ORACLE®

Lustre SMP Scaling Improvements

Liang Zhen
Lustre Group

Agenda

- Where we started from
- Why we need this project
- The problems
- What we can improve
- Current status

Where we started from

- Initial goal of this project
  - Soft lockup of LNet on client side
    - RDMA Portal can have very long buffer list (hundreds even thousands of match-entries on the list, need to compare one by one)
- Survey on low-end 4-core machines
  - LNet has one single global spinlock to protect everything
  - Lockmeter shows extremely high contention on the global lock while running insanity network test (lnet_selftest)
    - 40+% UTIL (fraction of time that the lock was held during the report interval)
    - 60% CON (fraction of lock requests that found the lock was busy when it was requested)
  - RPC rate is not good enough - it’s CPU bound!

Why we need this project

- With more powerful CPU, is metadata performance improving?
  - Unfortunately… ☹
- Metadata performance is not disk bound
  - We have tested with ramdisk
  - Profiles show that performance is CPU-bound on scaling tests, especially on metadata stack
- Stability of Lustre
  - Not all soft lockup is a real BUG, it's probably just bad implementation
- I/O performance on NUMA systems

Why we need this project

Our objectives

- Make metadata performance faster
  - Unlock potential of higher IOPS from Flash/SSD
- Better I/O performance on NUMA systems
- Take advantage of rate of innovation in commodity microprocessor technology so our Lustre storage products can keep pace
- Less pressure on CMD ☺

The problems

LNet is the clue

- No heavy operation by LNet itself after we resolve the long ME-list (Match Entry) issue
  - List (search/change) operations, assignments, simple calculations
- Splitting the global lock by logic-path and making some cacheline optimizations…
  - 4 cores: better performance, Lockmeter: 4% UTIL, 15% CON
  - 8+ cores: barely better. It’s still a disaster while running insanity network test like lnet_selftest

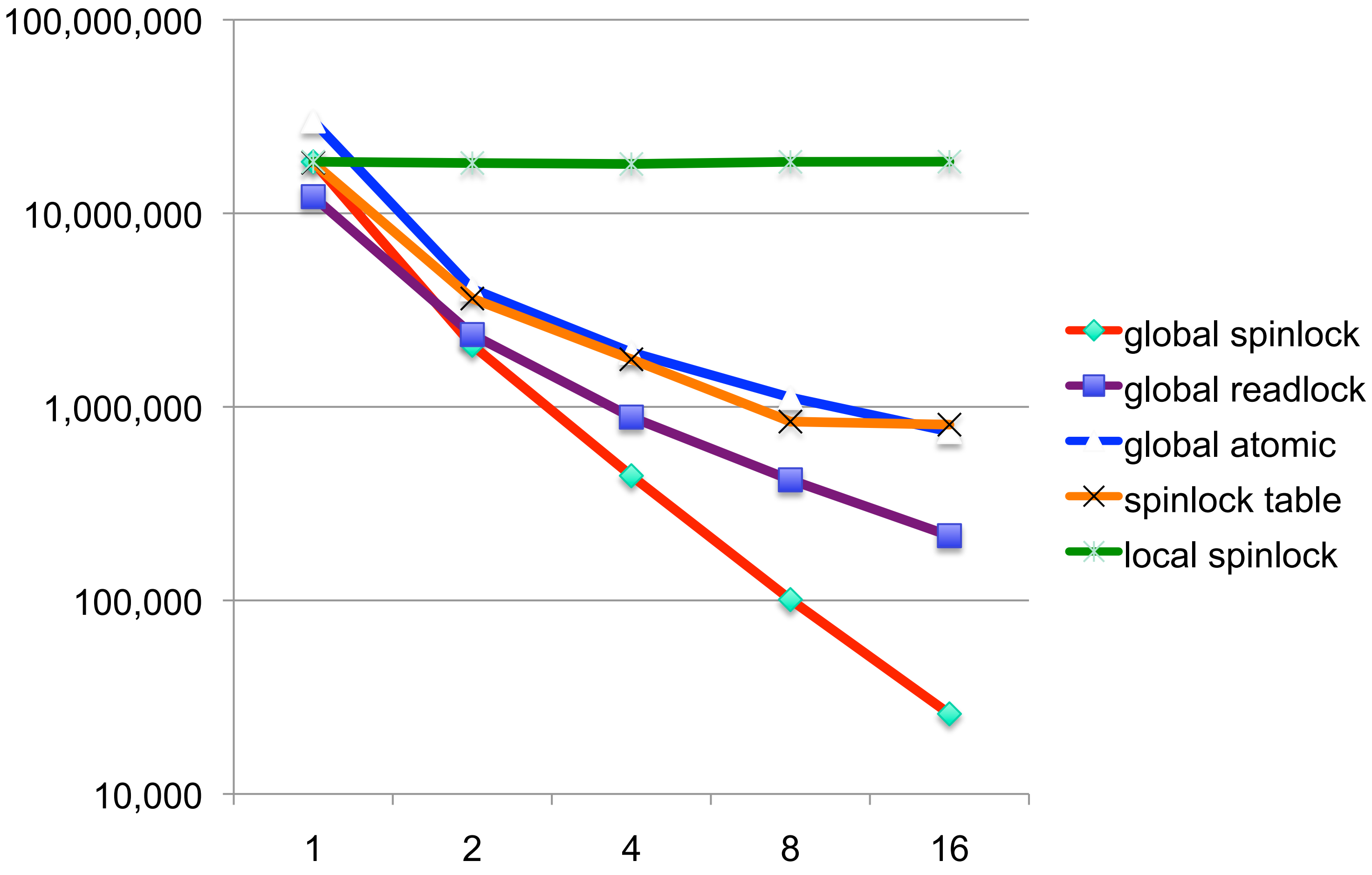
The problems

Overhead of synchronization (1 of 2)

- Memory speeds can’t catch up with CPU speeds
- Synchronization requires consistent view of data across CPUs, so synchronization is much much slower than normal instructions because of memory latency
- Huge amount of data traffic for synchronizations
- We tried to make critical section faster, but critical section efficiency is bad
  - Ta (lock acquisition), Tc (Critical section), Tr (lock release)
  - Efficiency = Tc / (Ta + Tc + Tr)

The problems

Overhead of synchronization (2 of 2)



The problems

All globals are hurting us

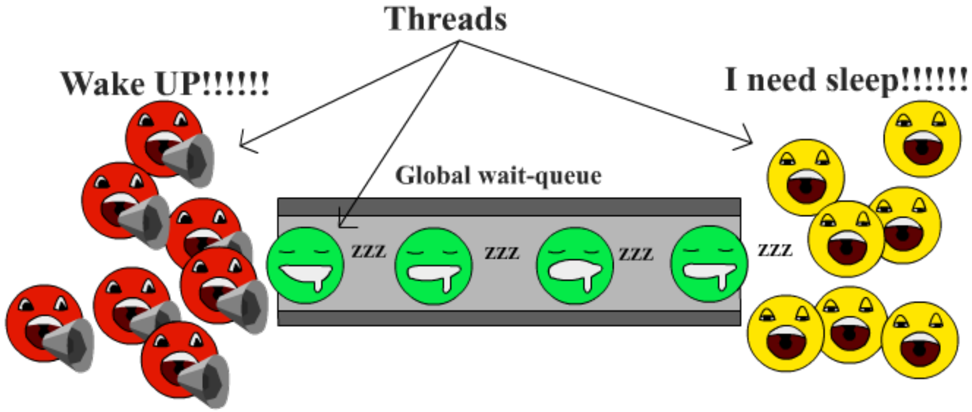
- Global locks are everywhere
  - Simple, but really bad
- Global stats, global refcount
  - Huge amount of data traffic between CPUs
- Few people care about cacheline conflicts
  - A simple code sample

```
struct foobar {
    spinlock_t        locka;
    Int               a;
    Spinlock_t        lockb;
    Int               b;
  }
```

The problems

Non-CPU affinity threads pool

- Most LND threads and ptlrpc service threads are not CPU affinity
  - Threads are scheduled by different CPUs, all data need to be taken to local cache of CPUs again and again
- Global waitq
  - Contention on waitq (sleep / wakeup)
  - Round robin wakeup, refresh cache again and again



The problems

Hash tables & Misc

- We are not careful enough about our hash tables
    - The two biggest hash tables are not well-hashed
        - Object hash
        - Ldlm hash
    - We have a hash table implementation for general purposes which is used everywhere, however…
        - Not good enough, a lot of unnecessary addref / decref, they are expensive atomic operations most of the time
        - Soft lockup
- Misc
    - Over-protected logic
    - LASSERT on very expensive conditions

What we can improve

libcfs infrastructure (1 of 2)

- CPU abstraction
  - CPU-node of libcfs can be (1-N) physical core, or NUMA node
- New interfaces for NUMA allocator
  - Local memory for each node, not only for MDT stack, also helpful for OST stack
- New interfaces for per-CPU data allocator
- New interfaces for cacheline aligned allocator
- LIFO wait-queue
  - Instead of FIFO wait-queue

What we can improve

libcfs infrastructure (2 of 2)

- Scalable local-global lock
  - Very fast local change
  - Slow global change
- A better implementation of cfs_hash
  - More flexible APIs
  - Different refcount modes and more efficient find-add
  - Much less addref/decref
  - Much SMP safer rehash & iteration

What we can improve

Restructured LNet & LND

- Each CPU has its local entry for LNet
- Each CPU has its own buffers (ME & MD list)
  - Requests are received on local buffer
  - Lazy portal is more important now
- EQ (Event Queue) improvements
  - EQ callback can happen concurrently on different CPUs
  - EQ has per-CPU refcount
- CPU affinity LND threads
  - Connections are hashed by NIDs
  - Each CPU has its own peer table
  - Completion vector of OFED

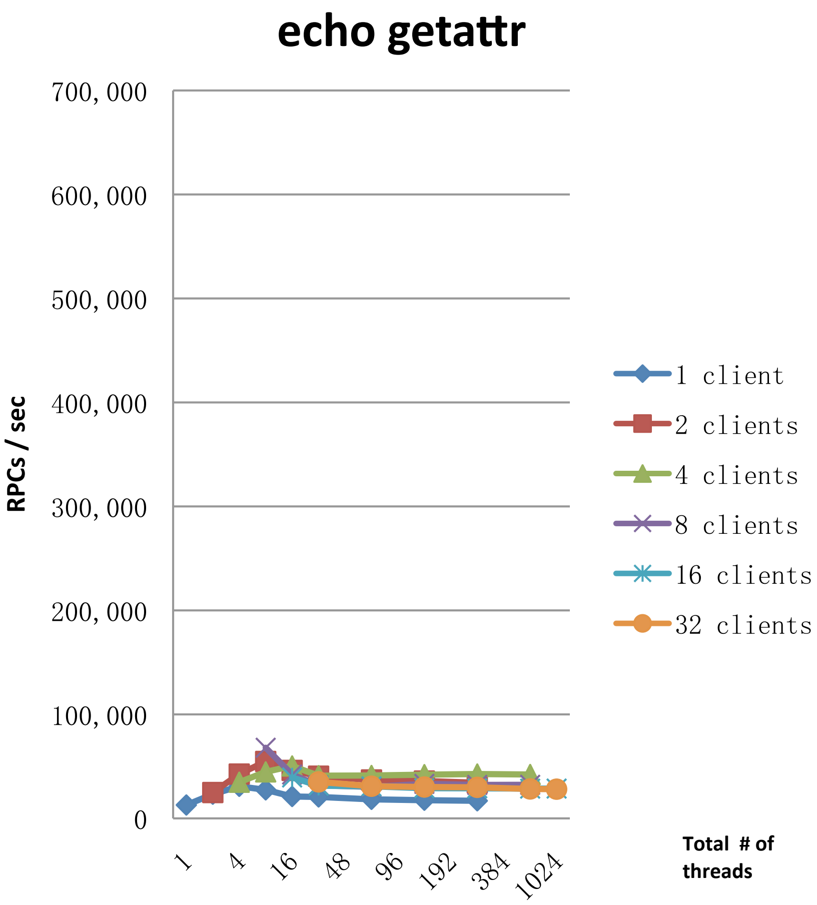
What we can improve

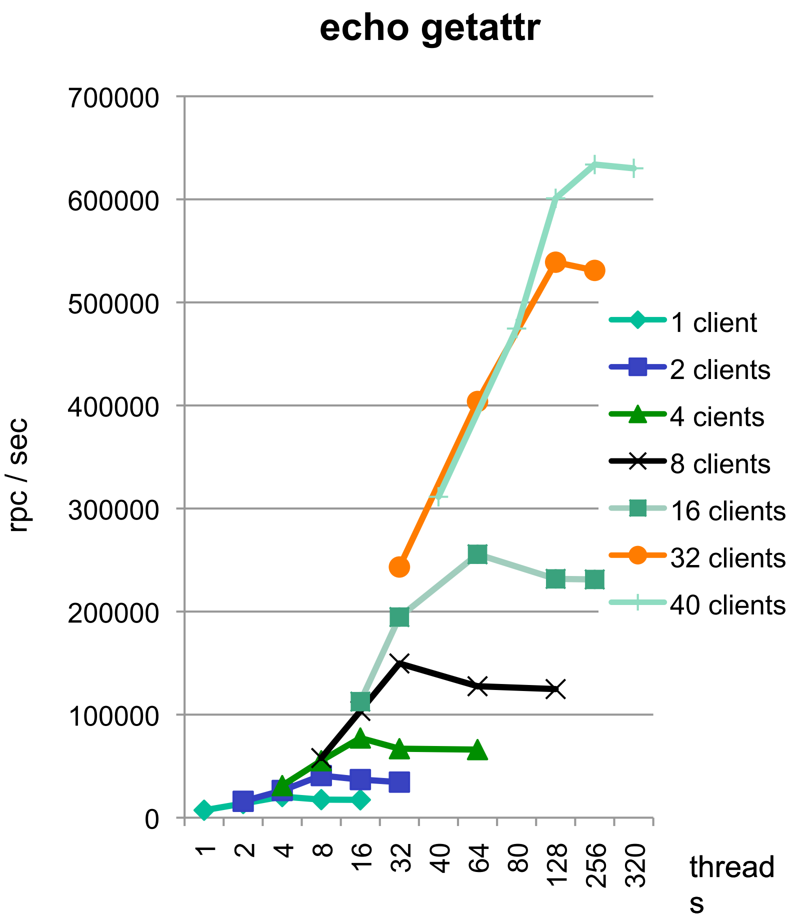
ptlrpc service

- Per-CPU service data
  - Locks, request buffer, request queue, reply state, AT…
  - More grained locks
    - Although they are local to each CPU, we still have cross CPU data access sometimes
- CPU affinity service threads pool
  - Local waitq for each CPU, otherwise all threads are serialized by the global waitq
  - LIFO wait queue can help to reduce active threads
- Cacheline optimization is always important

What we can improve

Ptlrpc performance





What we can improve

Hash tables and overprotected data

- Better hash for objects and ldlm resources
  - 1 million files tests, max search depth dropped from hundreds to less than 50
  - It not only reduces overhead of searching, also avoids cache pollution
    - One cache miss means hundreds of cycles on most processors
- Over-protected data
  - We protect the same data at different levels of stack
  - MDT takes 2 locks on create/unlink where as one would be enough (*survey still under* way)

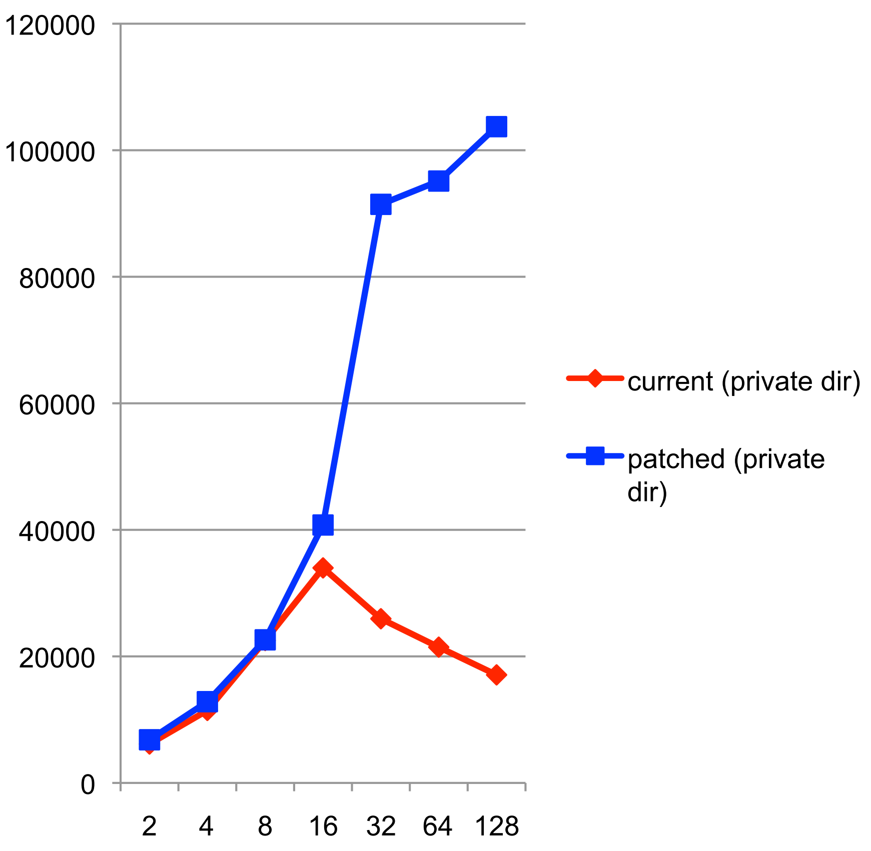
What we can improve

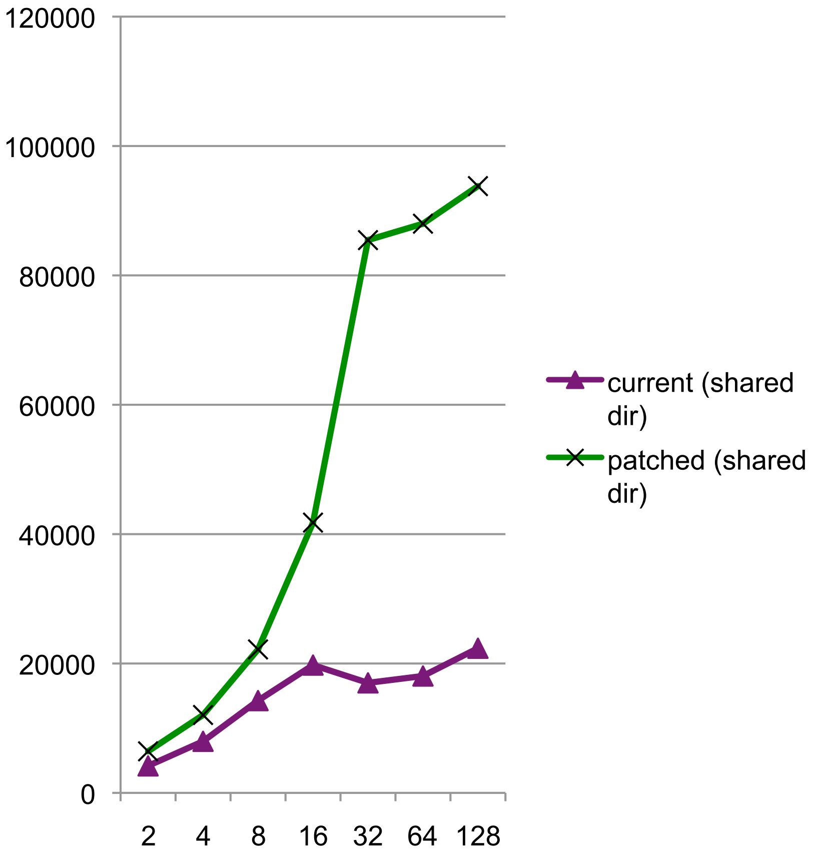
Everywhere

- Lazy update to globals
- Big reference count
- Per-CPU stats
- Code level improvement everywhere
  - Unnecessary lock dance
  - Wrong lock type
  - Redundant memset in our allocators

What we can improve

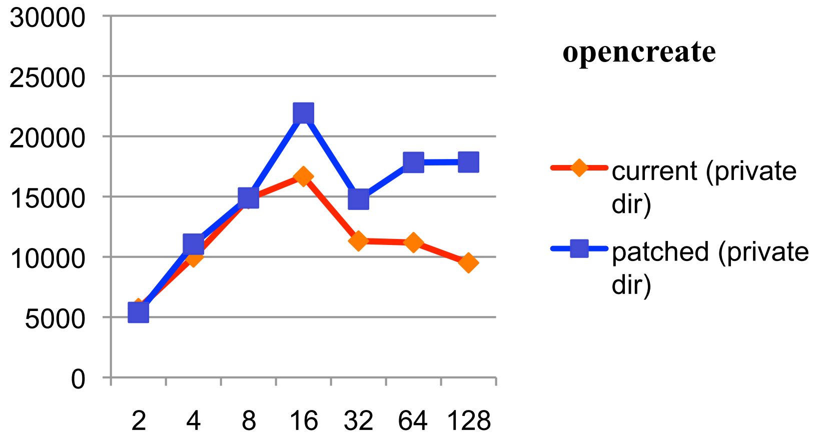
File stat (1-128 clients, 1 thread/client, 4K files/thread)

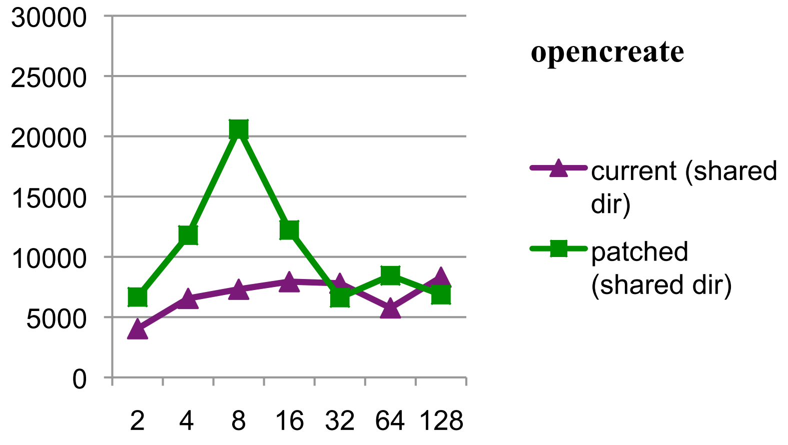


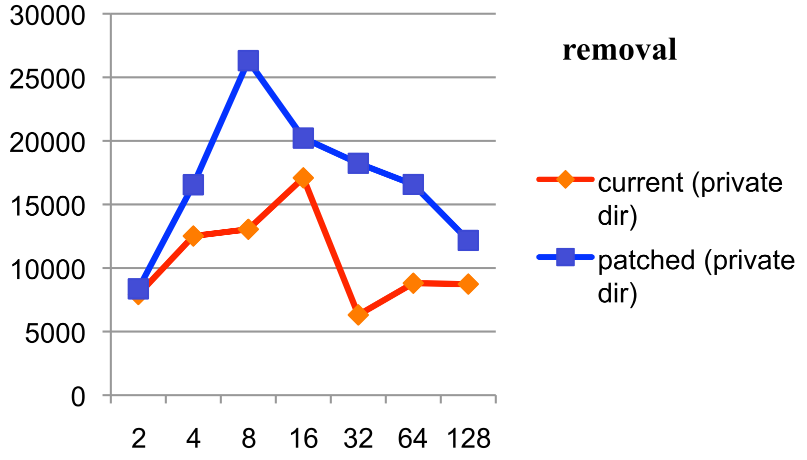


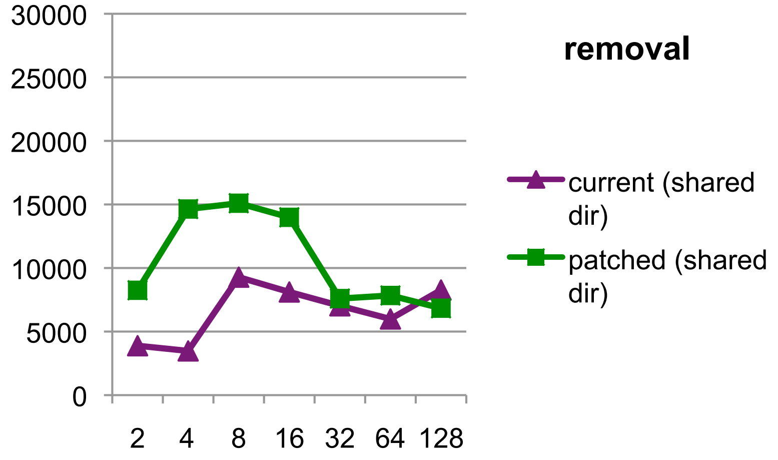
What we can improve

opencreate / removal
(1-128 clients, 1 thread/client, 4K files/thread)









Current status

- Implementation almost complete
- Initial tests show good result
- Need more survey on backend filesystem
- Metadata performance testing on Hyperion is underway
- BULL is helping us to test NUMIOA performance
- Changes are targeted for the Lustre 2 code branch

ORACLE IS THE INFORMATION COMPANY

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

ORACLE®