Scaling the Linux VFS

Nick Piggin
SuSE Labs, Novell Inc.

October 16, 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.

Scalability basics

- Scalability problem when there is contention for shared resources.
- Seen in software – eg. locks, where only one code path may proceed.
- Or hardware – eg. cachelines, where only one CPU may modify a line.
- Must avoid exclusive locks and writing to shared variables,
- achieved by changing locking techniques and/or data structures.

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

VFS overview 2

- Every mounted filesystem has a $struct\ super\_block$.
- May be mounted more than once, each mount point has a $struct\ vfsmount$.
- File data is manipulated with $struct\ inode$, also a cache.
- Directory entry layout manipulated with $struct\ dentry$, also a cache.
- Complex and inter-related data structures.

The major problems

- $files\_lock$
- $vfsmount\_lock$
- mnt_count
- $dcache\_lock$
- $inode\_lock$
- Several other write-often data (eg. counters)
- d_lock on common dentries

$files\_lock$

- Every open file is put on a per-superblock list.
- Global $files\_lock$ spinlock protects access to this list.
- Each $open(2)$ and $close(2)$ syscall takes this lock.
- Global limit on $open(2)$ and $close(2)$ scalability.

Scaling $files\_lock$

- Slowpath reading the list is very rare, fastpath is updates.
- Modifying a single object (the list head) cannot be scalable:
- must reduce number of modifications (eg. batching),
- or split modifications to multiple objects.
- I use per-CPU lists, protected by per-CPU locks.
- Potential cross-CPU file removal issue.

$vfsmount\_lock$

- Every mounted filesystem
- Largely, protects reading and writing mount hash.
- Path lookup of directory with fs mounted result in a mount lookup.
- Mount lookup searches the vfsmount hash for given mount point.
- Mounting, unmounting filesystems modifies the vfsmount hash.
- Global limit on path lookups over mount points.

Scaling $vfsmount\_lock$

- Fastpath is lookups, slowpaths are updates
- RCU not trivial, we must keep lookups away while unmounting,
- but per-vfsmount lock defeats single-sb scalability,
- and $synchronize\_rcu()$ is just too slow even for umount.
- Use per-cpu locks again, this time optimised for reading
- “brlock”, readers take a per-cpu lock, writers take all locks

mnt_count

- A refcount on vfsmount, not quite a simple refcount.
- Used importantly in open(2), close(2), and path walk over mounts.
- Per-mount limit on $open(2)$ and $close(2)$ scalability.

Scaling mnt_count

- Fastpath is get/put (increment and decrement-and-test refcount).
- "put" must check $count == 0$, making per-CPU counter hard.
- However $count == 0$ is always false when vfsmount is attached.
- So only check when not mounted (rare case), otherwise just decrement.
- Then per-CPU counters can be used, with per-CPU $vfsmount\_lock$.

$dcache\_lock$

- Most dcache operations require $dcache\_lock$.
- Name lookup is an exception, converted to RCU in 2.5.
- $dput$ last reference (common in $open(2)$/$close(2)$ lifecycle).
- Any namespace modification (file create, delete, rename).
- Any uncached namespace population (uncached path lookup).
- dcache LRU scanning and reclaim.
- Pipe and socket open/close (these create and delete dentries).

dcache locking classes

- $dcache\_lock$ protects several semi-independent cases:
- dcache hash,
- dcache LRU list,
- inode's dentry alias list,
- dentry's children list,
- dentry's parent,
- membership on lists (hash, LRU, parent's children, etc),
- dentry statistics counters.

Scaling $dcache\_lock$

- Use per-dentry lock to protect all dentry properties.
- Protect dentry children with d_lock too.
- dcache hash, LRU list, inode dentry list protected by new locking.
- Lock ordering can be difficult, trylock helps.

dcache locking classes difficulties

- “Locking classes” are not independent.

```
1: spin_lock(&dcache_lock);
2: list_add(&dentry->d_lru, &dentry_lru);
3: hlist_add(&dentry->d_hash, &hash_list);
4: spin_unlock(&dcache_lock);
```

is **not** the same as

```
1: spin_lock(&dcache_lru_lock);
2: list_add(&dentry->d_lru, &dentry_lru);
3: spin_unlock(&dcache_lru_lock);
4: spin_lock(&dcache_hash_lock);
5: hlist_add(&dentry->d_hash, &hash_list);
6: spin_unlock(&dcache_hash_lock);
```

- Multiple locks have ordering constraints. Trylock helps.

Scaling $dcache\_lock$ cont.

- dcache hash locking are per hash bucket.
- inode's dentry list is locked with the inode's lock.
- dcache statistics counters are using per-CPU counters.
- dcache LRU list protected with a global lock, could be made per-zone.

Scaling $dcache\_lock$ cont.

- Reverse path walking (from child to parent)

We have dcache parent$-$ $>$child lock ordering. Walking the other way is tough. Finding the path from a dentry to root would previously take $dcache\_lock$ to 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. This is similar to normal RCU lookups.

$inode\_lock$

- Most inode operations require $inode\_lock$.
- Except importantly dentry$-$ $>$inode lookup (read(2), write(2) etc.)
- inode lookup (uncached file open, file create, and nfsd)
- inode creation, and inode destruction (create, unlink)
- inode dirtying, writeback, syncing
- inode LRU walking and reclaim
- pipe and socket open/close (these create and delete inodes).

inode locking classes

- $inode\_lock$ protects several semi-independent cases:
- inode hash
- inode LRU list
- inode superblock inodes list
- inode dirty list
- membership on lists (hash, LRU, dirty, etc),
- inode fields (i_state)
- iunique
- $last\_ino$
- inode counters

Scaling $inode\_lock$

- Similar to approach to scale $dcache\_lock$
- Use per-inode lock to protect all inode properties.
- inode hash, superblock list, LRU, dirty lists protected by new locking.
- $last\_ino$, $iunique$ protected by new locking.

Scaling $inode\_lock$ cont.

- RCU free $struct\ inode$ to reduce locking, simplify lock order.
- inode hash locks are per hash bucket
- inode hash lookups are lock-free with RCU
- icache LRU list made lazy like dcache
- per-cpu inode superblock lists, per-cpu locking like $files\_lock$.
- inode statistics counters are using per-CPU counters.
- per-cpu inode number allocator (Eric Dumazet)
- inode LRU list has global lock, could be made per-zone.
- inode dirty list has a global lock, could be made per-superblock.

d_lock

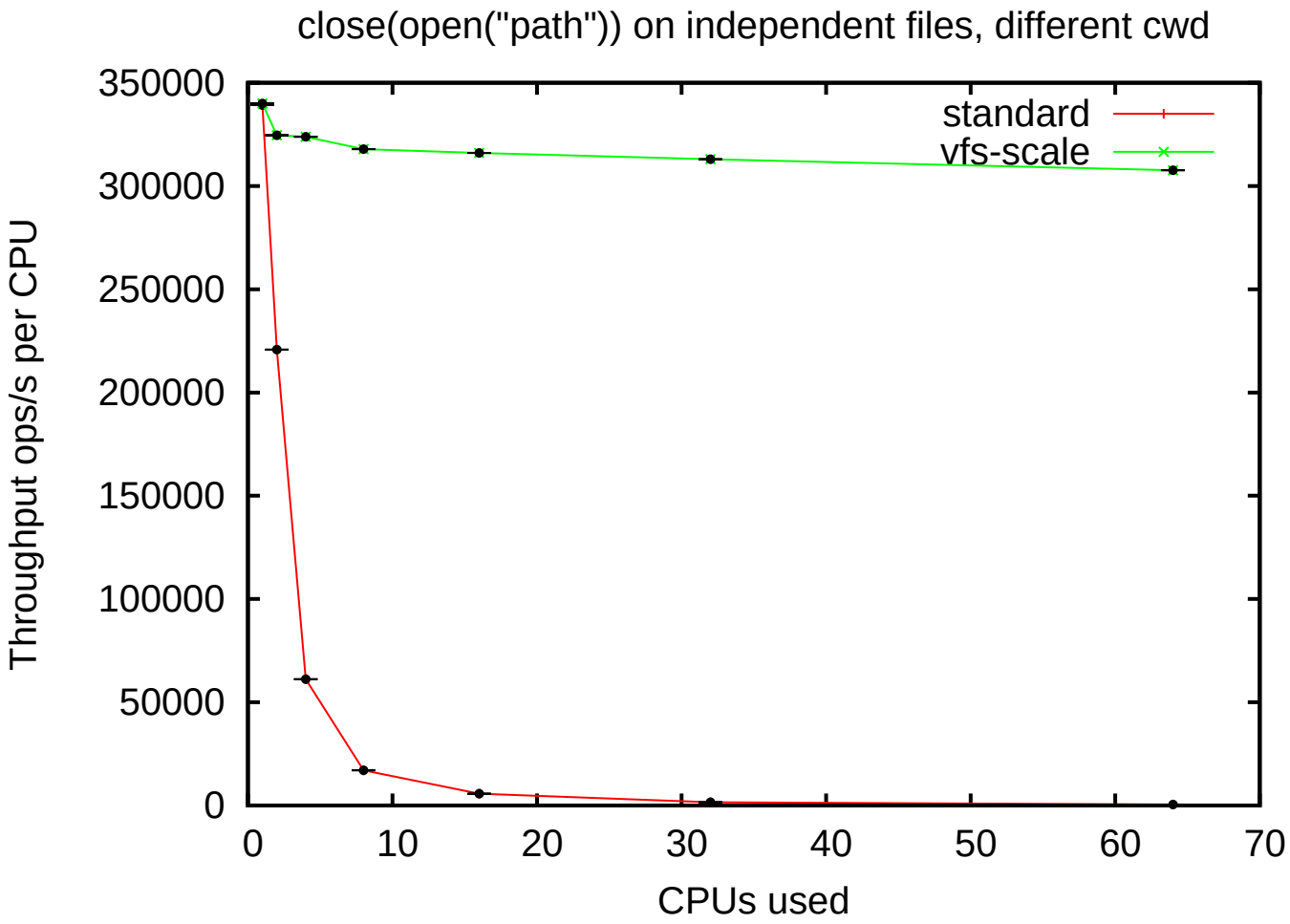
- Path lookup looks up each path element in turn, from dcache.
- For each dentry lookup, d_lock is taken and refcount is taken.
- These 3 atomic operations per path element are costly.
- Scalability problem for parallel lookup of common path elements.
- eg. root dentry or cwd dentry can be effectively a global lock.

Scaling d_lock

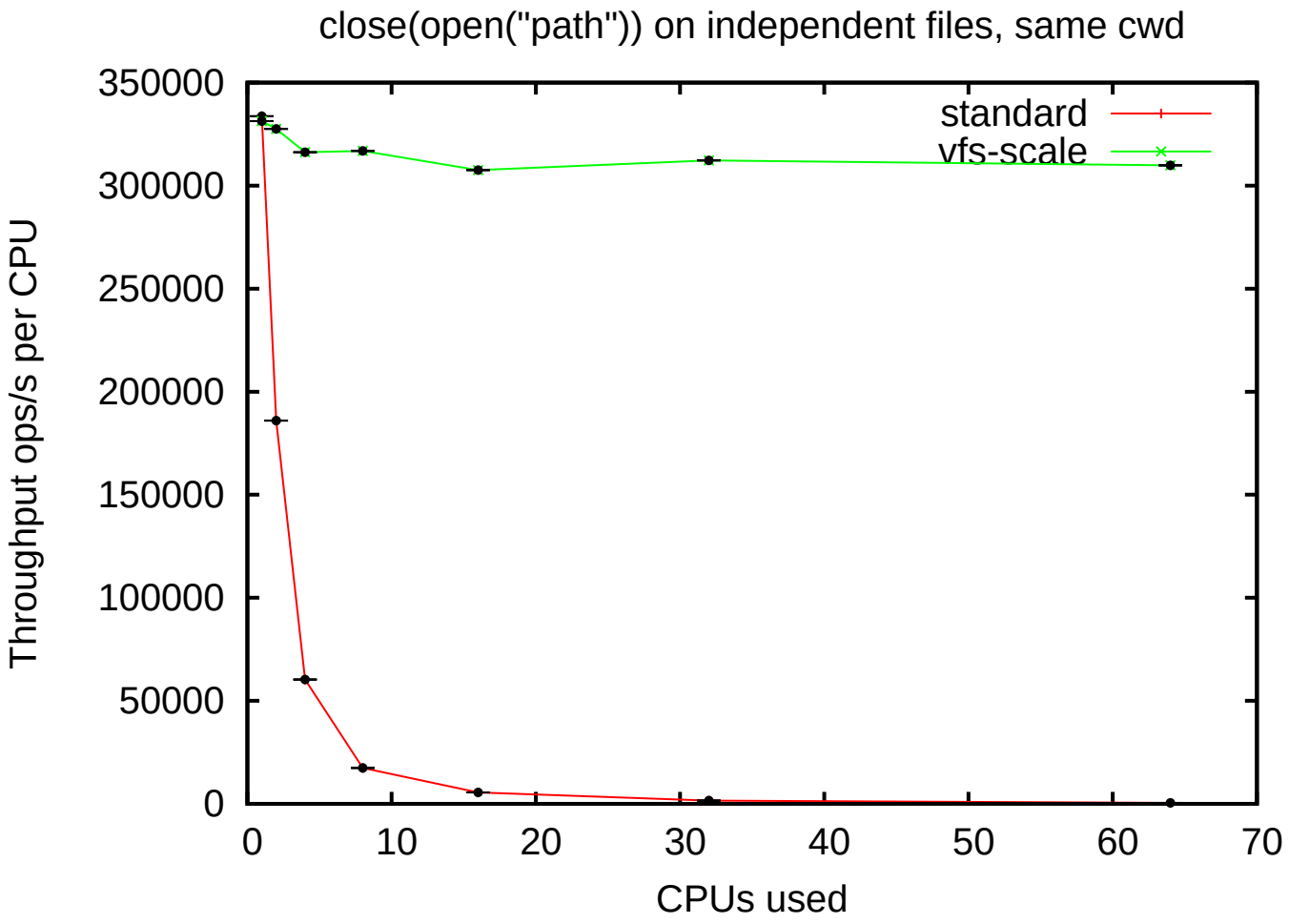
- Dcache lookup is already largely RCU.
- Locking and refcounting required for:
- blocking rename during name comparison,
- ensuring persistence of dentry and dentry's inode.
- inode now RCU freed, rcu_read_lock ensures inode persistence.
- seqlock can be used for atomic name comparison versus rename.
- Difficult cases (eg. fs call required), use old d_lock walk.

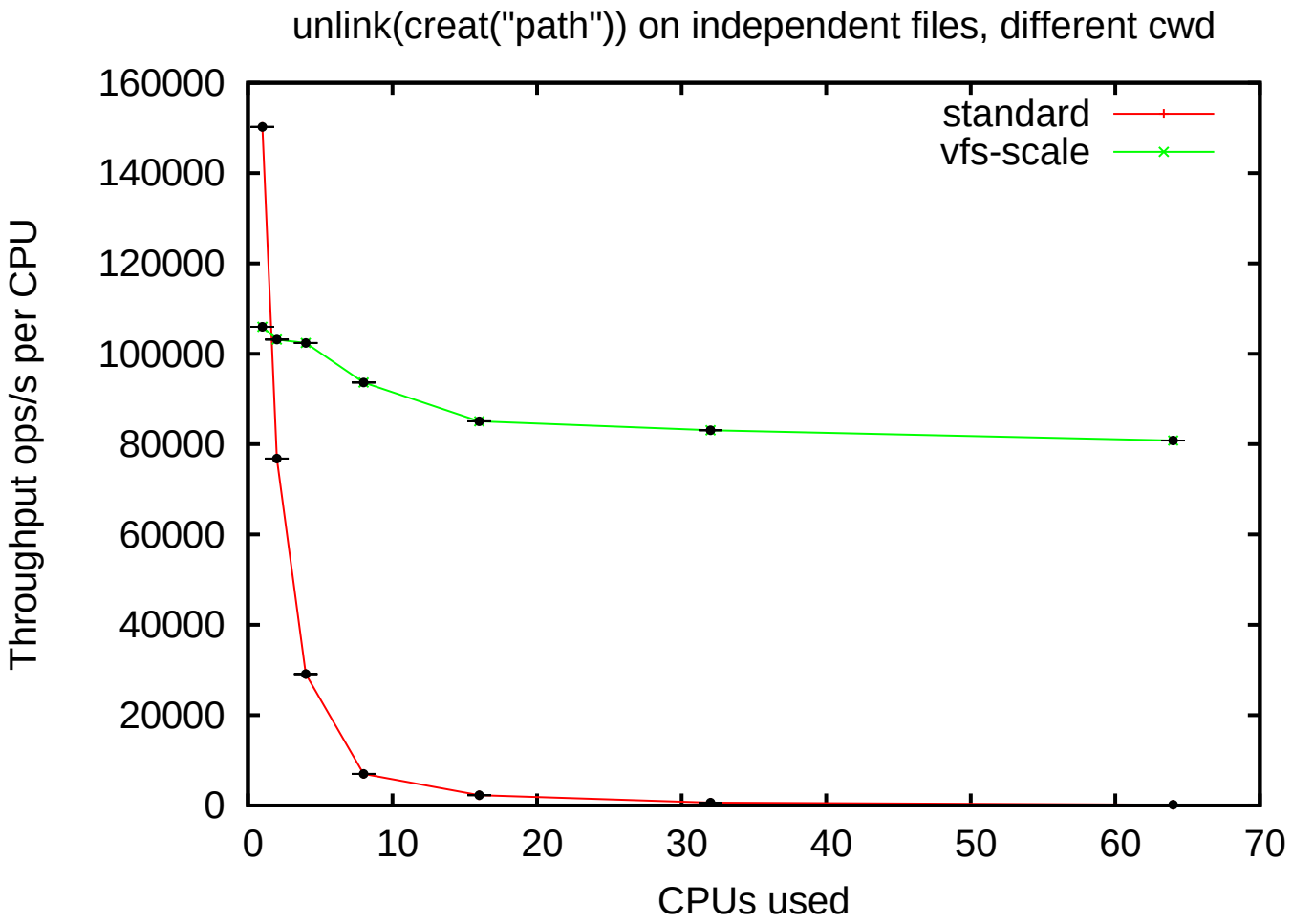
Performance results

- open(2)/close(2) seems perfectly scalable with lock free lookups.
- creat(2)/unlink(2) is very scalable, in seperate directories.
- Single-threaded performance is worse in some cases, better in others.
- Other benefits – eg. dentry/inode reclaim global lock.

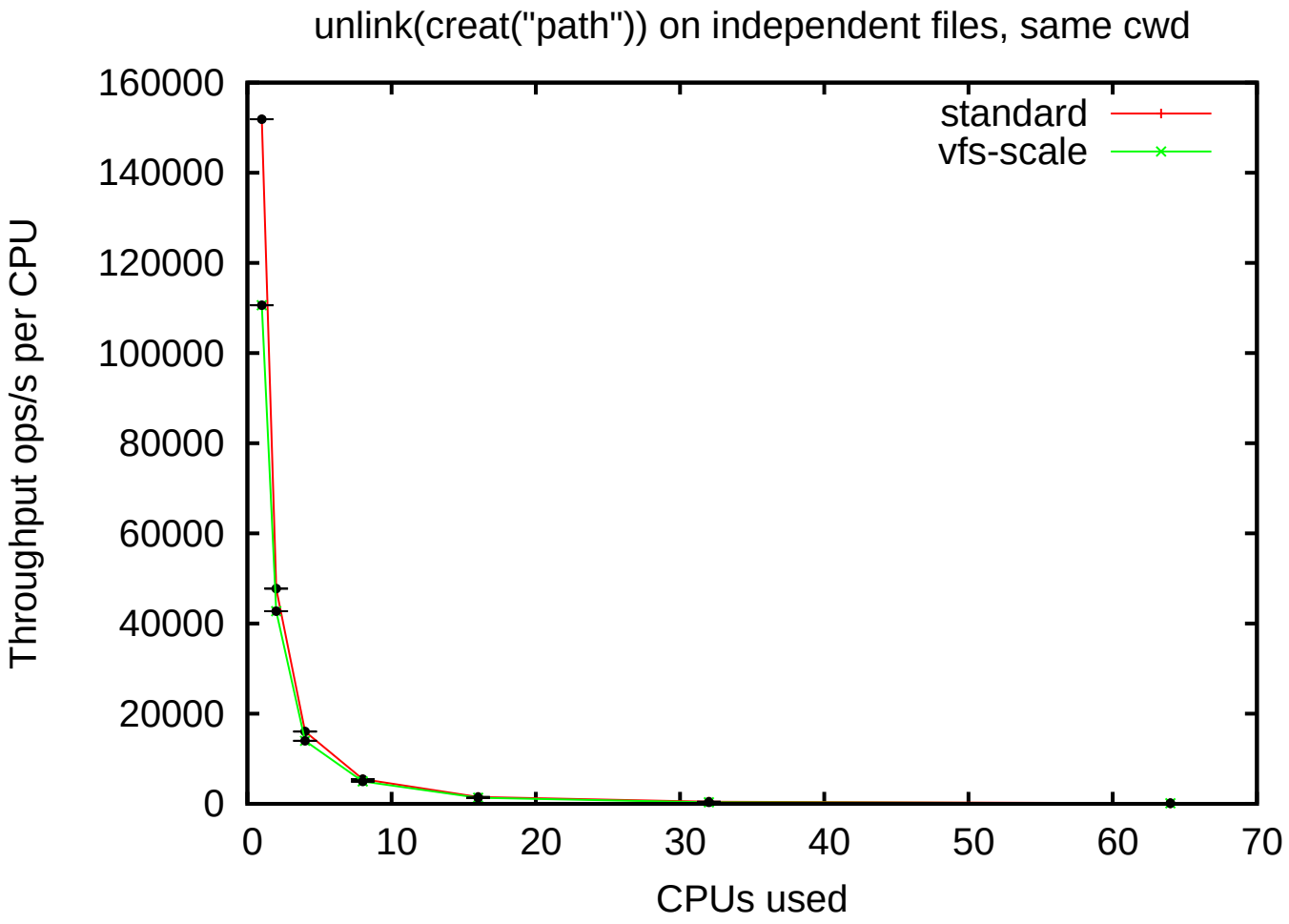


Plain kernel 450 ops/s per CPU at 64 CPUs

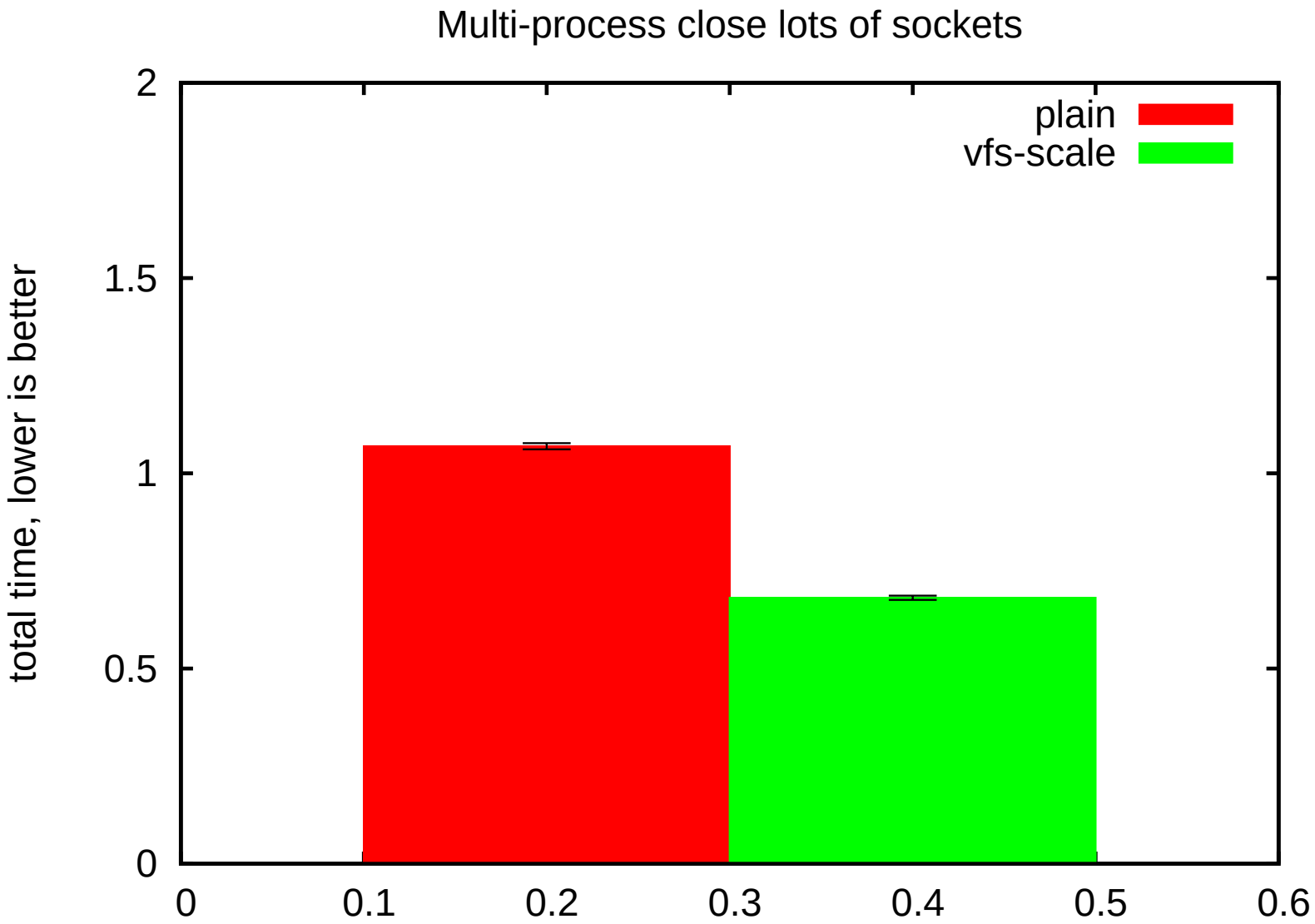

close(open("path")) on independent files, same cwd
Throughput ops/s per CPU
CPUs used
standard
vfs-scale
350000
300000
250000
200000
150000
100000
50000
0
0
10
20
30
40
50
60
70

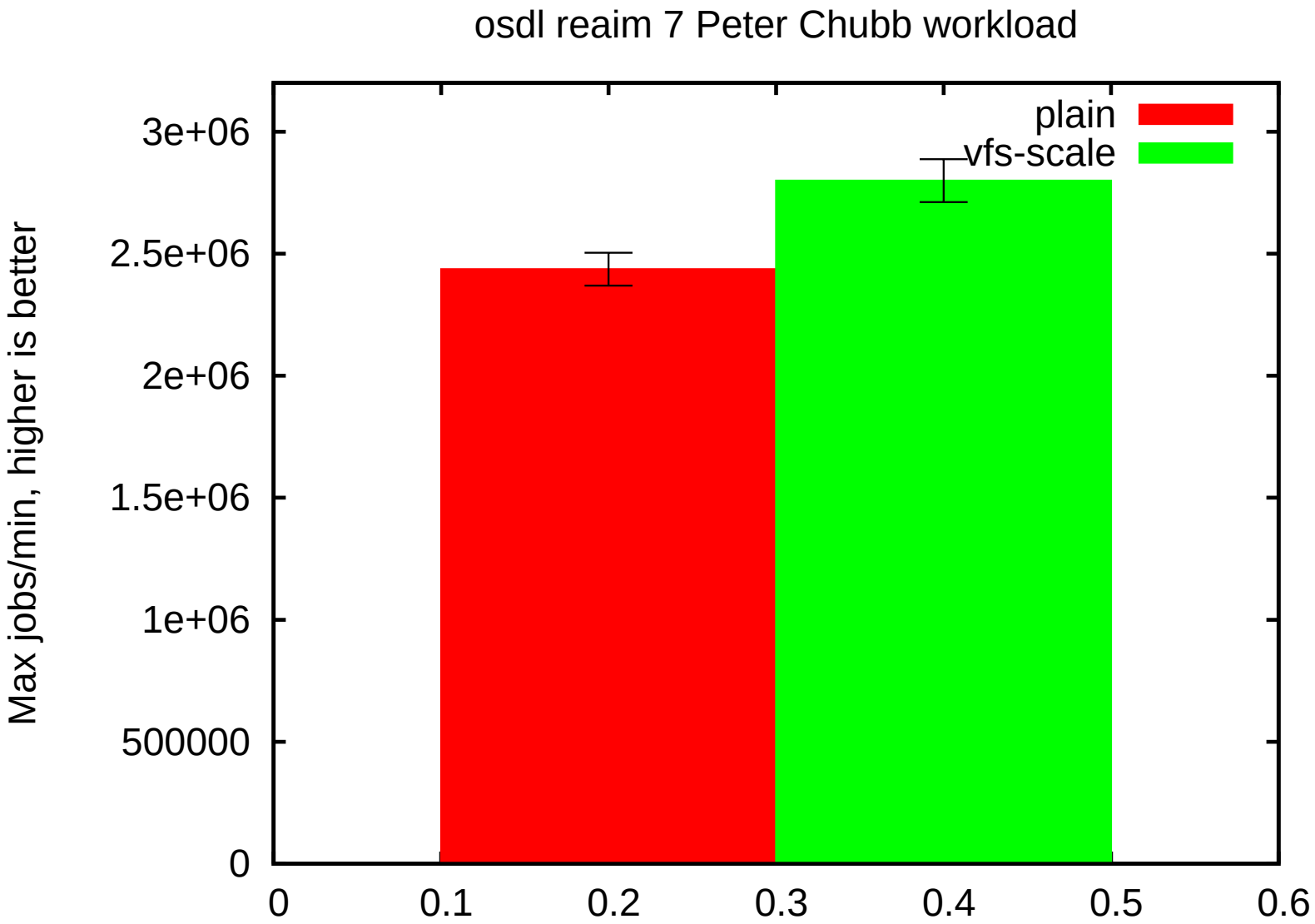


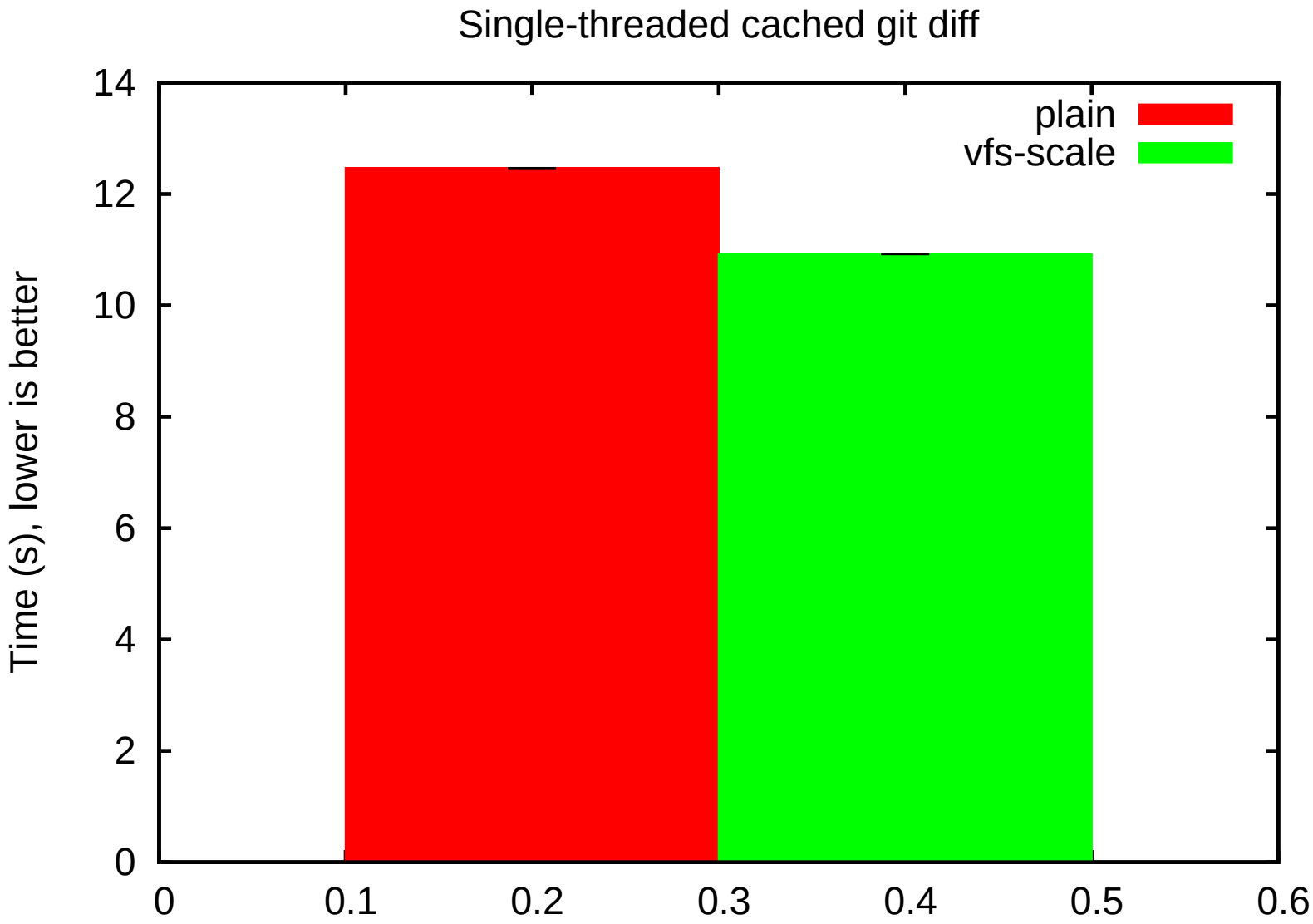
Plain kernel 140 ops/s per CPU at 64 CPUs

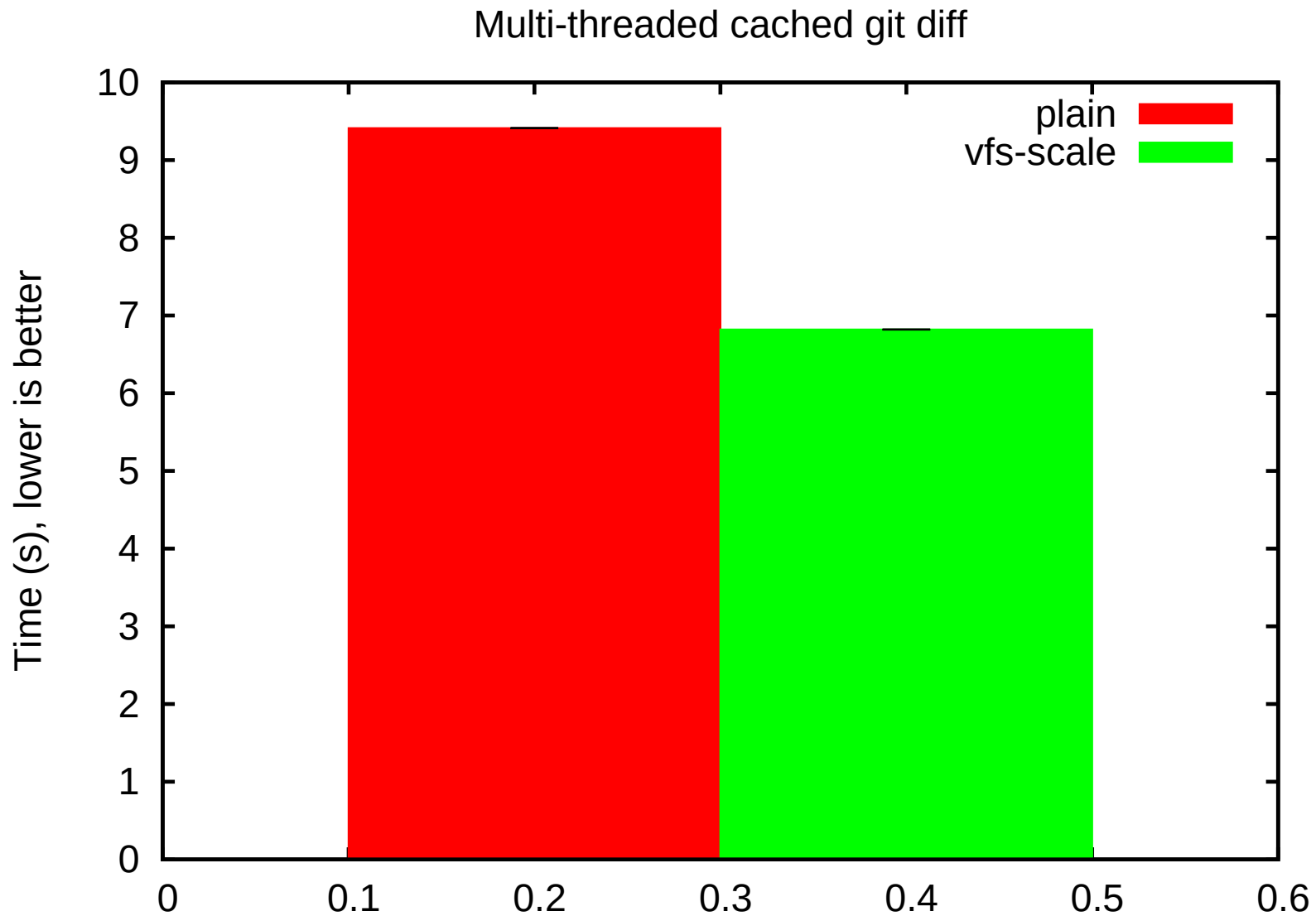


vfs patches give lower single-CPU performance


Multi-process close lots of sockets
total time, lower is better
plain
vfs-scale
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
3e+06
2.5e+06
2e+06
1.5e+06
1e+06
500000
0
0
0.1
0.2
0.3
0.4
0.5
0.6


Single-threaded cached git diff
Time (s), lower is better
plain
vfs-scale
0
2
4
6
8
10
12
14
0
0.1
0.2
0.3
0.4
0.5
0.6


Multi-threaded cached git diff
Time (s), lower is better
plain
vfs-scale
0
1
2
3
4
5
6
7
8
9
10
0
0.1
0.2
0.3
0.4
0.5
0.6

Current progress

- Very few fundamentally global cachelines remain.
- I'm using tmpfs, ramfs, ext2/3, nfs, nfsd, autofs4.
- Particularly dcache changes not audited in all filesystems.
- Still stamping out bugs, doing basic performance testing.

Future work

- Look at single threaded performance, code simplifications

Interesting future possibilities:

- Add more cases that lock free path walk can handle.
- Further improve scalability (eg. LRU lists, inode dirty list).
- This work paves the way for NUMA aware dcache/icache reclaim.
- Re-evaluate data structures (eg. trees instead of hash for lookup).

Conclusion

- VFS has scalability weak points.
- CPU core and thread count continues to increase.
- So the need to improve scalability is probably inevitable.
- I have developed reasonable ways to improve scalability.
- I am very interested in feedback, testing, alternative ideas.

Thank you