lockless `find_get_page` for SMP kernels, with some comments removed, and preprocessor macros expanded, for clarity.

The uniprocessor implementation is omitted because it is trivial, with lines 12-24 replaced by an unconditional atomic refcount increment (see Section 5.1.4).

5.2.1.1 Implementation walk-through

Line 6 performs the RCU read-side ‘lock’ (*not* a traditional lock, see Section 2.6), to enter an RCU read-side critical section required by the lockless radix-tree lookup.

In lines 8-10, a lookup is performed on the radix-tree. `find_get_page` returns NULL if the lookup failed to find a page, otherwise the variable `page` is assigned the address of the struct page of the page found.

At line 12, the page’s refcount is incremented if it was not previously 0; if it was, the whole operation is restarted (this operation requires the permanence of struct page for the existence guarantee.)

```

1: struct page *find_get_page(struct address_space *mapping,
2:                            unsigned long offset)
3: {
4:     struct page *page;
5:
6:     rcu_read_lock();
7:     again:
8:     page = radix_tree_lookup(&mapping->page_tree, offset);
9:     if (!page)
10:        goto out;
11:
12:     if (!get_page_unless_zero(page))
13:        goto again; /* page has been freed */
14:
15:     while (PageNoNewRefs(page))
16:        ; /* wait for NoNewRefs to become clear */
17:
18:     smp_rmb();
19:
20:     if (page != radix_tree_lookup(&mapping->page_tree, offset)) {
21:        /* page is no longer there, retry */
22:        put_page(page);
23:        goto again;
24:     }
25:
26:     out:
27:     rcu_read_unlock();
28:     return page;
29: }

```

Figure 5.1: Lockless find_get_page for SMP

Lines 15-16 busy-wait while the page’s ‘NoNewRefs’ flag is set (this flag can be set to enforce the *no new references* guarantee.)

The `smp_rmb()` on line 18 is a memory barrier, which is required to ensure the NoNewRefs flag is clear *before* the page is verified to be in pagecache.

When NoNewRefs is clear, lines 20-24 recheck that this page is present in pagecache (which provides the accuracy guarantee). If the recheck found the page in the correct position, then the operation is successful and the page is returned; otherwise, that page’s refcount is decremented (and will be freed if that caused it to reach 0), and the operation is restarted.

Line 27 exits the RCU read-side critical section, which was entered in line 6.

Note that, `find_get_page` 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. Comments in the source code [Piggin 2006] of the implementation explain all memory ordering in detail.

5.2.1.2 `find_get_page` return value

It is important to ensure that the possible return values of the lockless `find_get_page` implementation remains the same as those for the implementation using locks.

Note that, the return values for `find_get_page` (see Section 2.7.2.1) match the return values for the lockless radix-tree lookup operation returns (see Section 4.3). With that in mind, it can be verified (from Figure 5.1) that the value returned by the lockless `find_get_page` will match those required.

If the pagecache location given by `(mapping, offset)`

- *always* contained a particular page, it will be found on line 8, its refcount will be non-zero so line 12 will succeed, then line 20 will find it is still in pagecache, so that page will be returned.
- was *always* empty, line 8 will find no page, so `NULL` will be returned.

Then it is also easy to see that if the page is concurrently removed or added, it *may* be returned, or another page, or `NULL` *may* be returned, without violating the `find_get_page` semantics.

5.2.1.3 `find_get_page` side-effects

Next, a returned page's refcount must be elevated *while it is in the pagecache*.

It can be seen that after incrementing the page's refcount on line 12, it is then verified that the page is indeed in the correct position in the pagecache in the test on line 20. If this test succeeds, then it is true that page did have an elevated refcount while it is in the correct location in the pagecache.

5.2.1.4 Satisfying the no new references guarantee

`find_get_page`, being a pagecache lookup function, must not return a reference to a page to any caller while the no new references guarantee is being enforced. The no new references guarantee is required when a clean page with no references is to be discarded from the pagecache (see 2.7.3).

The no new references guarantee is provided with a lockless synchronisation protocol between the code that discards pagecache pages, and `find_get_page`. The steps of each operation is shown in Table 5.1. These steps are atomic with respect to concurrently executing read or write side operations. And they will be carried out, and visible, in the orders shown, due to memory barriers and dependencies in the code.

find_get_page	discard page
1. find page in radix tree	A. set PageNoNewRefs
2. conditionally increment refcount	B. check refcount is correct
3. wait for PageNoNewRefs	C. remove from pagecache
4. check the page is still in pagecache	D. clear PageNoNewRefs

Table 5.1: Steps involved in find_get_page and discarding a page

It is possible that these operations may be interleaved and executed concurrently in almost any order, however there are just 2 critical orderings that need to be considered:

- 2 runs before B: in this case, B sees elevated refcount and bails out.
- B runs before 2: in this case, A ensures 3 will not complete until after D is finished, and thus 4 will occur *after* C. In which case, 4 will notice that C has removed the page, and find_get_page will retry.

It is possible that between 1 and 2, the page is removed then the exact same page is inserted into the same position in pagecache. This corner case is considered in 5.1.2.

5.2.2 find_lock_page

The lockless find_lock_page follows this pattern, using the lockless find_get_page to take a reference to the page without locking. The implementation is given in Figure 5.2.

The lockless find_lock_page uses the standard construct of taking a reference to the page, then locking it, then rechecking that it is still in pagecache. This is enough to satisfy the required semantics, given in 2.7.2.2.

5.2.3 find_get_pages

The find_get_pages function finds up to a specified number of pages from a given offset in a file, 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 a new lockless function, find_get_pages_nonatomic, is introduced which is implemented as the equivalent of multiple calls to find_get_page (optimised by using radix-tree gang lookup).

Fortunately, callers of find_get_pages are much less common than those to find_get_page and find_lock_page, so this is less important. Also, many of the callers were actually con-

```
1: struct page *find_lock_page(struct address_space *mapping,
2:                             unsigned long offset)
3: {
4:     struct page *page;
5:
6:     repeat:
7:     page = find_get_page(mapping, offset);
8:     if (page) {
9:         lock_page(page);
10:        /* Is the page in the correct location? */
11:        if (page->mapping != mapping ||
12:            page->index != offset) {
13:            /* No, retry the operation */
14:            unlock_page(page);
15:            page_cache_release(page);
16:            goto repeat;
17:        }
18:    }
19:    return page;
20: }
```

Figure 5.2: Lockless find_lock_page

verted to use `find_get_pages` instead of multiple calls to `find_get_page`, for efficiency reasons. These callers do not require the full atomicity of `find_get_pages`, so they may use the lockless `find_get_pages_nonatomic`.

5.2.3.1 Truncation and invalidation

Truncation and invalidation are the main operations which use `find_get_pages` (via `pagevec_lookup`). They are typically invoked on a range of pages in a file, and `pagevec_lookup` is used to find these pages.

The truncate and invalidate operations themselves only operate on a single page at a time, which does not rely on any other pages. So it is possible to use the lockless `find_get_pages_nonatomic` as their pagecache lookup function, instead of `find_get_pages`.

Performance results

In this chapter, the performance properties of the lockless pagecache is presented, and compared with the standard Linux 2.6 `tree_lock` based pagecache synchronisation.

6.1 Benchmarking methodology

The benchmarks presented here aim to give a fair representation of the micro and macro performance behaviour of the various pagecache synchronisation schemes.

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; locking, atomic operations and memory consistency operations can differ significantly.

Unless indicated, benchmarks are run 10 times, and the error bars represent a 99.9% confidence interval. Given the repeatability of many tests, errors can be small enough that the error bars are unable to be discerned.

6.1.1 Kernels tested

The ‘standard’ kernel tested was 2.6.17. The ‘lockless’ kernel is 2.6.17 with the lockless pagecache patches [Piggin 2006].

6.1.2 Machines tested

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). 4GB RAM.

Altix SGI Altix 3000. 64 CPUs (Itanium 2, 1.5GHz, 3MB L2), 64GB RAM (32 nodes).

6.2 Pagecache lookup scalability benchmark

First, pagecache lookup scalability is tested within the context of a page fault operation that is performed by the kernel. Page faults in a memory area backed by an `mmap()`ed file must query the pagecache to find the page for the given file offset. This benchmark was performed on the Altix machine.

When such an area is first mapped, the kernel registers the region as being backed by the file, but the process's pagetable is not populated. So upon first access to a page in the mapped area, a pagefault occurs which calls `find_get_page` in order to find the physical page that the virtual address should refer to.

Scalability was tested by varying the number of concurrent processes accessing a single file. Each process accessed a unique region of the file. The file was made resident in pagecache before the test began (so IO is not involved). The pages in each region were allocated from the same node as the process accessing them would run. Tests were run from 1 to 64 processes, each bound to a different processor, providing a workload scaling from 1 to 64 processors.

Figure 6.1 shows this benchmark running on the standard kernel. The x axis on the graphs correspond to the number of processors used in the test, while the y axis corresponds to the total pagefault throughput for all processors combined.

The standard kernel performs about 16GB/s of pagefaults with a single process; this corresponds to about 1 million calls to `find_get_page` per second. At 2 processors, throughput is up to 22GB/s which corresponds to 11GB/s per processor (with ideal scaling, total throughput should be 32GB/s). Beyond 2 processors, total throughput actually falls below that which a single process can achieve; at 64 processors, throughput is only $\frac{1}{4}$ of that which a single process can achieve.

Figure 6.2 shows the lockless kernel scales much more efficiently than the standard kernel. At 64 processors, the lockless kernel achieved almost 250 times the throughput of the standard kernel, and attains about 80% of ideal scalability. Total throughput approaches 1TB/s. The lockless kernel does not quite reach 'ideal' throughput (the throughput attained with a single processor, multiplied by the number of processors used). A possible reason is that the test also involved regular unmapping and remapping of memory regions (to generate a sufficient load without using more memory than RAM). These operations are likely to cause some cacheline contention which would impact scalability.

Note that Figure 6.2 includes the results of the standard kernel from Figure 6.1. Given the difference in magnitude, the standard implementation appears as a flat line at the bottom of the graph!

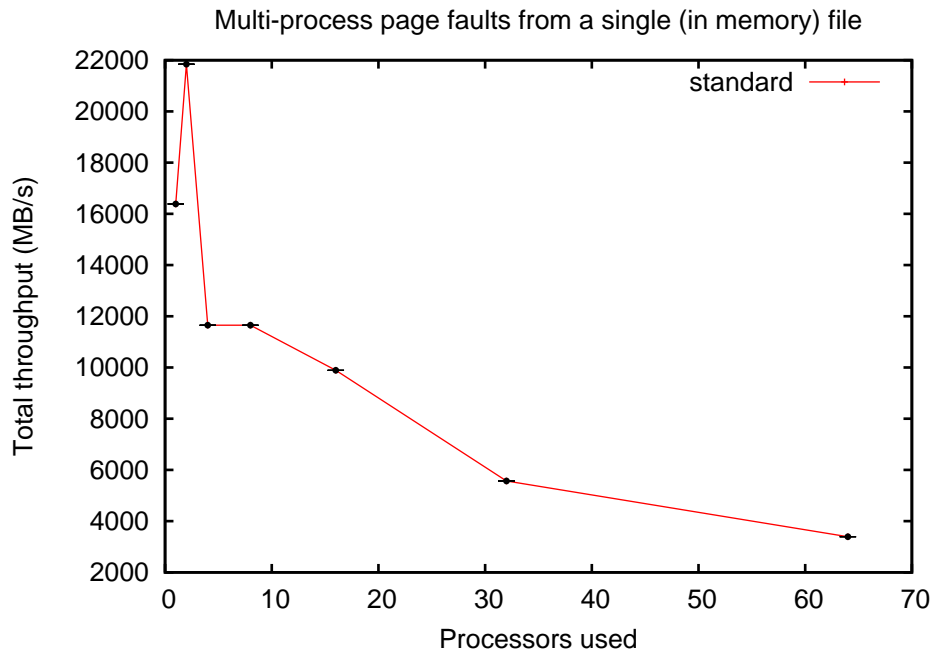


Figure 6.1: Pagefault scalability, standard kernel

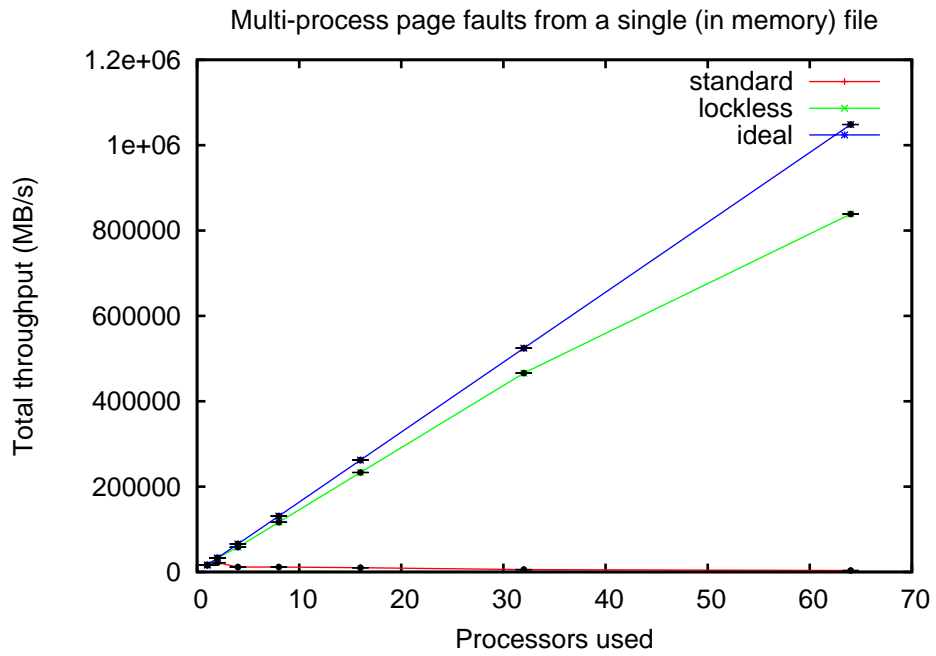


Figure 6.2: Pagefault scalability, lockless vs standard kernel

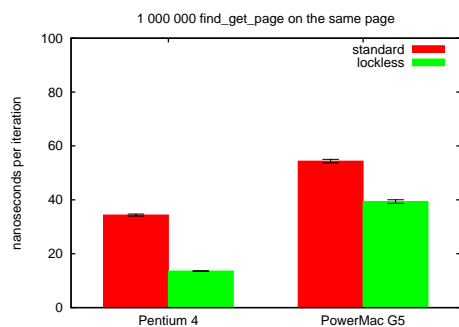


Figure 6.3: `find_get_page` on UP kernel, cache hot

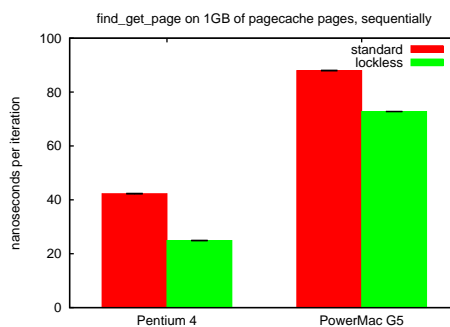


Figure 6.4: `find_get_page` on UP kernel, cache cold

6.3 `find_get_page` kernel level benchmarks

When improving scalability, it is common for single-threaded performance to be degraded. This is due to more complex code, more branches, atomic operations, and memory barriers in the read-side operations. This is one of the biggest problems when justifying scalability improvements, because the vast majority of Linux installations operate on single or dual processor systems. Thus it is important to get an idea of the change in single-threaded performance.

The following tests were performed by a loop running in kernel mode calling the `find_get_page` operation many times in order to amortise the overhead of the single system call used to enter the kernel¹. All the `find_get_page` tests are performed on a single file.

6.3.1 `find_get_page` single threaded benchmarks

Single threaded performance on SMP compiled kernels was tested from by looking up a single page 1 000 000 times (Figure 6.5), and by looking up each page of a cached 1GB file in turn (Figure 6.6). In the former test, the working set should completely fit in the cache of all CPUs (the ‘cache hot’ case); in the latter case, each `struct page` being operated upon will not be in CPU cache (the ‘cache cold’ case).

Uniprocessor (or UP) kernels were also tested in single threaded benchmarks. UP kernels have spinlocks optimised away during compilation, and the lockless pagecache implementation also includes some conditional compilation optimisations for uniprocessor kernels. Figures 6.3 and 6.4 show the results of the same two tests on UP kernels.

¹`fadvise` was modified, and used to initiate the loop.

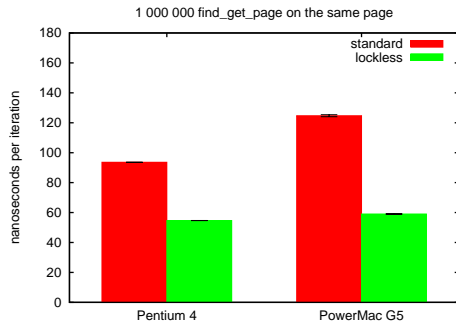


Figure 6.5: *find_get_page* on SMP kernel, cache hot

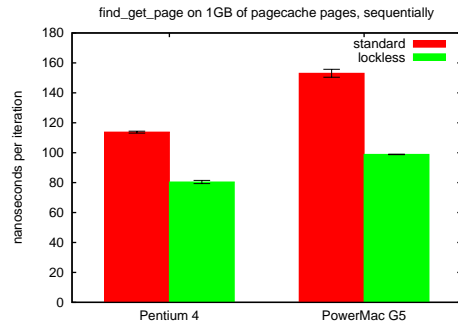


Figure 6.6: *find_get_page* on SMP kernel, cache cold

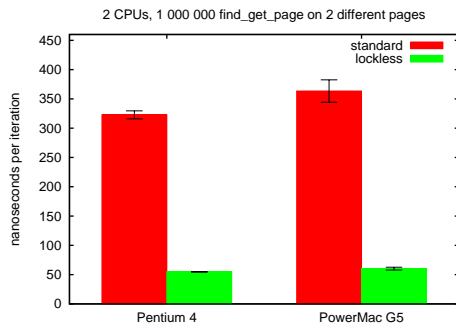


Figure 6.7: *find_get_page* on SMP kernel, two threads, different pages

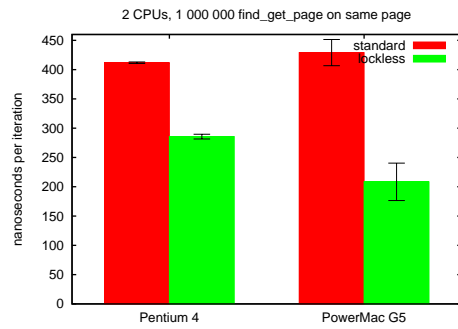


Figure 6.8: *find_get_page* on SMP kernel, two threads, same page

6.3.2 `find_get_page` multi threaded benchmarks

Multi threaded performance was tested by having two CPUs running the loop concurrently. Each CPU will call `find_get_page` 1 000 000 times, both on the same page in Figure 6.8), then on different pages in Figure 6.7.

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

6.4 IO and reclaim benchmark

When improving the scalability of read-side operations, write-side performance is often degraded. This degradation is a result of more complex operations, more atomic operations, and more memory barriers in the write-side. It is important to ensure that write-side operations are not impacted badly.

Page reclaim is an important operation for the kernel, as it is part of almost any workload that is filesystem IO intensive, and where 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 introduces changes to both.

Figure 6.9 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.

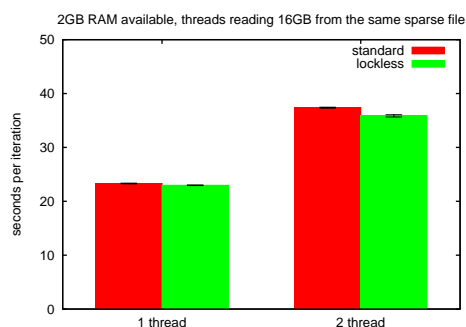


Figure 6.9: Page reclaim, SMP kernel

Conclusion

7.1 Summary

Trends in hardware and software development are increasing the relative cost of lock based synchronisation. This is particularly evident in multiprocessor systems, where the cost of cacheline bouncing can result in poor scalability if frequently executed operations are required to take a lock.

One solution is to implement a lockless synchronisation method. Traditionally lockless synchronisation has been difficult to implement for shared, dynamic data structures, however the Read-Copy Update algorithm provides a framework to provide the difficult ‘existence’ guarantee without using a lock.

The Linux pagecache is a data structure that must synchronise access by multiple processors. Linux 2.4 uses a single lock for access to the pagecache, which was shown to be a bottleneck on multiprocessor systems of just 4 CPUs. Linux 2.6.17 uses a lock per inode, which performs much better when accesses are spread over multiple files. However not all workloads spread accesses over multiple files, furthermore, Linux is being run on much larger multiprocessor systems today.

The pagecache is a central part of the Linux kernel, and is responsible for all file caching. The scalability of the pagecache is an important issue in Linux.

This paper presented the lockless pagecache method for Linux, which was shown to greatly improve the scalability of common pagecache read-side operations on single files. Performance of some operations was improved by nearly 250 times on a 64 CPU system. The lockless pagecache was also found to improve single threaded performance, and did not negatively impact the performance of write-side operations.

7.2 Future work

There are two avenues for further performance improvements that are made evident by this work. First is to improve the performance of write-side pagecache operations, such as inserting and removing pages from the pagecache. The second is to improve the read-side scalability of paths that make use of the pagecache read-side operations, for example it was found that read system call performance had scalability problems beyond the pagecache lookup.

There is also a need to evaluate the performance of the lockless pagecache in real world workloads on large systems. This effort is difficult because of the cost of such systems and the amount of work involved in setting up realistic workloads to test.

This work is has not been included in the official kernel as of the time of writing, though it has been submitted to kernel lists for review several times. Having it included in the kernel would require further review and testing, and probably evidence that it improves performance significantly for a range of workloads.

Engineering report

The software engineering processes involved in this research project share many similarities with the process used in a software development project. However, there are also differences, and considerations that are unique to such a research project.

This report describes the software engineering processes involved in developing the lockless pagecache for Linux, and contrasts these with the processes involved in a non-research project.

A.1 Project management

The project management activities involved in this project were minimal. Essentially I was the only person on the project team. Constraints were not too demanding and deadlines were reasonable. For these reasons, there was no team management to be done, and the estimates and planning work was minimal.

However, there were real elements of project management involved. There was a set of deliverables, and a hard deadline at the end of the year. There was a contract, a project schedule, and risks.

A.1.1 Constraints

There were two main constraints on the project, time, and scope. The work done to assess these constraints was important for the success of the project. However compared with a development project, the constraints are fewer and simpler.

A.1.1.1 Time

Time was a constraint applied by the course requirements. Two deliverables were due, with a hard deadline of the end of the school year. Time had to be allowed for literature review, design and implementation, testing and evaluation, and writing up the findings in a thesis.

Due to the simplicity of the timeline and the few deliverables, no formal planning or documentation was required to manage the project schedule. Rough milestones were agreed upon with my supervisor but they were very useful to keep the project on track.

These milestones were: literature review, implementation, testing, seminar presentation, final thesis draft, and final thesis submission.

A.1.1.2 Scope

The scope is constrained at minimum by the course requirements, which is to undertake a useful research oriented project and contribute to a field of computer science or software engineering. It must be something interesting enough for a supervisor to take on.

The scope is bounded above by the time allowed for the project, and the resources that can be put into it by an individual (being an *individual* project).

This project didn't encounter problems with or much volatility of scope. I had an idea for the lockless pagecache method before the start of the project, and estimated that it could be implemented well within the time allowed. After an implementation was developed, the plan was then to gather results, and present it as a research paper.

The scope did change slightly early on in the project when it was suggested by my supervisor that the scalability problem of cacheline bouncing should be given a theoretical basis and not just an empirical one, and that it would be helpful to be able to predict the performance improvement that the lockless pagecache would give. This resulted in the addition of Chapter 3.

A.1.2 Process activities

A.1.2.1 Project planning

Though the project plan was not laid out in a formal document, there were a number of distinct intermediate steps that needed to be completed for completion of the deliverables.

Related work and research had to be investigated, a working implementation had to be created, empirical testing had to be performed, and the seminar and thesis themselves had to be completed.

The plan was not very complex: with essentially only one significant schedulable resource (myself), every task is on the critical path. So the plan was to keep working on the unfinished items that were not blocked by other unfinished items. Rough milestones were set throughout the project timeline to ensure it wasn't going off track.

A.1.2.2 Risk management

In some sense, the risk involved in a research project is higher than in a software development project. The work, by definition, must be a new contribution to the field, so there is more uncertainty involved.

For example, the lockless pagecache method could have been found to have a fundamental correctness problem before the project had finished. If this could not be worked around by changing the algorithms used, then it would have been a big problem for the project.

The best plan to manage this risk was to attempt to be thorough in testing and arguing the correctness of the method, and ask for peer review of the ideas and implementation, early and often.

In other aspects, the risk is lower than what might be expected in a development project. The actual amount of work done (in terms of project management and deliverables) is smaller, involving just one person over one year.

The management of these risks – of the project slipping or the deliverables to be of poor quality – was to meet with my supervisor often, and get feedback and review. Also trying to have good personal time management was important. Aiming to finish the project early would allow for slips in the schedule.

Actually two unforeseen issues did arise in the course of the project. The first was that I was invited to a conference in Canada to present a paper (on the lockless pagecache work), which took nearly two weeks, including recovery from jetlag; and the second was that I injured my hand which required surgery and four weeks in a cast, during which time I didn't do much work. Despite these issues, I plan to hand in the thesis on time. Although the schedule has, not surprisingly, slipped more than I had planned, this demonstrates that the small amount of risk management that was done paid off.

A.1.2.3 Resource management

There was not many resources critical to the project that had to be managed. As far as computer systems went, I have access to a 64-CPU SGI Altix system through my employment at SuSE Labs, Novell Inc. that I was able to run the tests on. I didn't need to spend a very long time performing tests on this system, and I was normally able to get access to it quite soon after it was requested. Other computer systems used were my own personal computers, so no management was needed there.

Human resource management was only required so far as scheduling meetings with my supervisor, and requesting reviews. Eric was always accommodating and helpful, so there was no real need to do any 'management' here!

A.1.2.4 Work breakdown

Work breakdown was mostly a matter of personal time management, with broad milestones set with input from my supervisor.

Regular meeting times needed to be scheduled with my supervisor, however these were generally short meetings, and with only two people, rescheduling the meeting was an easy task.

A.1.2.5 Tracking, reporting progress

Progress was tracked mostly using a simple text-based ‘done and todo list’. For the thesis, the partially finished thesis document itself was helpful to see what items needed work. The software implementation, while complex, was quite small, so there was no need for tracking progress beyond the text-based list.

Reporting progress was a simple matter of summarising my progress to my supervisor at regular intervals.

A.1.3 Process artifacts

There were a number of artifacts produced by the project process. These were the contract, the charter, the implementation, and the thesis. It was found unnecessary to put other traditional artifacts, such as the requirements, into formal documents.

A.1.3.1 Project contract

The project contract was formed with my request to undertake a research project, and the subsequent agreement from my supervisor, Eric McCreath, and the software engineering research project convener, Chris Johnson.

A formal document called the ‘Independent Study Contract’ was signed. This document is part of the requirements for the software engineering research project unit.

A.1.3.2 Project charter

A project charter was written by Eric and myself, and was attached as part of the ‘Independent Study Contract’.

This charter gave the project a title, start and anticipated end dates, goals and timeframes for goals. The project members identified (myself, my supervisor Eric), and our roles defined.

A.1.3.3 Feasibility study

Scoping and feasibility was discussed between Eric and myself. There were no major feasibility problems identified, and scope did not deviate significantly from initially planned.

A.1.3.4 Plans, schedules, reports

In semester 1, Eric and I scheduled fortnightly meetings in which I reported the status of the schedule, and issues I had run into. Eric would review and provide suggestions and direction. During semester 2, the frequency of meetings was increased to once a week.

The schedule followed a standard research project schedule (literature review, implementation, testing, writeup, etc).

A.2 Project lifecycle

The project lifecycle did play a part in this research project. It was not so formalised as it would be in a software development project, due to the relative simplicity of the process management side of the project.

A.2.1 Process model

A.2.1.1 Iterative, evolutionary prototyping

The process model used to develop the lockless pagecache implementation and this thesis, was an iterative one which included early prototypes and samples of documents / software in order to get reviews and testing performed. The results of these reviews and tests were incorporated back into the prototypes rather than discarding them.

This process was only carried out on a small scale, however it worked well. It provided a working implementation early, that could be peer reviewed on the public Linux Kernel Mailing List. It also meant that the thesis document could evolve over the life of the project in response to review and input, and changes in direction.

A.2.2 Process activities

The activities in the research project can be described in terms of their software development project counterparts. Although some steps are software centric, similar concepts can be applied to the other deliverables as well (eg. the thesis).

A.2.2.1 Requirements analysis

Requirements analysis and specification had two parts. The first was the requirements set out by the course description, and agreed upon with the course convener and supervisor. These were negotiable, but included things like the start and end dates, deliverables, and reporting / meetings.

The second aspect was the requirements I set myself. I chose this project topic, and in some sense I was one of the clients. I had an expectation of what I wanted to achieve so these are an implicit part of the requirements. These requirements *were* also subject to guidance and input from my supervisor.

The process of discovering these requirements, either explicitly by agreeing on deadlines and deliverables; or implicitly by thinking about my expectations of the project, were a form of requirements analysis.

The requirements included:

- Examine past and present pagecache implementations in Linux.
- Introduce some techniques such as IBM developed RCU, required for a lockless page-cache.
- Propose an original model for a lockless pagecache in the Linux kernel.
- Examine the feasibility of such an implementation.
- Develop, verify, and test the implementation.
- Argue convincingly that correctness is achieved.
- Present various benchmarks comparing implementations.

This was later expanded

A.2.2.2 Architecture / design

The architecture and design of the software implementation was not a major undertaking. After the idea was conceived, the implementation was a fairly small conversion from one algorithm to another (albeit complex) one.

Coming up with the idea in the first place is perhaps not something that fits into the software engineering lifecycle. Although large projects probably commonly require creative thought to solve problems, it is much less risky to use existing methods and algorithms in a solution. In some sense, software development projects that depend on finding fundamentally new solutions to problems are, in part, research projects as well.

Designing performance tests (as opposed to QA tests), was quite simple and was just a process of simply defining what it was that I wanted tested, and then writing the small test programs or scripts to carry that out.

The design of the thesis itself was important. This was started early, by taking the ANU template, and looking at other theses and papers to come up with a tailored skeleton document. This proved invaluable as structure and guideline for future work, though it also required changes when shortfalls were revealed during the ‘implementation’ of the document.

A.2.2.3 Implementation

It was a large effort to go through and audit the kernel and submit the various adjustments required for this method to work.

The lockless radix-tree and lockless pagecache methods themselves weren’t a big task to actually turn into code: the final implementation is under 1 000 lines added, removed and changed. The bulk of the work was probably in QA and testing, detailed below.

Implementing performance tests and gathering data was straightforward. None of the tests were big or complicated.

Implementing the thesis and this engineering report was a large undertaking. I found it was difficult to put complex technical ideas into comprehensible English. I found it was simply a matter of putting in time, and putting in time to iteratively review and reimplement.

A.2.2.4 QA / testing

Quality testing was an interesting part of this project. The software implementation itself was very complex, so a number of approaches were taken for QA.

Firstly, external review and comments were sought frequently. The concept was described and when a working implementation was created, they were sent to the public Linux Kernel Mailing List with a request for review and comments. This was a fruitful process, with several helpful discussions and issues raised with my approach. For example, upon performing a code inspection, Paul McKenney [McKenney 2006b], RCU co-inventor, discovered several implementation problems with the lockless radix-tree. These issues were able to be quickly fixed.

Secondly, an attempt was made to convincingly argue the correctness of the algorithms, and examine all possible concurrency situations. This did not amount to a formal proof of the entire implementation, but it did provide some confidence in the correctness.

Lastly, correctness testing was employed. It is not enough to convincingly argue correctness or necessarily even prove the correctness of the software if there has been a mistake somewhere.

The mistake might be in the implementation or an assumption. With concurrent algorithms, the problem of testing is magnified because failure cases can often only be hit in rare circumstances and never in the testing of normal workloads.

The lockless radix-tree was perhaps less conceptually difficult than the lockless page synchronisation, however the implementation was much larger and more complex in practice. Testing the radix-tree in the kernel had proven inadequate, even when designing tests specifically to exercise the desired cases. The reason for this is that the radix-tree manipulation has to be driven via pagecache manipulation, so the actual time spent in the radix-tree is very small and concurrency isn't exercised well.

Instead, a concurrent user-space test harness was implemented that could really stress the concurrency in isolation – a unit test. This required creating an RCU implementation in userspace, which was not a trivial task itself, although the implementation could afford to be much less sophisticated than in Linux. This test harness discovered several problems on my 2 CPU workstation within the first few minutes of running that were not discovered even after hours of running kernel level tests on the 64 CPU Altix.

Integration testing of the whole kernel was also employed, and was also helpful to catch implementation bugs and oversights.

A.2.2.5 Documentation

Documentation of the software is provided by comments in the source code itself; and by this thesis, which is covered above as a 'first class' deliverable.

A.2.2.6 Maintenance

This software has not been deployed in released software, so the project did not involve any maintenance work.

However, if the implementation gets included in the kernel in future, it would follow the Linux / open source maintenance methodology. The code will be reviewed by all interested developers. When a reasonable consensus is reached, it will be tested in various stages of alpha and beta releases. Upon being deemed ready, the code will be included in a release kernel, which can be deployed in production.

I would be somewhat responsible for problems and issues reported, but would not have ultimate control of the code, however. Others could propose their own improvements and bug fixes, or argue against a future change proposed by me.

A.3 Configuration management

Roger Pressman [Pressman 2000] describes configuration management as a

set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Configuration management in this project involved the implementation of revision control systems for the main project artifacts.

A.3.1 Revision control

Revision control was the main form of configuration management required by this project. Revision control was required for the thesis, and the implementation.

The revision control system used was CVS for the management of the thesis document (and presentation slides). CVS was chosen for its familiarity and relative simplicity.

The ‘quilt’ patch manager was used to manage the implementation patches, with CVS used to provide versioning.

A.4 Conclusion

In conclusion, the software engineering processes used in this research project were important to the success of the project. Planning, risk management, requirements analysis, and testing are some of the processes used, that were particularly important. For example, due to reasonable planning and risk management, the project was able to tolerate an unforeseen schedule slip of 4-6 weeks; due to specialised unit testing, bugs were discovered that could have otherwise gone unnoticed into a production environment in future.

While there are parallels with a development project, there are also differences in the focus of the effort. For example, in the research project there was little project management to be done, because of the much simpler or non-existent project, development, and team, management issues. However there was a much larger relative emphasis on correctness including testing, inspections, and development of arguments for correctness.

While the full software engineering and project management approach was too much for this project, and not always applicable, there were many similarities in the issues faced. Ignoring good software engineering practices could have resulted in disaster.

Bibliography

- AGARWAL, A., SIMONI, R., HENNESSY, J. L., AND HOROWITZ, M. 1998. An evaluation of directory schemes for cache coherence. In *25 Years ISCA: Retrospectives and Reprints* (1998), pp. 353–362. Available: <http://citeseer.ist.psu.edu/agarwal88evaluation.html>.
- ARCHIBALD, J. AND BAER, J.-L. 1986. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.* 4, 4, 273–298. (p. 10)
- BACH, M. J. 1986. *The Design of The Unix Operating System*. Prentice Hall. (p. 5)
- DOELLE, J. 2001. Re: [patch] align vm locks, new spinlock patch. Available: <http://marc.theaimsgroup.com/?l=linux-kernel&m=100107844605006&w=2> [Viewed December 29, 2005]. (p. 8)
- FLYNN, M. 1972. Some computer organizations and their effectiveness. *IEEE Transactions on Computers* 21, 9, 948–960. (p. 9)
- GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. 1999. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 87–100. Preprint Available: <http://www.research.ibm.com/K42/osdi-preprint.ps> [Viewed Dec 29, 2005]. (pp. 18, 50)
- GORMAN, M. 2004. *Understanding the Linux Virtual Memory Manager*. Prentice Hall Ptr. Available: <http://www.phptr.com/bookstore/product.asp?isbn=0131453483&rl=1>. (p. 21)
- HOWELLS, D. 2006. Linux kernel memory barriers. Available: in Documentation/memory-barriers.txt of the Linux source tree. (p. 11)
- LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* 9, 690–691. (p. 10)
- LITTLE, J. 1961. A proof of the queueing formula $l = \lambda w$. *Operations Research* 9, 383–387. (p. 30)
- MCKENNEY, P. 2006a. Linux rcu documentation. Available: in Documentation/RCU/ of the Linux source tree. (p. 20)
- MCKENNEY, P. 2006b. Re: [patch 3/3] radix-tree: Rcu lockless readside. Available: <http://marc.theaimsgroup.com/?l=linux-kernel&m=115081531627333&w=2>. (p. 73)
- MCKENNEY, P. E. 2005. Abstraction, reality checks, and rcu. In *Cider Seminar* (Toronto, Canada, July 2005). Available: <http://www.rdrop.com/users/paulmck/RCU/RCUintro.2005.07.26bt.pdf> [Viewed Dec 31, 2005]. (p. 12)

- MCKENNEY, P. E. AND SLINGWINE, J. D. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518. (pp. 18, 19)
- MOLLNAR, I. AND MILLER, D. 2002. Scalable pagecache. Available: <http://people.redhat.com/~mingo/smp-pagecache-patches/pagecache-2.5.3-A0.patch> [Viewed December 29, 2005]. (p. 15)
- MORAN, J. P., GINGELL, R. A., AND SHANNON, W. A. 1987. Virtual memory architecture in sunos. In *USENIX Conference Proceedings* (Phoenix, AZ, Summer 1987), pp. 81–94. Available: http://www.opensolaris.org/os/project/muskoka/doc_attic/vm-arch.pdf [Viewed Sep 15, 2006]. (p. 6)
- MOSBERGER, D. 1993. Memory consistency models. *Operating Systems Review* 27, 1, 18–26. Available: <ftp://ftp.cs.arizona.edu/reports/1993/TR93-11.ps.Z>. (p. 10)
- PIGGIN, N. 2006. Lockless pagecache. Available: <http://www.kernel.org/pub/linux/kernel/people/npiggin/patches/lockless/2.6.17/2.6.17-lockless.gz>. (pp. 55, 59)
- PRESSMAN, R. S. 2000. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education. (p. 75)
- RITCHIE, D. M. AND THOMPSON, K. 1978. The UNIX time-sharing system. *The Bell System Technical Journal* 57, 6 (part 2), 1905+. (p. 5)
- WOODACRE, M., ROBB, D., ROE, D., AND FEIND, K. 2003. *The SGI Altix 3000 Global Shared-Memory Architecture*. Available: <http://www.sgi.com/pdfs/3392.pdf>. (p. 35)
- WULF, W. A. AND MCKEE, S. A. 1995. Hitting the memory wall: Implications of the obvious. *Computer Architecture News* 23, 1, 20–24. Available: <ftp://ftp.cs.virginia.edu/pub/techreports/CS-94-48.ps.Z>. (p. 11)