Scaling the Linux Kernel (Revisited): Using Ext4 as a Case Study

Theodore Ts'o
Google
January 25, 2011

Back to the Future!

Back to the Future!

In 2001...

- Linux 2.4
- IBM had just announced it was going to invest a billion dollars into Linux
- Linux had SMP, but it was barely scalable to 4 CPU's
    - What does this mean?
    - Why does this matter?


SMP:
When one CPU is not enough

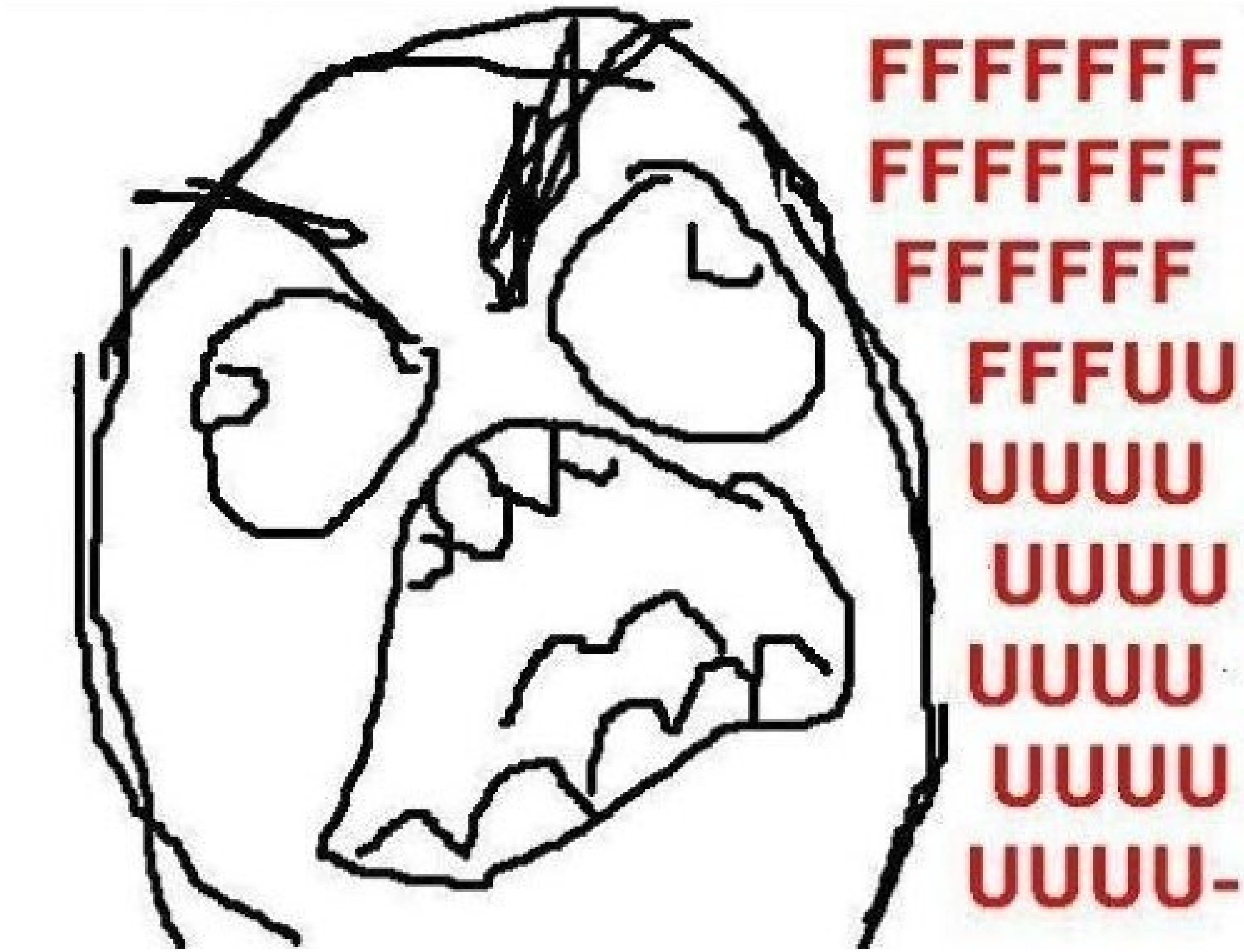
SMP: For when one CPU isn't enough

- Back then, one CPU was a lot slower
  - 6 Moores Law doublings in the last 9 years
- People who needed more computing power than that might buy a Sequent system with 8, 16, or 32 Pentiums

Then as now, SMP is expensive

- Coordinating a large number of sockets is expensive
  - What if two CPU's want to read/write to the same memory location?
  - How is cache coherency handled?
- The interconnect between between the CPU's is critically important
  - Cheating with NUMA (Non-uniform Memory Access)
- A 4 CPU machine costs far more than 4 times single CPU machines

I spent $$$ for 16 CPU's



I expect to get 16 CPU's!

Measuring SMP scalability

- Run one or more benchmarks using a single CPU
  - Volanomark
  - FSCache
  - Netperf
  - SpecWeb99
  - SPEC sdet
  - IOZONE
  - TPC-C/D/H

Measuring SMP scalability

- Run one or more benchmarks using a single CPU
- Now run the same benchmarks on a N CPU machine
- S = (score on 1 CPU / score on N CPU's)
- The OS is said to have a scalability of S out of N on that benchmark
  - Example: “Linux 2.6 has scales to 12 out of 16 CPU's on the fooblatz benchmark”

Scalability is ***HARD***

- 12 out 16 CPU's is actually considered pretty good for some benchmarks
  - But that means we're only using 75% of a machine that costs way more than 16 times a single CPU server!
- At the time, Linux 2.4 barely scaled to 4 CPU's on many benchmarks
  - Well less than 3x the single CPU benchmark score
  - On some benchmarks, Linux was actually *slower* on a 4 CPU machine (negative scalability!)

Linux Scalability Effort

- Spearheaded by IBM's Linux Technology Center
  - Other companies:
    - SGI
    - Intel
    - VA Linux
    - University of Michigan CITI

Linux Scalability Effort

- Spearheaded by IBM's Linux Technology Center
  - Weekly conference calls
  - Regular (weekly/monthly) benchmark measurements by a performance team
  - Developers stared at CPU and lock profiles to find and then fix bottlenecks
  - Wash. Rinse. Repeat.
- This went on for 2-3 years, and then victory was declared and everyone went home

Linux scalability as of 2003-4

- Good, but certainly not perfect
  - Scaled to 6-7 out of 8 CPU's on most benchmarks
  - Scaled to at least 12 out 16 CPU's on many benchmarks
  - Scaled to an acceptable number of CPU's on 32 CPU's
- Why did people stop?

Linux succeeded wildly on x86

- … but not necessarily on other platforms
  - Turned out people who spend $$$ on a high-end Sparc or Power server tended to prefer other Legacy OS's
  - At least in the enterprise market...
- At the time, few x86 servers had more than 8-16 CPU's, so there was less need to scale beyond that
- Linux was/is the king of scale-out computing

And so matters remained for 4-5 years

- (In an industry where 2 years == infinity)
- Rise of Linux on the embedded and mobile market
- CPU frequencies stopped doubling every 12 months
- CPU manufacturers have started putting 2, 4, 8+ cores in a socket
  - My desktop at work has 12 cores and it's not that expensive...

So here we are in 2010

- Scalability has started to matter again
  - Servers with 4 sockets aren't all that rare
  - And with 8 cores/socket, that means 32 CPU's will soon be a common configuration for Linux
- Time for kernel programmers to rediscover the lessons of scalability tuning
- Time for application programmers to start thinking about multi-threaded programming

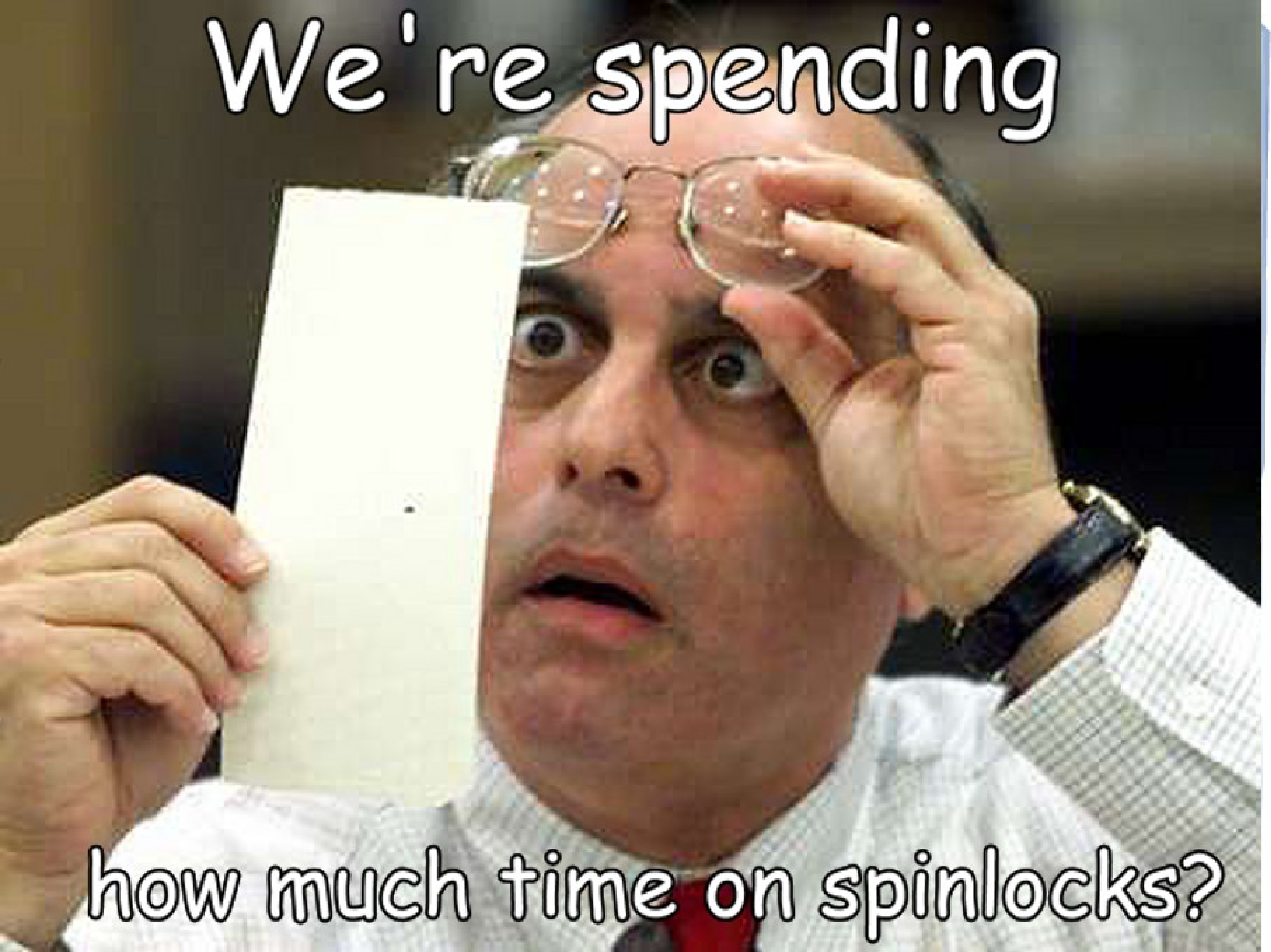

32 cores on a socket
sounds good to you?

Ext3 – Good enough scalibility

- Many x86 Linux workloads don't really stress the file system
  - Hit other bottlenecks first
- Enterprise databases tend to use Direct I/O to preallocated files
- Ext3 doesn't do well on head-to-head benchmark competitions
  - But most system administrators didn't care
    - It worked
    - Easy to service if things went wrong

Ext4 Scalability

- The story starts in April 2010
  - IBM real-time team was improving file system when when CONFIG_RT_PREEMPT is enabled
  - They noticed a minor problem with dbench...


We're spending
how much time on spinlocks?

Oprofile Report

```
27.39%     dbench  [kernel]              [k]
_raw_spin_lock_irqsave
            |
            |--90.91%-- rt_spin_lock_slowlock
            |          rt_spin_lock
            |          |
            |          |--66.92%-- start_this_handle
            |          |          jbd2_journal_start
            |          |          ext4_journal_start_sb
            |          |          |
```

What do the locks protect?

- Fortunately, this was well documented in the jbd/jbd2 header files
  - The j_state_lock protects fields in the journal structure
  - The t_handle_lock protects fields in the transaction handle structure
- The jbd2_journal_start() and jbd2_journal_stop() functions were taking both locks

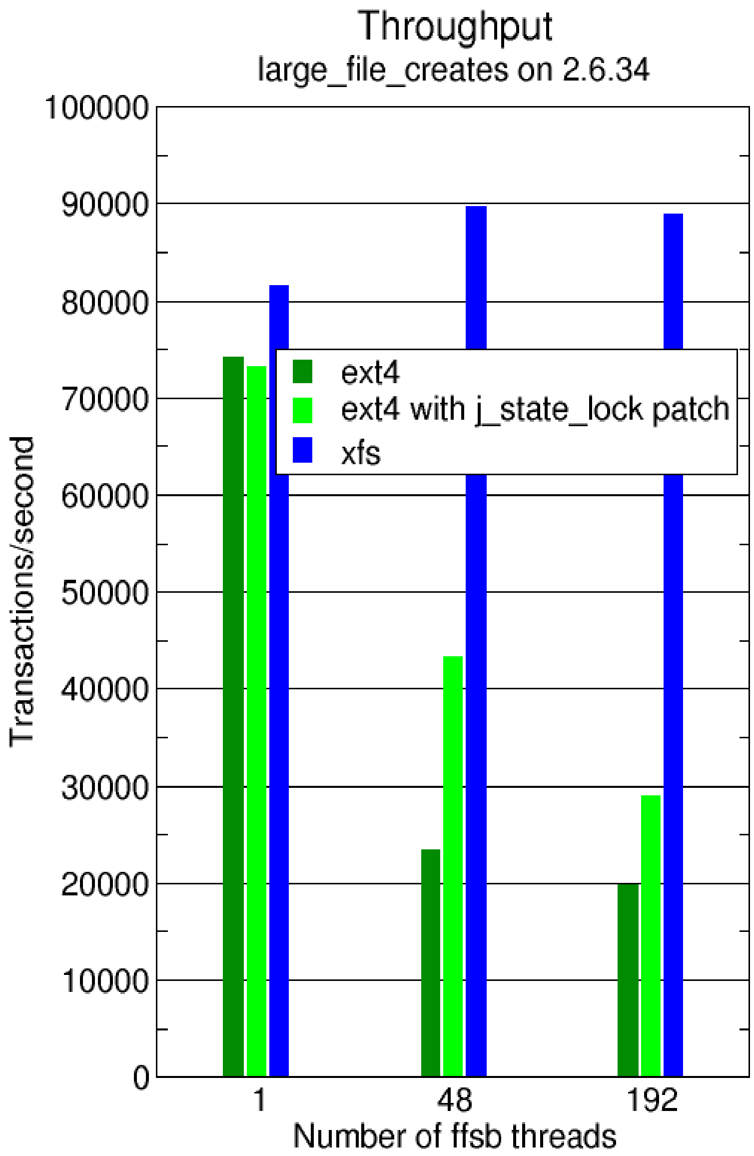
A quick jbd2 lesson...

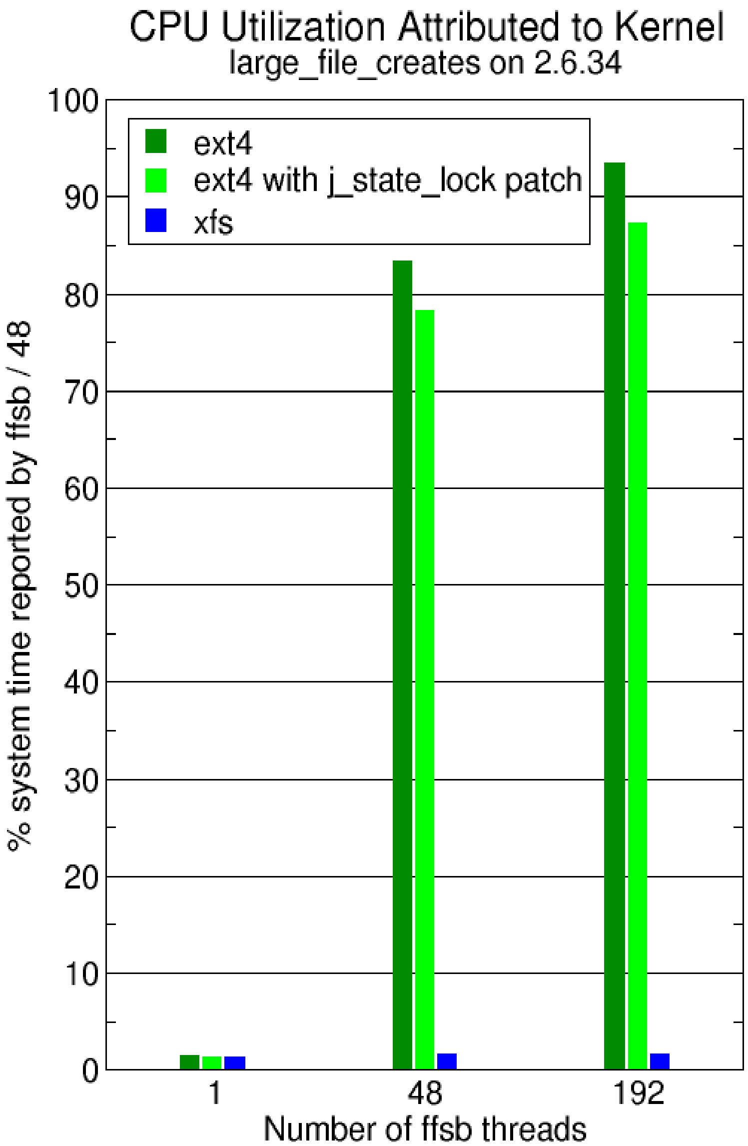
- Transactions are expensive → group multiple file system operations into a single transaction
  - Transaction commits happen every 5 seconds or when the transaction or the journal is full
- Each file system operation is bracketed by a jbd2_journal_{start,stop}() call
  - jbd2_journal_start() gets passed a worst-case estimate of how many blocks will be modified
    - Checks to see if a new transaction must be started

Removing unnecessary locking

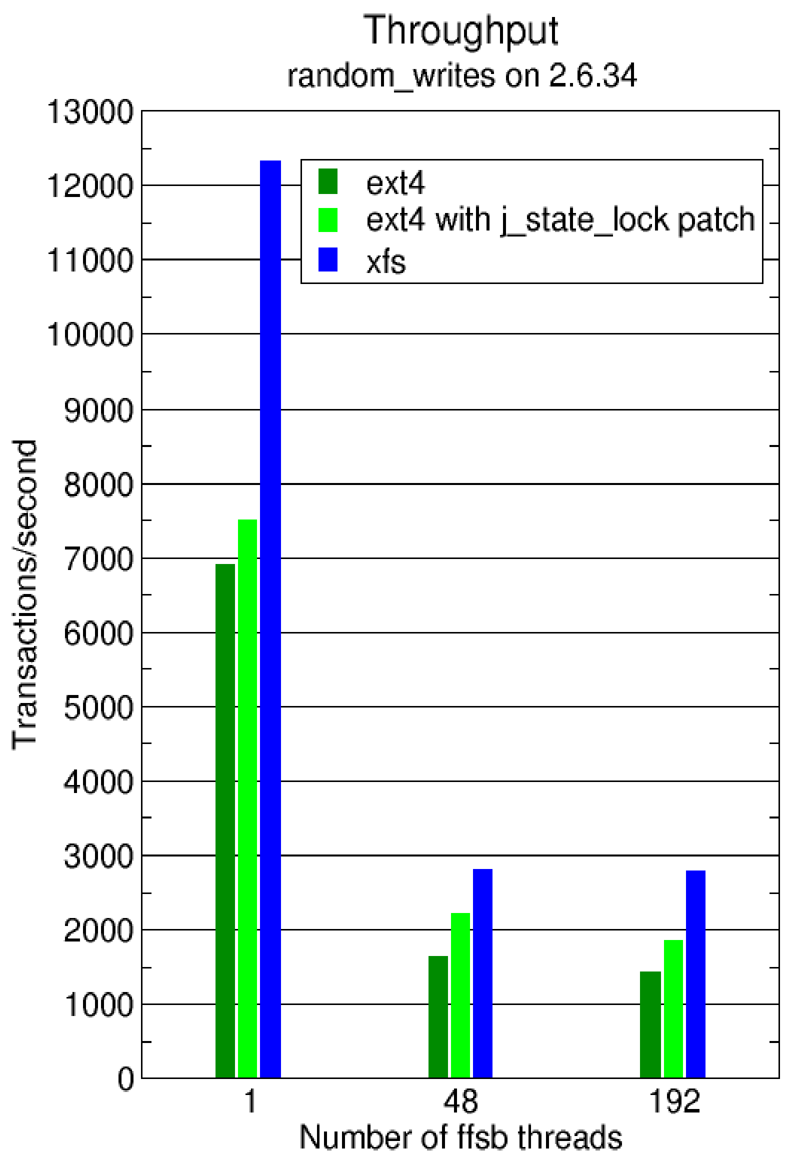
- Turns out in jbd2_journal_stop() was taking the j_state_lock spinlock, but...
  - It was not touching anything protected by that lock
- Removing it resulted in an immediate improvement for the real-time folks
- Eric Whitney from HP ran some tests using a large (48 core) AMD system with hw RAID....

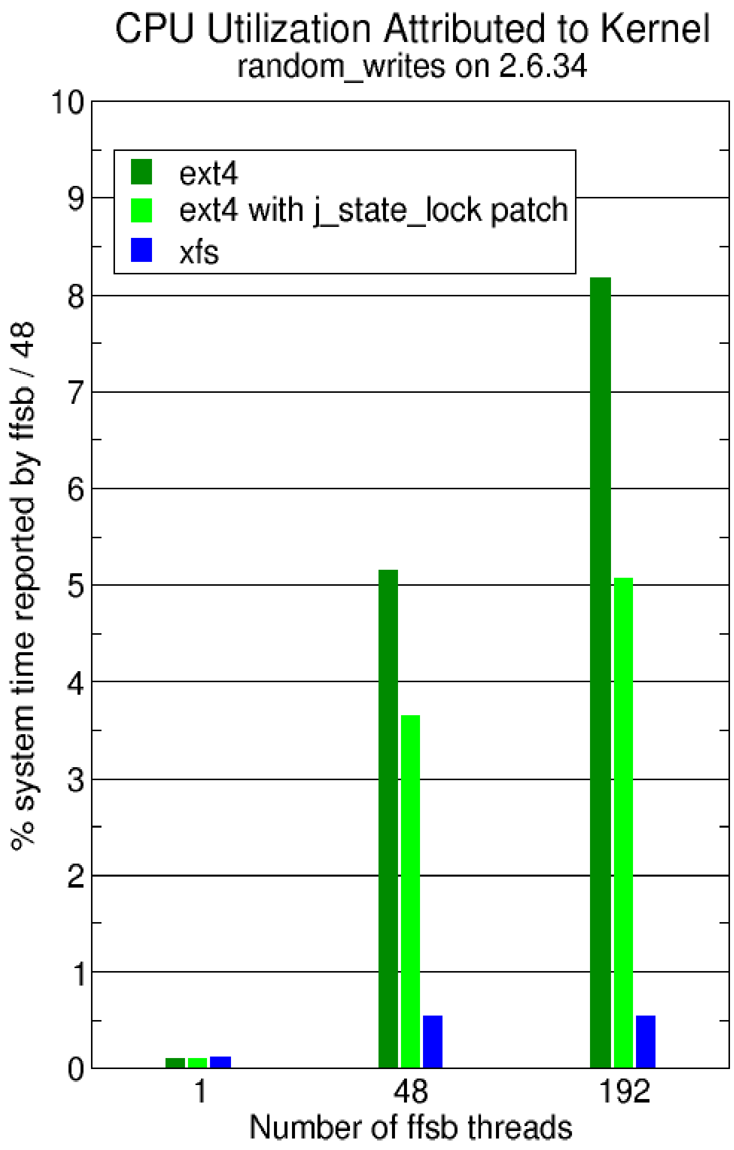
j_state patch results





j_state patch results, II





Can we do better?

- While benchmarking Eric Whitney also took measurements using an even more powerful lock profiling tool: lock_stat
  - Enabled via CONFIG_LOCK_STATS
  - Start profiling: echo 1 > /proc/sys/kernel/lock_stat
  - Stop profiling: echo 0 > /proc/sys/kernel/lock_stat
  - Get results: cat /proc/lock_stat
  - Clear statistics: 0 > /proc/lock_stat

lock_stat eye chart

class name con-bounces contentions waittime-min waittime-max waittime-total acq-bounces acquisitions holdtime-min holdtime-max holdtime-total

----

&(&journal->j_state_lock)->rlock#2: 99868941 114503908 0.10 1949243.34 109274601519.1 114792575 115009746 0.00 15221.59

lock_stat results (after j_state patch)

lock con-bounces contentions waittime-max waittime-total

acq-bounces acquisitions holdtime-max holdtime-total

j_state_lock 60044534 64207387 2334498.88 65679240103.52

66614119 71942836 53365.97 812877772.83

t_handle_lock 16221754 16230567 4919.93 64694019.69

j_state_lock details

50052861 start_this_handle+0xb9/0x540 [jbd2]

14148330 jbd2_log_start_commit+0x2b/0x50 [jbd2]

102 kjournald2+0x22d/0x240 [jbd2]

175 jbd2_journal_commit_transaction+0xb1/0x15d0 [jbd2]

----

t_handle_lock details

13912339 jbd2_journal_stop+0x15c/0x280 [jbd2]

2318336 start_this_handle+0xdc/0x540 [jbd2]

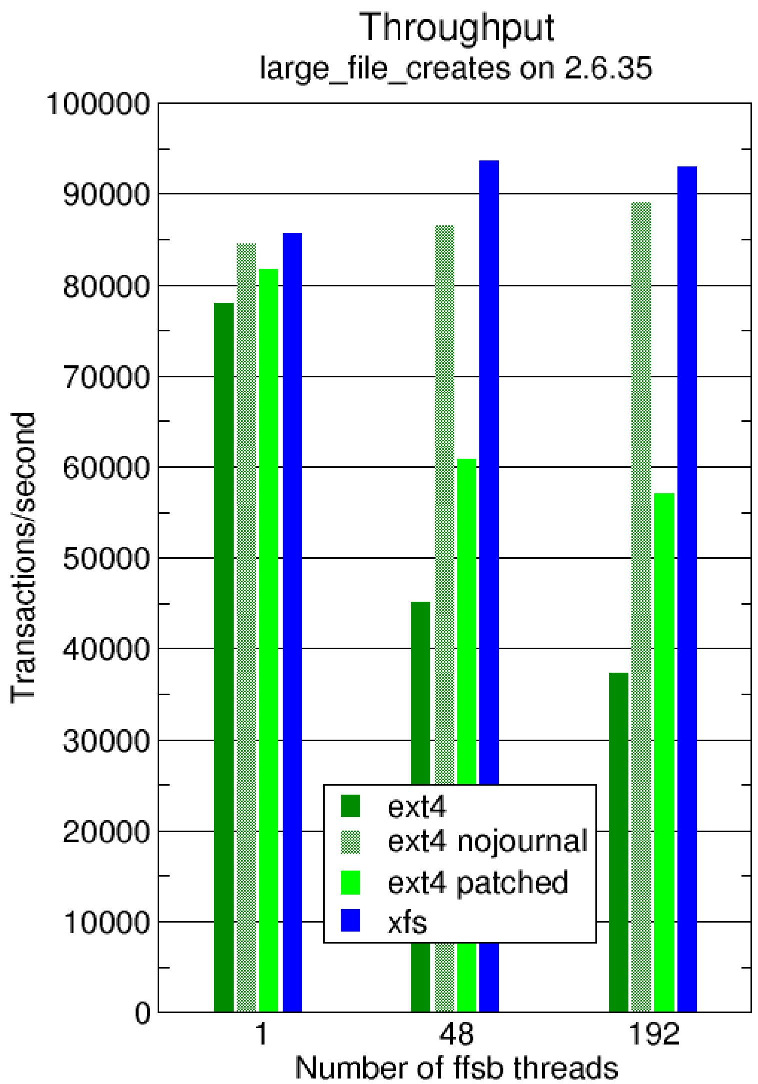
6 jbd2_journal_commit_transaction+0x127/0x15d0 [jbd2]

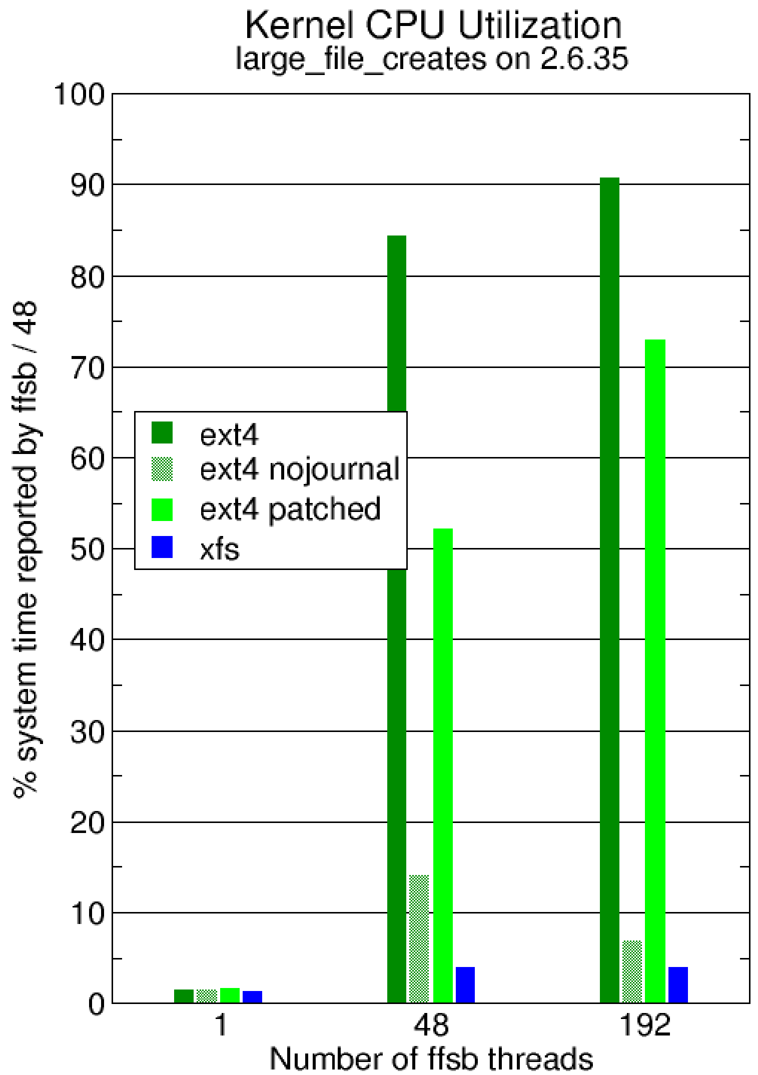
3562931

What to do, what to do....

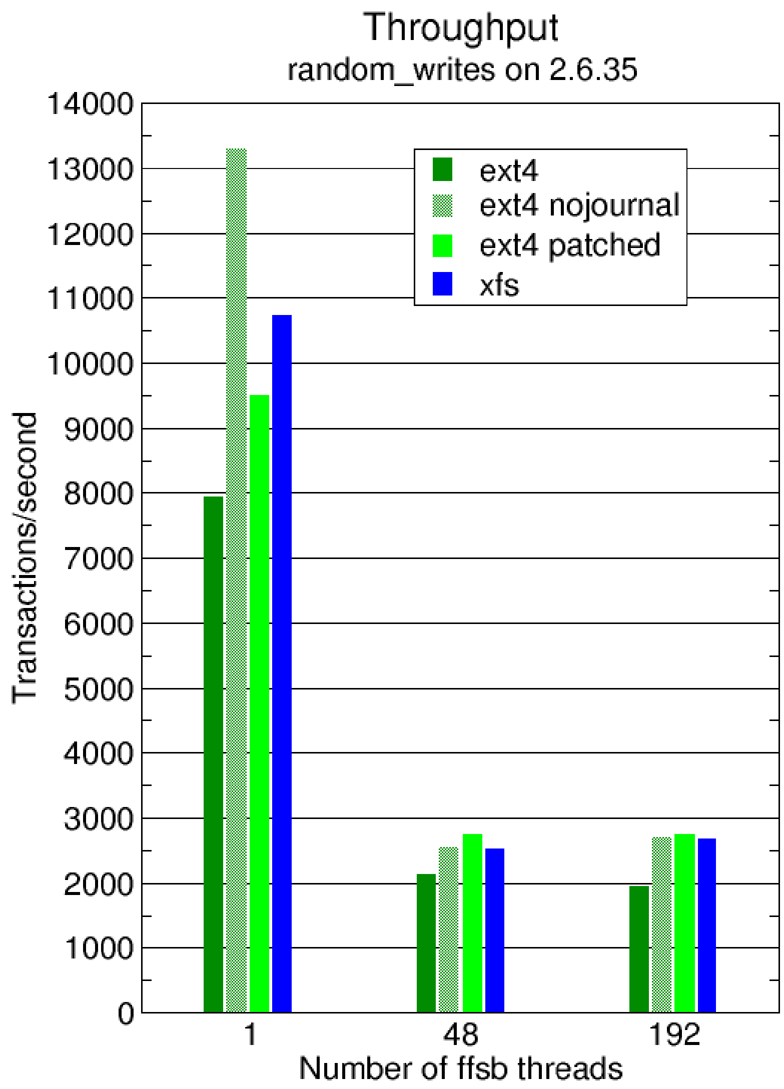
- Use atomic_t variables to avoid taking t_handle_lock
  - For statistics (are the statistics really needed?)
  - Journal accounting
    - # of handles
    - # of blocks modified to be stored in the journal
- Use a read/write lock for j_state_lock
  - Most of the time, starting and stopping handles only needs a read lock for j_state_lock
- With these changes, jbd2 handles can now be started and stopped in parallel

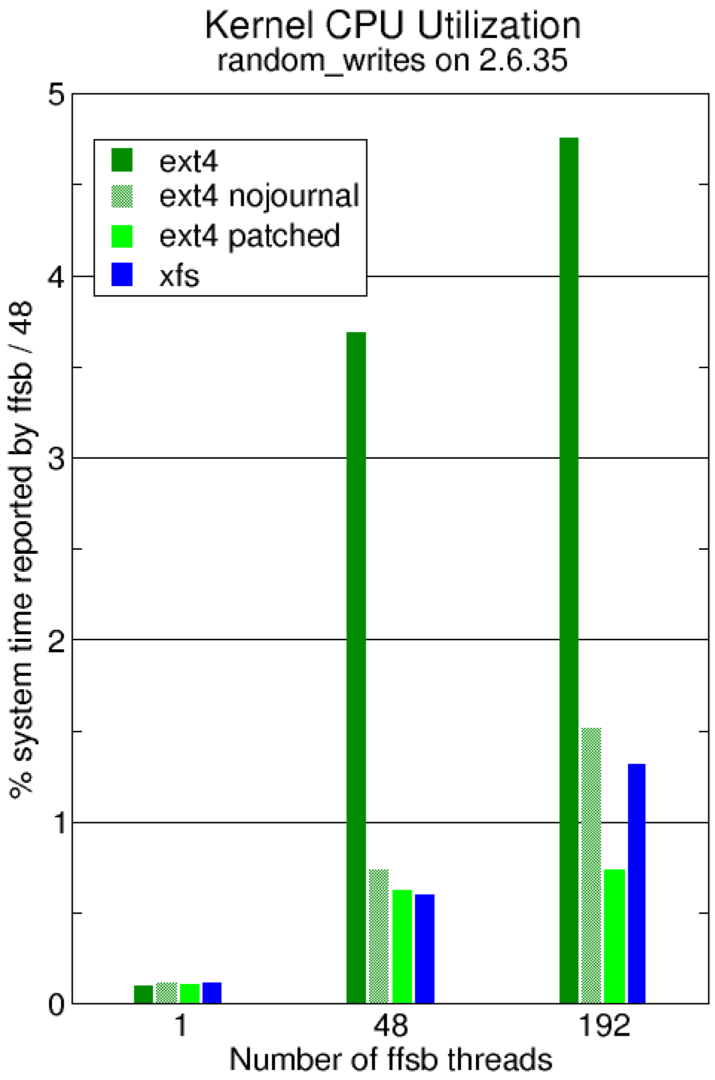
Journal scalability benchmarks





Journal scalability benchmarks, II





lock_stat results after patch

class name con-bounces contentions waittime-max waittime-total

acq-bounces acquisitions holdtime-max holdtime-total

blk queue_lock 25476870 25509021 1047374.56 7611283803.70

26031778 58491403 891590.17 307965771.19

j_state_lock-W 12374549 12459235 5137551.04 14262268154.07

Queue lock details

23935236 __make_request+0x54/0x4b0

195527 scsi_request_fn+0x3de/0x540

9791 generic_unplug_device+0x26/0x50

209210 scsi_device_unbusy+0xa5/0xe0

17483 generic_unplug_device+0x26/0x50

23794087 __make_request+0x54/0x4b0

j_state_lock details

12457104 jbd2_log_start_commit+0x2b/0x60

8468910 start_this_handle+0x98/0x540

2 kjournald2+0x204/0x220

9 jbd2_journal_commit_transaction+0x36a/0x14a0

3804131 start_this_handle+0x98/0x540

17119236 jbd2_log_start_commit+0x2b/0x60


The top lock
is no longer in jbd2

Now what?

- The top lock (as of 2.6.36) is no longer a lock in the jbd2 layer
  - The primary contention point left shows up when we need to start a new commit, and have to wait for all outstanding handles to finish → unavoidable
- Now the primary problem is caused by how ext4 submits its out to the block I/O layer


Ext4 buffered writes
are sent 4k at a time to the bio layer

Ext4 Buffered Writes

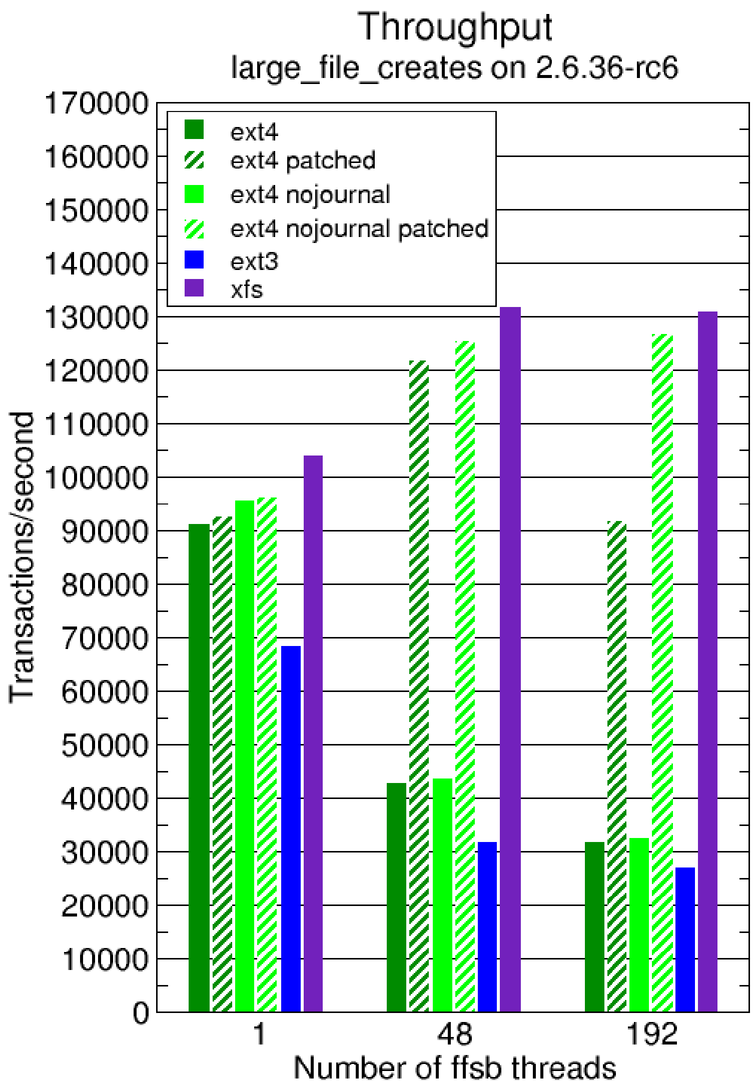
- Are submitted 4k at a time...
  - … buffered reads use mpage_readpages()
  - … writes can't due to journaling requirements
- The writes get merged back together by the block queue layer, but wastes CPU time and locking overhead
  - Makes blktraces very large
  - I/O statistics can be confusing
    - Are stats pre-merge or post-merge

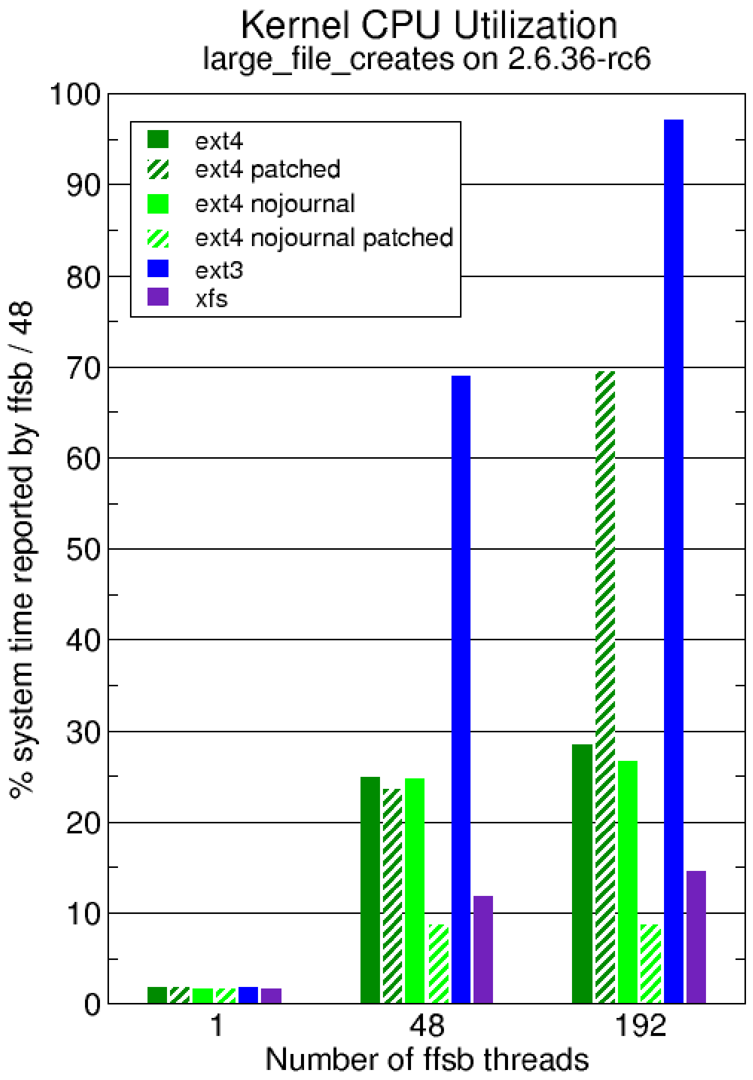

WA - TZAA!

A small matter of programming...

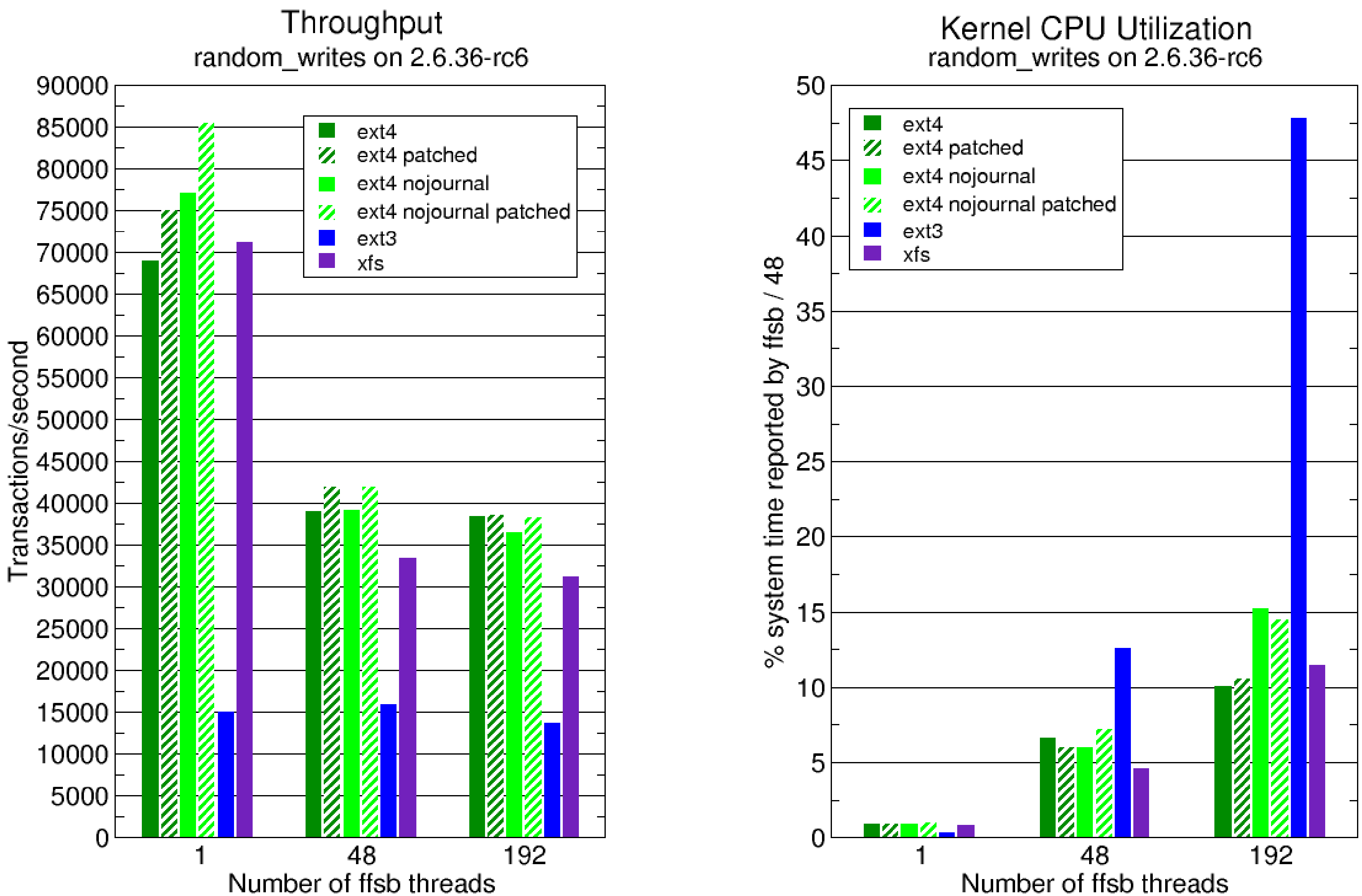
- Implementation in fs/ext4/page_io.c
  - Provides an (almost) drop-in replacement for block_write_full_page(), named ext4_bio_write_page()
  - ext4_io_submit writes all of the batched pages
- Required a massive overhaul of the bottom half-of the ext4 buffered write submission path
  - Everything from mpage_da_submit_io() on down...

And the results...





Results, II



Throughput
random_writes on 2.6.36-rc6
Transactions/second
Number of ffsb threads
1
48
192
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000
55000
60000
65000
70000
75000
80000
85000
90000
ext4
ext4 patched
ext4 nojournal
ext4 nojournal patched
ext3
xfs
Kernel CPU Utilization
random_writes on 2.6.36-rc6
% system time reported by ffsb / 48
Number of ffsb threads
0
5
10
15
20
25
30
35
40
45
50

Unfortunately not quite done

- Bug reported when using dm_crypt and postgres
  - Causes data corruption


Your code:
it has a bug

Unfortunately not quite done

- Bug reported when using dm_crypt and postgres
  - Causes data corruption
- Enhancement disabled before 2.6.37 released
  - Can be re-enabled via mblk_io_submit mount option
- Now all we have to do is find the bug....

Summing up...

- We need to pay attention to SMP scalability
  - Requires careful thinking about multi-threading
    - Harder to debug; lots of potential for race conditions
    - Performing tuning can be tricky
- Techniques for Performance
  - Atomic variables
  - Read/write locking
  - Finer-grain locking
  - Batch work together

For Userspace, too!

- Can userspace applications do the same thing?
  - atomic_t variables can work if you import the headers
  - Use pthread mutexes
    - Linux futex can speed up things
  - Don't use spinlocks!
- Multithreading tools for userspace:
  - Valgrind's drd tool to find data races
  - Lennart Poettering's mutrace

Picture from Paul Downey, http://www.flickr.com/photos/psd/2086641/