Scaling the Linux VFS

Nick Piggin
SuSE Labs, Novell Inc.

September 19, 2009

Outline

I will cover the following areas:

- Introduce each of the scalability bottlenecks
- Describe common operations they protect
- Outline my approach to improving synchronisation
- Report progress, results, problems, future work

Goal

- Improve scalability of common vfs operations;
- with minimal impact on single threaded performance;
- and without an overly complex design.
- Single-sb scalability.

VFS overview

- Virtual FileSystem, or Virtual Filesystem Switch
- Entry point for filesystem operations (eg. syscalls)
- Delegates operations to appropriate mounted filesystems
- Caches things to reduce or eliminate fs responsibility
- Provides a library of functions to be used by fs

The contenders

- $files\_lock$
- $vfsmount\_lock$
- mnt_count
- $dcache\_lock$
- $inode\_lock$
- And several other write-heavy shared data

$files\_lock$

- Protects modification and walking a per-sb list of open files
- Also protects a per-tty list of files open for ttys
- $open(2), close(2)$ syscalls add and delete file from list
- remount,ro walks the list to check for RW open files

$files\_lock$ **ideas**

- We can move tty usage into its own private lock
- per-sb locks would help, but I want scalability within a single fs
- Fastpath is updates, slowpath is reading – RCU won't work.
- Modifying a single object (the list head) cannot be scalable:
- must reduce number of modifications (eg. batching),
- or split modifications to multiple objects.
- Slowpath reading the list is very rarely used!

$files\_lock$ my implementation

- This suggests per-CPU lists, protected by per-CPU locks.
- Slowpath can take all locks and walk all lists
- Pros: “perfect” scalability for file open/close, no extra atomics
- Cons: larger superblock struct, slow list walking on huge systems
- Cons: potential cross-CPU file removal

$vfsmount\_lock$

- Largely, protects reading and writing mount hash
- Lookup vfsmount hash for given mount point
- Publishing changes to mount hierarchy to the mount hash
- Mounting, unmounting filesystems modify the data
- Path walking across filesystem mounts reads the data

$vfsmount\_lock$ **ideas**

- Fastpath are lookups, slowpath updates
- RCU could help here, but there is a complex issue:
- Need to prevent umounts for a period after lookup (while we have a ref)
- Usual implementations have per-object lock, but per-sb scalability
- Umount could $synchronize\_rcu()$, this can sleep and be very slow

$vfsmount\_lock$ **my implementation**

- Per-cpu locks again, this time optimised for reading
- “brlock”, readers take per-cpu lock, writers take all locks
- Pros: “perfect” scalability for mount lookup, no extra atomics
- Cons: slower umounts

mnt_count

- A refcount on vfsmount, not quite a simple refcount
- Used importantly in open(2), close(2), and path walk over mounts

mnt_count **my implementation**

- Fastpath is get/put.
- A “put” must also check count==0, makes per-CPU counter hard
- However count==0 is always false when vfsmount is attached
- So only need to check for 0 when not mounted (rare case)
- Then per-CPU counters can be used, with per-CPU $vfsmount\_lock$
- Pros: “perfect” scalability for vfsmount refcounting
- Cons: larger vfsmount struct

$dcache\_lock$

- Most dcache operations require $dcache\_lock$.
- except name lookup, converted to RCU in 2.5
- dput last reference (except for “simple” filesystems)
- any fs namespace modification (create, delete, rename)
- any uncached namespace population (uncached path walks)
- dcache LRU scanning and reclaim
- socket open/close operations

dcache_lock **is hard**

- Code and semantics can be complex
- It is exported to filesystems and held over methods
- Hard to know what it protects in each instance it is taken
- Lots of places to audit and check
- Hard to verify result is correct
- This is why I need vfs experts and fs developers

$dcache\_lock$ **approach**

- identify what the lock protects in each place it is called
- implement new locking scheme to protect usage classes
- remove $dcache\_lock$
- improve scalability of (now simplified) classes of locks

dcache locking classes

- dcache hash
- dcache LRU list
- per-inode dentry list
- dentry children list
- dentry fields (d_count, d_flags, list membership)
- dentry refcount
- reverse path traversal
- dentry counters

dcache my implementation outline

- All dentry fields including list mebership protected by d_lock
- children list protected by d_lock (this is a dentry field too)
- dcache hash, LRU list, inode dentry list protected by new locks
- Lock ordering can be difficult, trylock helps
- Walking up multiple parents requires RCU and rename blocking. Hard!

dcache locking difficulties 1

- “Locking classes” not independent.

```
spin_lock(&dcache_lock);
list_add(&dentry->d_lru, &dentry_lru);
hlist_add(&dentry->d_hash, &hash_list);
spin_unlock(&dcache_lock);
```

is **not** the same as

```
spin_lock(&dcache_lru_lock);
list_add(&dentry->d_lru, &dentry_lru);
spin_unlock(&dcache_lru_lock);
spin_lock(&dcache_hash_lock);
hlist_add(&dentry->d_hash, &hash_list);
spin_unlock(&dcache_hash_lock);
```

Have to consider each $dcache\_lock$ site carefully, in context.

d_lock does help a lot.

dcache locking difficulties 2

- $EXPORT\_SYMBOL(dcache\_lock);$
- $->d\_delete$

Filesystems may use $dcache\_lock$ in non-trivial ways for protecting their own data structures and locking parts of dcache code from executing. Autofs4 seems to do this, for example.

dcache locking difficulties 3

- Reverse path walking (from child to parent)

We have dcache parent$-$ $>$child lock ordering. Walking the other way is tough. $dcache\_lock$ would freeze the state of the entire dcache tree. I use RCU to prevent parent from being freed while dropping the child's lock to take the parent lock. Rename lock or seqlock/retry logic can prevent renames causing our walk to become incorrect.

dcache scaling in my implementation

- dcache hash lock made per-bucket
- per-inode dentry list made per-inode
- dcache stats counters made per-CPU
- dcache LRU list is last global $dcache\_lock$, could be made per-zone
- pseudo filesystems don't attach dentries to global parent

dcache implementation complexity

- Lock ordering can be difficult
- Lack of a way to globally freeze the tree
- Otherwise in some ways it is actually simpler

$inode\_lock$

- Most inode operations require $inode\_lock$.
- Except dentry$->$inode lookup and refcounting
- Inode lookup, cached and uncached, inode creation and destruction
- Including socket, other pseudo-sb operations
- Inode dirtying, writeback, syncing
- icache LRU walking and reclaim
- socket open/close operations

$inode\_lock$ **approach**

- Same as approach for dcache

icache locking classes

- inode hash
- inode LRU list
- inode superblock inodes list
- inode dirty list
- inode fields (i_state, i_count, list membership)
- iunique
- $last\_ino$
- inode counters

icache implementation outline

- Largely similar to dcache
- All inode fields including list membership protected by i_lock
- icache hash, superblock list, LRU+dirty lists protected by new locks
- $last\_ino$, $iunique$ given private locks
- Not simple, but easier than dcache! (less complex and less code)

icache scaling my implementation

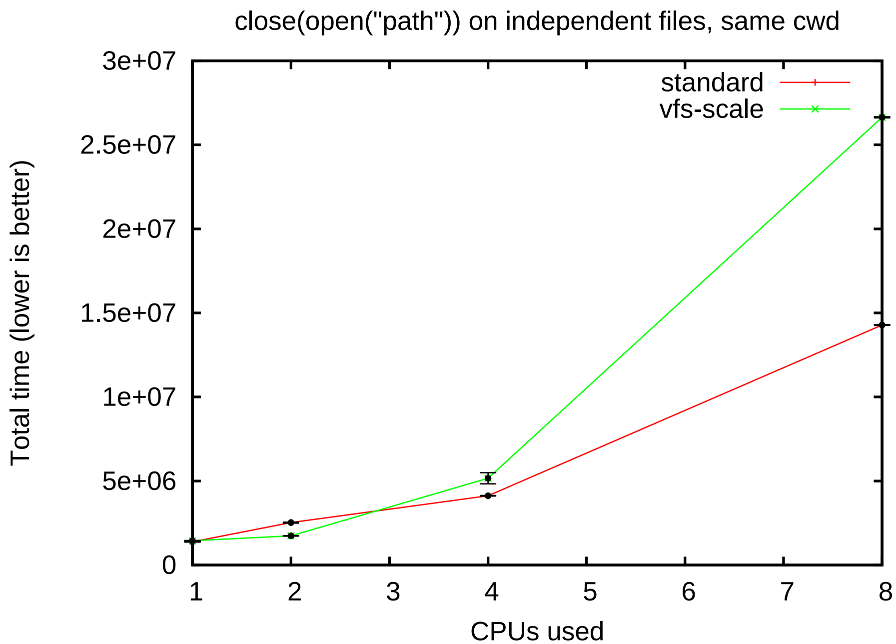
- inode made RCU freed to simplify lock orderings and reduce complexity
- icache hash lock made per-bucket, lockless lookup
- icache LRU list made lazy like dcache, could be made per-zone
- per-cpu, per-sb inode lists
- per-cpu inode counter
- per-cpu inode number allocator (Eric Dumazet)
- inode and dirty list remains problematic.

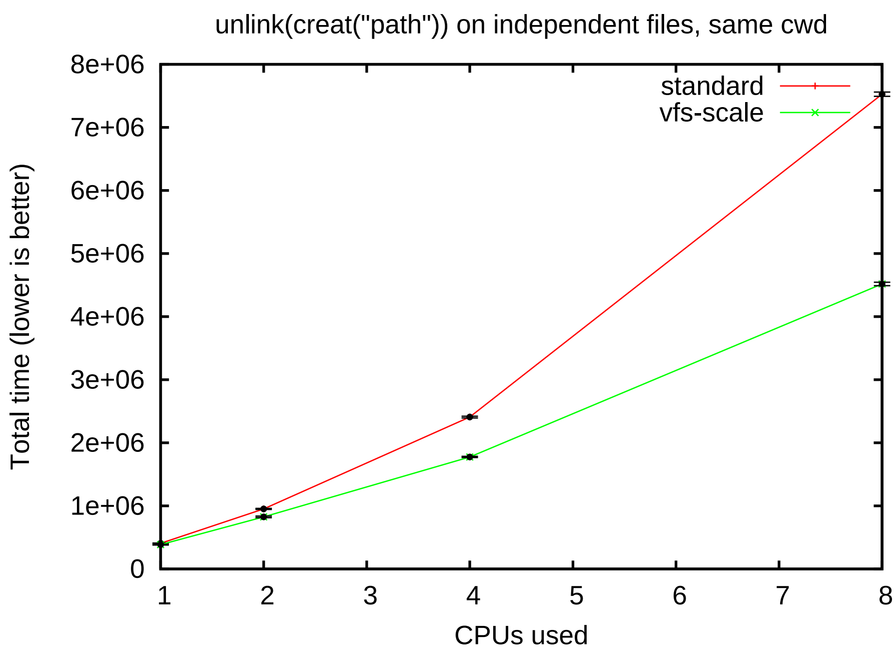
Current progress

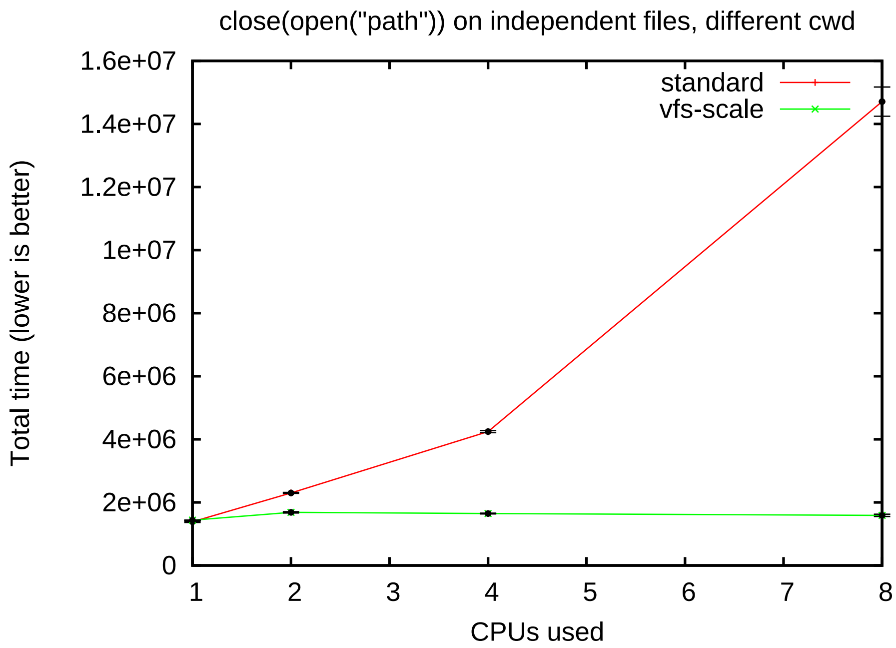
- Very few fundamentally global cachelines remain
- I'm using tmpfs, ramfs, ext2/3, nfs, nfsd, autofs4.
- Most others require some work
- Particularly dcache changes not audited in all filesystems
- Still stamping out bugs, doing some basic performance testing
- Still working to improve single threaded performance

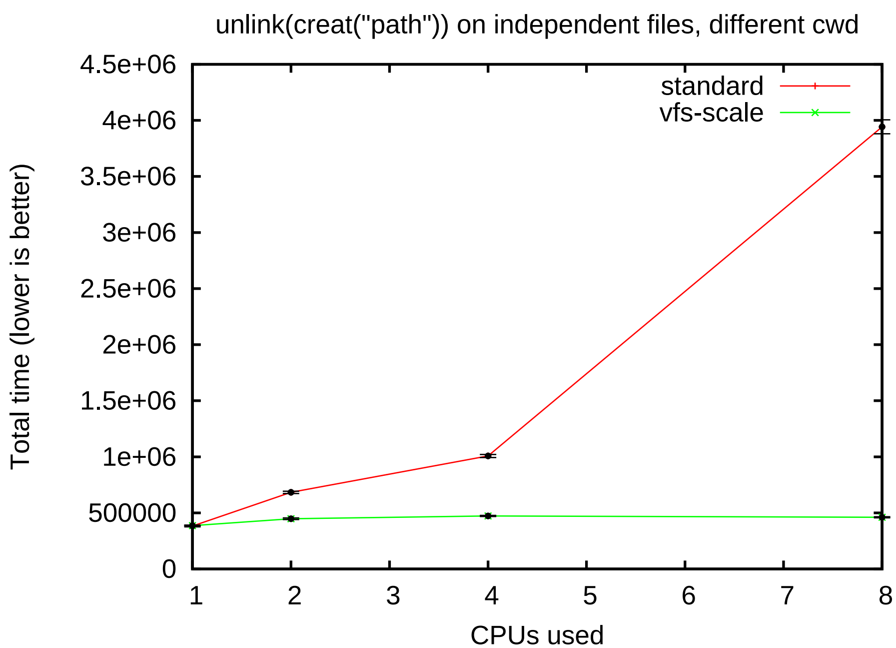
Performance results

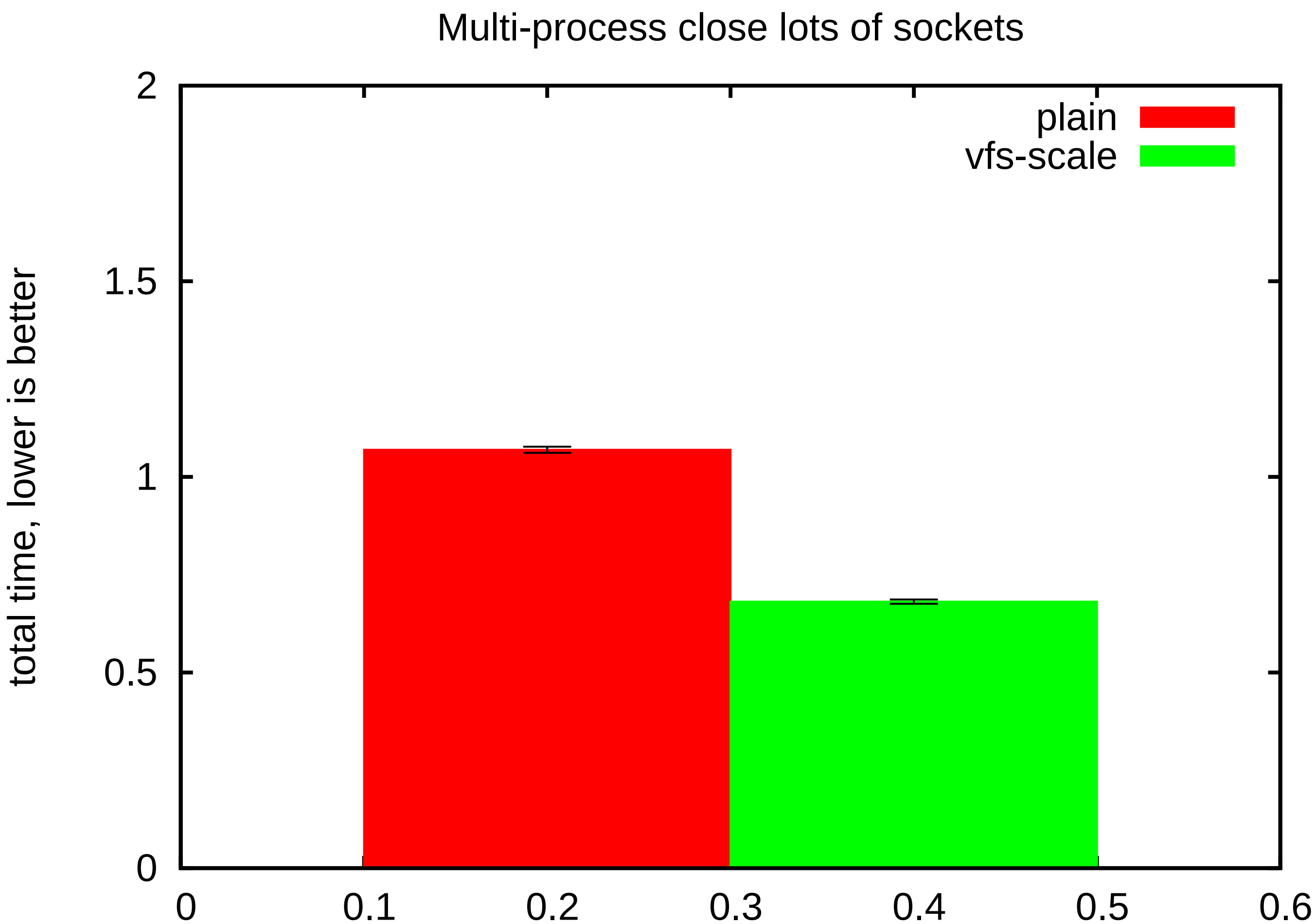
- The abstract was a lie!
- open(2)/close(2) in seperate subdirs seems perfectly scalable
- creat(2)/unlink(2) seems perfectly scalable
- Path lookup less scalable with common cwd, due to d_lock in refcount
- Single-threaded performance is worse in some cases, better in others

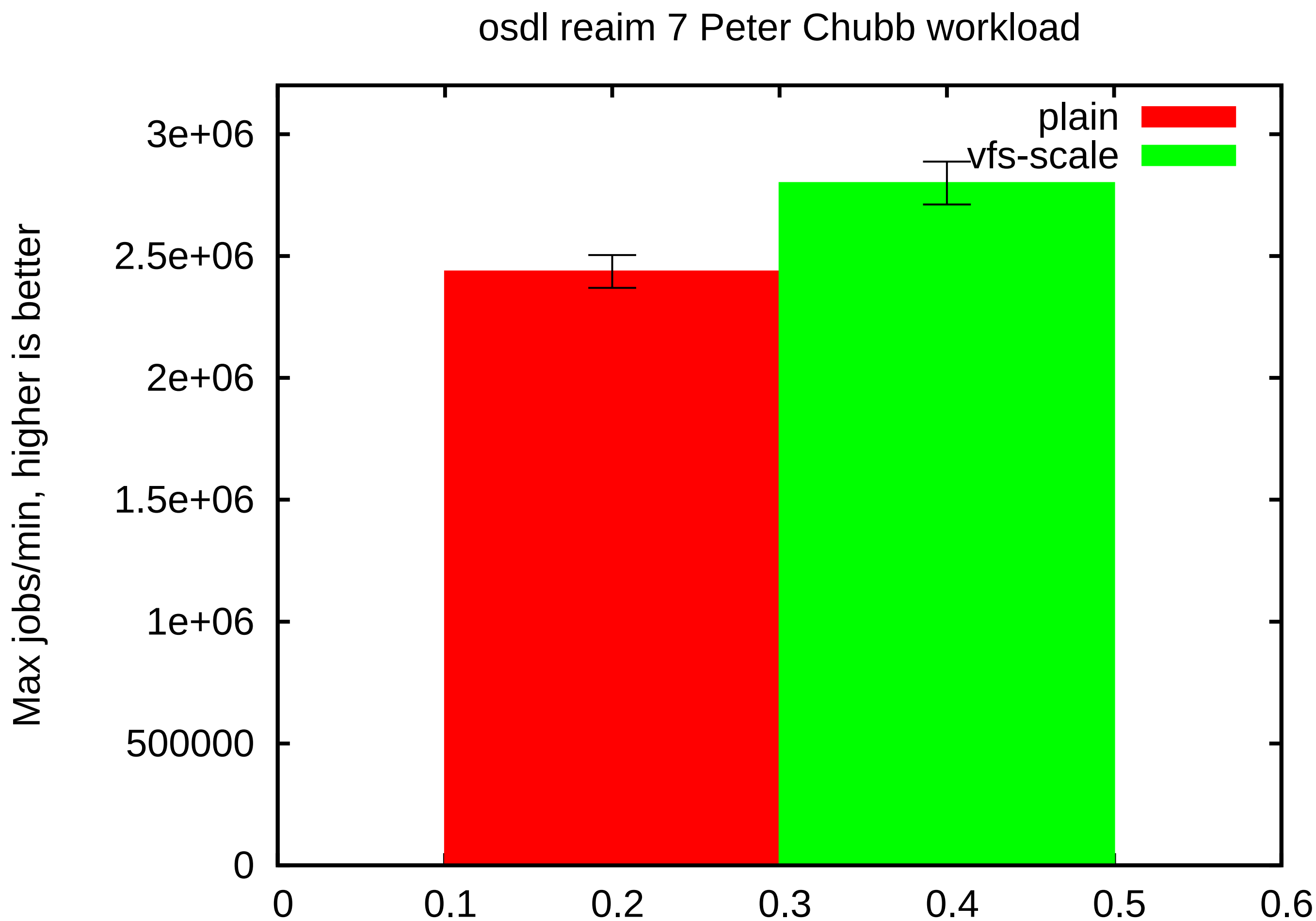

close(open("path")) on independent files, same cwd
Total time (lower is better)
CPUs used
standard
vfs-scale
0
5e+06
1e+07
1.5e+07
2e+07
2.5e+07
3e+07
1
2
3
4
5
6
7
8


unlink(creat("path")) on independent files, same cwd
Total time (lower is better)
CPUs used
standard
vfs-scale
0
1e+06
2e+06
3e+06
4e+06
5e+06
6e+06
7e+06
8e+06
1
2
3
4
5
6
7
8


close(open("path")) on independent files, different cwd
Total time (lower is better)
CPUs used
standard
vfs-scale
0
2e+06
4e+06
6e+06
8e+06
1e+07
1.2e+07
1.4e+07
1.6e+07
1
2
3
4
5
6
7
8


unlink(creat("path")) on independent files, different cwd
Total time (lower is better)
CPUs used
standard
vfs-scale
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
4.5e+06
1
2
3
4
5
6
7
8


Multi-process close lots of sockets
plain
vfs-scale
total time, lower is better
2
1.5
1
0.5
0
0
0.1
0.2
0.3
0.4
0.5
0.6


osdl reaim 7 Peter Chubb workload
Max jobs/min, higher is better
plain
vfs-scale
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
0
0.1
0.2
0.3
0.4
0.5
0.6

Future work

- Improve scalability (eg. LRU lists, inode dirty list)
- Look at single threaded performance, code simplifications

Interesting future possibilities:

- Path walk without taking d_lock
- Paves the way for NUMA aware dcache/icache reclaim
- Can expand the choice of data structure (simplicity, RCU requirement)

How can you help

- Review code
- Audit filesystems
- Suggest alternative approaches to scalability
- Implement improvements, “future work”, etc
- Test your workload

Conclusion

VFS is hard. That’s the only thing I can conclude so far.

Thank you