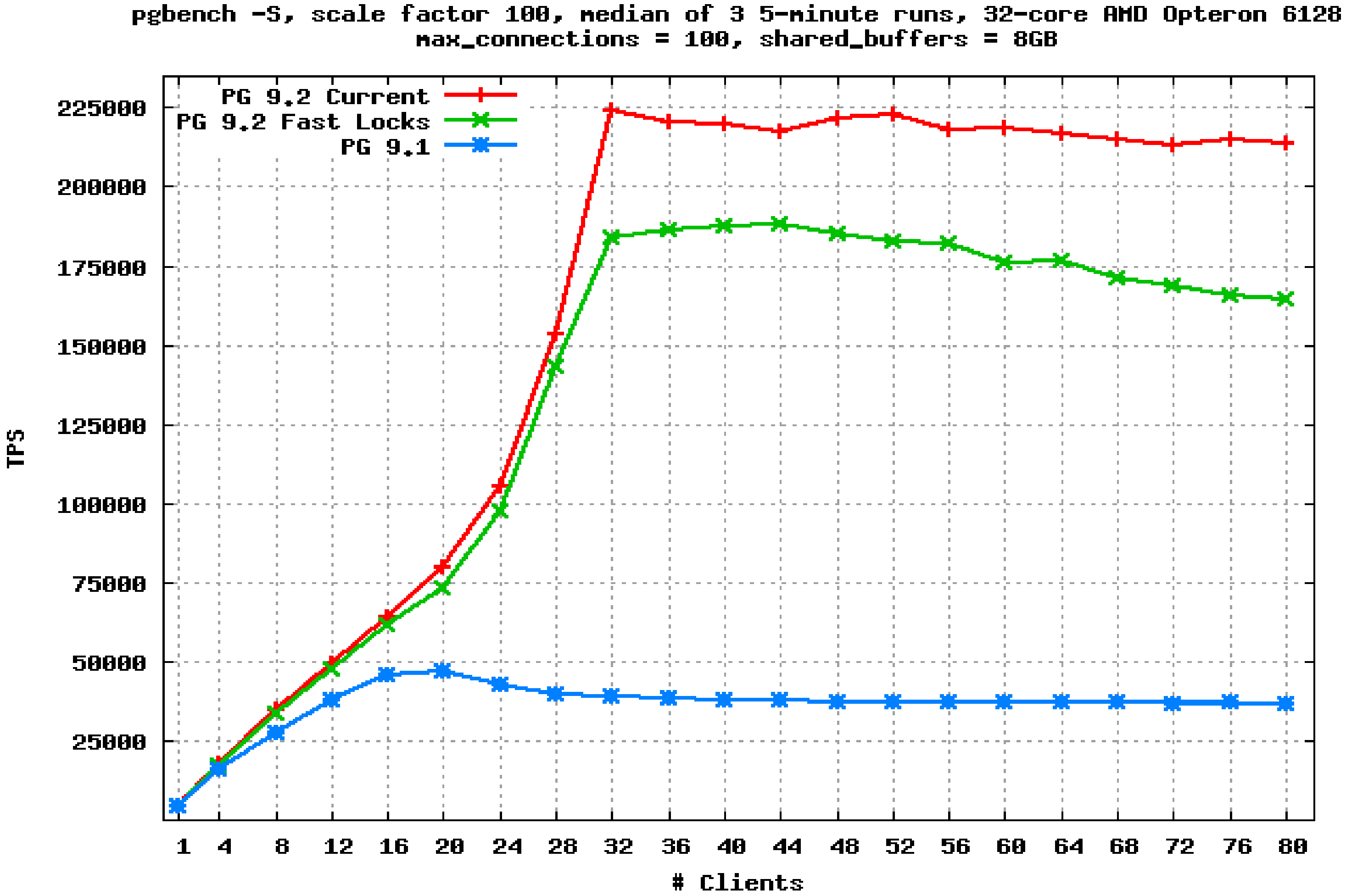
# Performance and Scalability Enhancements in PostgreSQL 9.2

Robert Haas
Senior Database Architect

# Areas of Improvement

- High Concurrency
- Index-Only Scans
- Parameterized Plans
- Indexing
- Sorting
- Power Consumption
- Miscellaneous

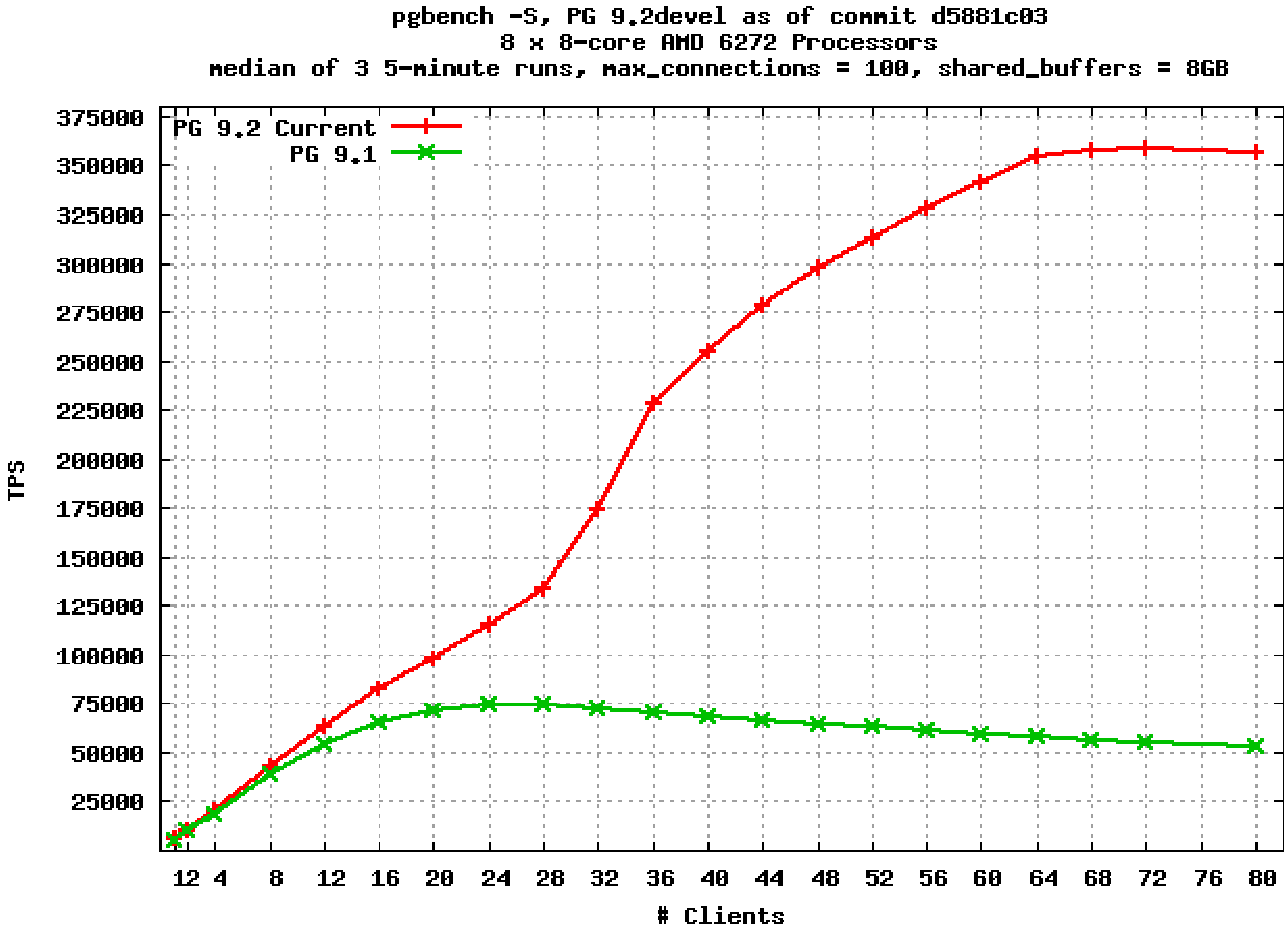
# Read Scalability (as of September 2011)



# High Read Concurrency

- Added a “fast path” to lock manager to allow “weak” relation locks to bypass the main lock manager in most cases.
- Extended the “fast path” to allow “virtual transaction ID locks” to bypass the main lock manager.
- Wait-free test for new shared invalidation messages.

# Did I Say 32 Cores?  (April 2012)



EnterpriseDB®
The Enterprise PostgreSQL Company

# High Write Concurrency

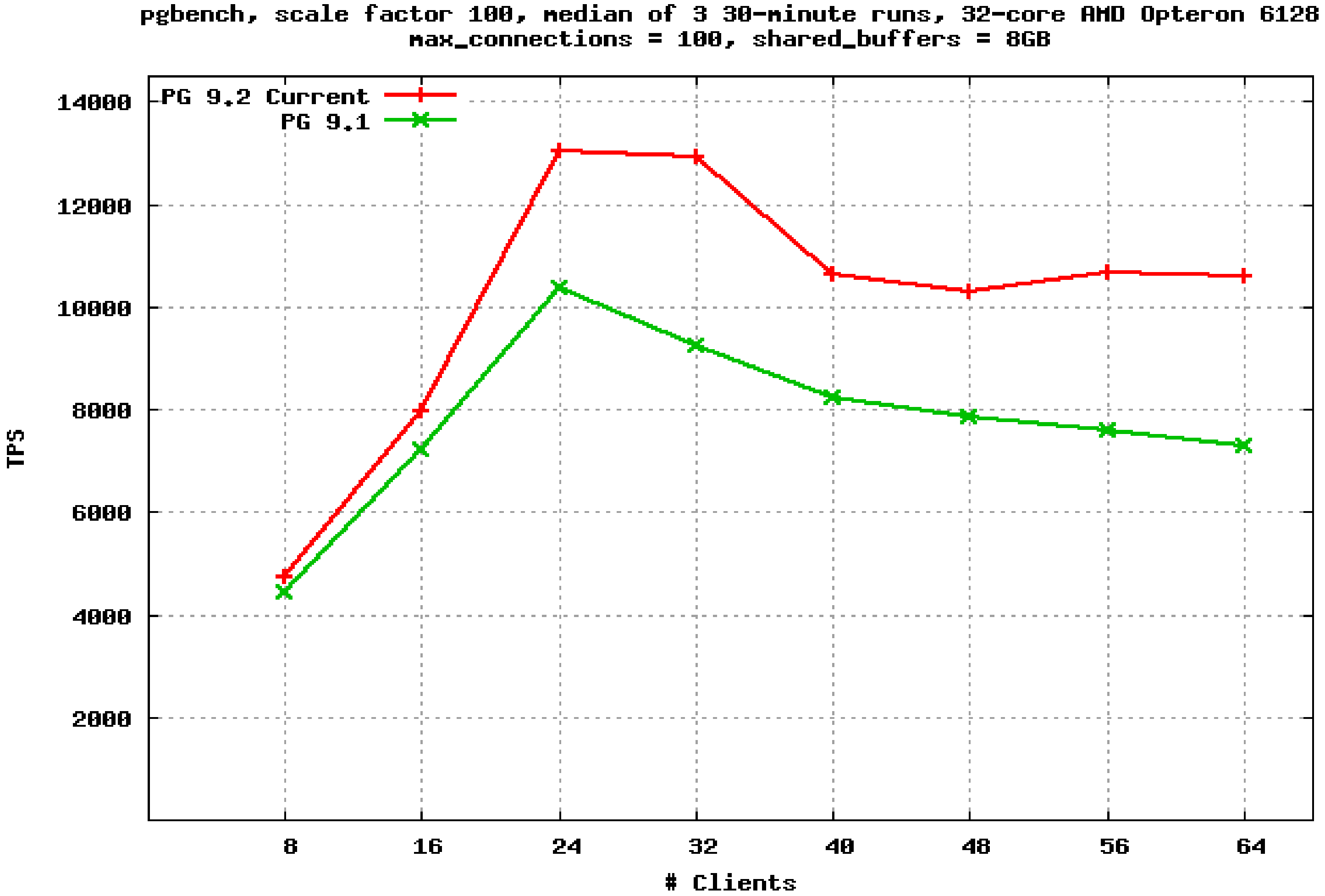
- Moved “hot” members of PGPROC data structure to a separate array to minimize cache line passing.

- Improved WAL-writer responsiveness to reduce CLOG traffic.
- Increased number of CLOG buffers.
- Fixed SLRU buffer replacement algorithm.
- Eliminated some redundant CLOG lookups during index scans.

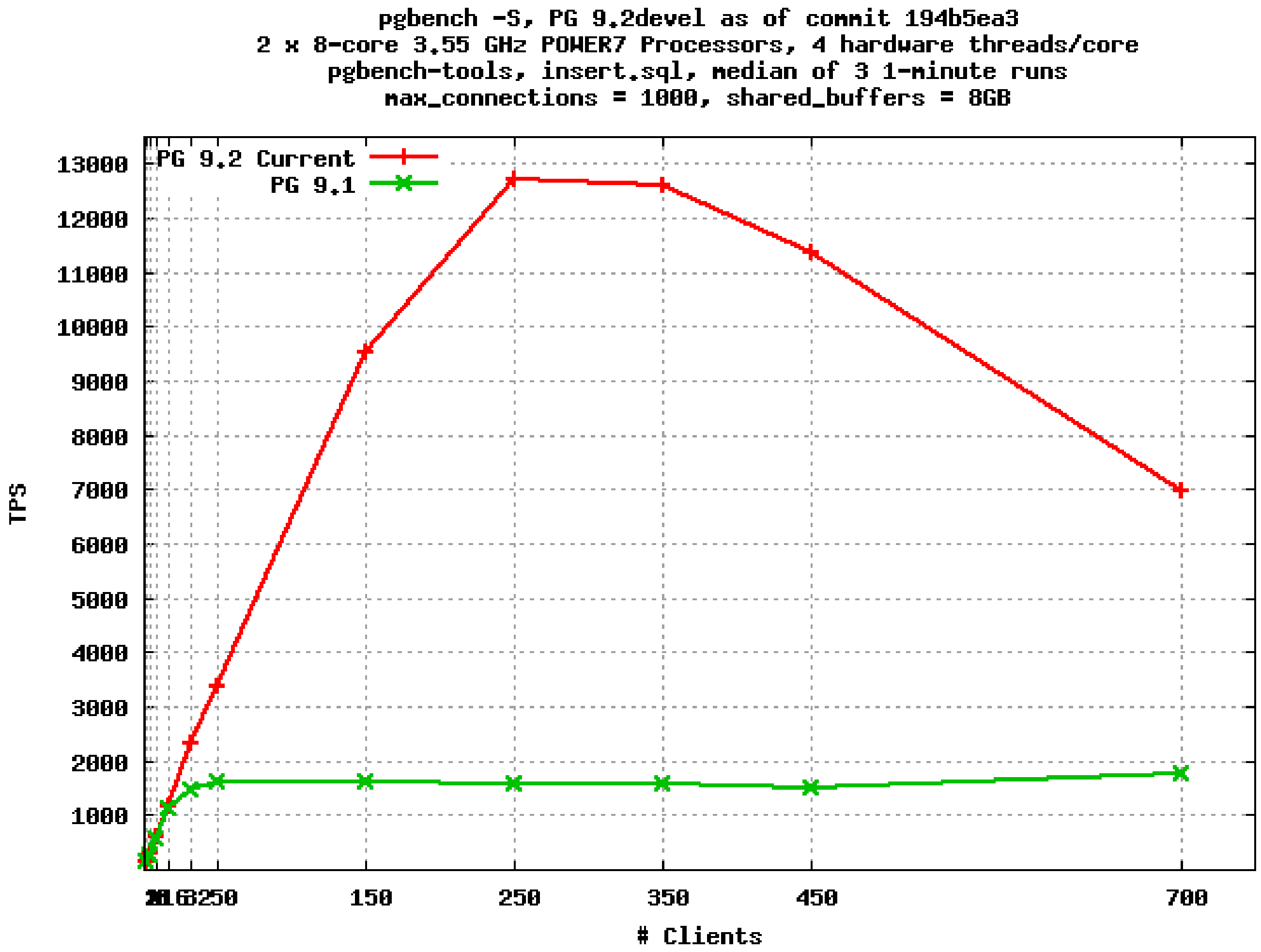
- Improved “piggybacking” of WAL flush, resulting in  better “group commit” behavior.

- Reduce volume of WAL generated by COPY.

# Write Scalability (as of February 2012)



# Commit Scalability (as of March 2012)



# Index-Only Scans: Architecture

- Visibility map stores 1 bit per heap (table) page, which may be set only if every tuple on the page is visible to all current (and future) transactions.  One visibility map page (8kB) covers ~512MB of heap.

- If all necessary data is available from index tuple, probe visibility map rather than loading heap page; if page is all-visible, skip heap fetch; otherwise, fetch heap page normally.

- Visibility map has existed since 8.4 as a hint for VACUUM; was not crash-safe before 9.2.

# Index-Only Scans: Example

- MacBook Pro, 4GB, shared_buffers=400MB
- pgbench -i -s 1000
- pgbench -n -S -T 300 -c 8 -j 8
  - SELECT abalance FROM pgbench_accounts WHERE aid = :aid
  - Primary key lookup via pgbench_accounts_pkey
- create index pgbench_accounts_covering on pgbench_accounts (aid, abalance);
- pgbench_accounts is 13GB; each index is 2142MB

# Index-Only Scans: Test Results

- Default Configuration:
  tps = 63.288028 (including connections establishing)

- With Covering Index:
  pgbench -n -S -T 300 -c 8 -j 8
  tps = 302.459713 (including connections establishing)

# Parameterized Plans: Example

- Same MacBook Pro, same pgbench -i -s 1000
- select * from generate_series(1, 10) g left join (pgbench_accounts a join pgbench_accounts b on a.aid = b.aid) on g * 10000000 = a.aid;
- Times on 9.1: 246963.437 ms, 250191.531 ms, 251019.811 ms
- Times on 9.2devel: 579.341 ms, 1.398 ms, 1.211 ms

# Plan with No Parameterization: 9.1

```
Merge Right Join
(cost=59.83..10472347.77 rows=1000 width=198)
   Merge Cond: (a.aid = ((g.g * 10000000)))
   ->  Merge Join
        (cost=0.00..9972270.44 rows=100000000 width=194)
         Merge Cond: (a.aid = b.aid)
         ->  Index Scan using pgbench_accounts_pkey
               on pgbench_accounts a
               (cost=0.00..4236135.22 rows=100000000 width=97)
         ->  Index Scan using pgbench_accounts_pkey
               on pgbench_accounts b
               (cost=0.00..4236135.22 rows=100000000 width=97)
   ->  Sort  (cost=59.83..62.33 rows=1000 width=4)
         Sort Key: ((g.g * 10000000))
         ->  Function Scan on generate_series g
               (cost=0.00..10.00 rows=1000 width=4)
```

# Plan with Parameterization: 9.2devel

```
Nested Loop Left Join
(cost=0.01..235886.80 rows=1000 width=198)
  ->  Function Scan on generate_series g
      (cost=0.00..10.00 rows=1000 width=4)
  ->  Nested Loop
      (cost=0.00..235.86 rows=1 width=194)
        ->  Index Scan using pgbench_accounts_covering
              on pgbench_accounts a
              (cost=0.00..117.94 rows=1 width=97)
            Index Cond: ((g.g * 10000000) = aid)
        ->  Index Scan using pgbench_accounts_covering
              on pgbench_accounts b
              (cost=0.00..117.91 rows=1 width=97)
            Index Cond: (aid = a.aid)
```

# Indexing

- indexedcol op ANY(ARRAY[...]) in plain indexscans
- Better selectivity estimation for array operators <@, &&, @>
- Improvements to GiST indexing: indexes build more quickly, and are of better quality
- New index type: SP-GIST

# Sorting

- “Sort support” infrastructure reduces argument packing and unpacking.

- Specialized versions of qsort are more efficient than a general qsort.

# Reduced Power Consumption

- In PostgreSQL 9.1, there are approximately 11.5 auxilliary process wake-ups per second.
- In PostgreSQL 9.2beta1, as of 2012-05-18, there are approximately 0.4 auxiliary process wakeups per second (on a system that's been idle for a few minutes).
- For hosting providers with many virtualized, lightly-used copies of PostgreSQL, fewer wake-ups translates into real cost savings.

# Other Improvements

- Plan cache reduces the danger of getting a “bad” query plan when using prepared queries (however, more work is probably still needed).
- User-space Access Vector Cache for sepgsql.
- Faster array assignment in PL/pgsql due to caching of type information.
- Improved spinlock implementation on HP Itanium.

# Lessons Learned

- Plain old pgbench is helpful.  Write runs need to be at least 30 minutes long, 5 minutes is enough for reads.  Repeat each test 3x to identify outliers.

- TPS is constant for read-only tests, but varies widely for write tests.  pgbench -l output allows construction of tps-vs-time graph; but overhead is a problem.

- LWLOCK_STATS are very helpful.  Adding instrumentation to count number of “spins” required to acquire the lwlock's spinlock is even better.

# Lessons Learned (1 of 3)

- Plain old pgbench is helpful. Write runs need to be at least 30 minutes long, 5 minutes is enough for reads. Repeat each test 3x to identify outliers.

- TPS is constant for read-only tests, but varies widely for write tests. pgbench -l output allows construction of tps-vs-time graph; but overhead is a problem.

- LWLOCK_STATS are very helpful. Adding instrumentation to count number of “spins” required to acquire the lwlock's spinlock is even better.

# Lessons Learned (2 of 3)

- CPU profiling via gprof or oprofile is almost useless, because the overhead is too high. perf record has acceptable overhead, but not too useful for scalability because LWLock contention deschedules the process.
- Context switch profiling (perf record -e cs -g) is useful for identifying which call paths are causing LWLock-related context switches.
- Custom instrumentation is awesome.

# Lessons Learned (3 of 3)

- Benchmarking extreme workloads (like pgbench) exacerbates bottlenecks, making it easier to judge the effectiveness of solutions.
- Many (but not all) problems are easy to fix once you understand what's really happening.  But that can take months.
- Measuring system performance along multiple axes (tps, latency, frequency of lock contention, duration of lock stalls) reveals different problems.
- LWLocks are poorly suited to many synchronization problems, but it's not exactly clear what would be better.

# What's Next?

- Buffer replacement is mostly single-threaded.
- WAL insertion is single-threaded. Particularly nasty just after a checkpoint!
- fsync on a busy system can take many seconds to complete – can cause nasty CLOG stalls, and slowness around checkpoints.
- The more other things we fix, the worse ProcArrayLock looks: snapshot acquisition vs. transaction commit.

# Thank You!

- Any Questions?