Multi-CPU performance in PostgreSQL 9.2

Heikki Linnakangas / EnterpriseDB

Scalability 101

- Scalability defined:
- when you throw more hardware at a problem, the software can make use of it
- This presentation focuses on multi-CPU scalability

Multi-CPU systems today

- Even laptops typically have 2 cores
- Servers
  - Low-end: 4 cores
  - High-end: 32 cores and beyond

- RAM:
  - > 128 GB

Journey begins: Itanium test box

machinfo

CPU info:

8 Intel(R) Itanium(R) Processor 9350s (1.73 GHz, 24 MB)

4.79 GT/s QPI, CPU version E0

32 logical processors (4 per socket)

Memory: 392917 MB (**383.71 GB**)

...

Platform info:

Model: "ia64 hp Integrity BL890c i2"

Making software to scale

1. Benchmark
2. Identify bottleneck
3. Fix bottlenck

Choosing the benchmark

- There are workloads where PostgreSQL scales great
  - SELECTs, bundled into large transactions
  - > 64 CPUs, no problem!
- On other workloads, PostgreSQL scales poorly
  - Concurrent inserts choke at 2 CPUs

COPY

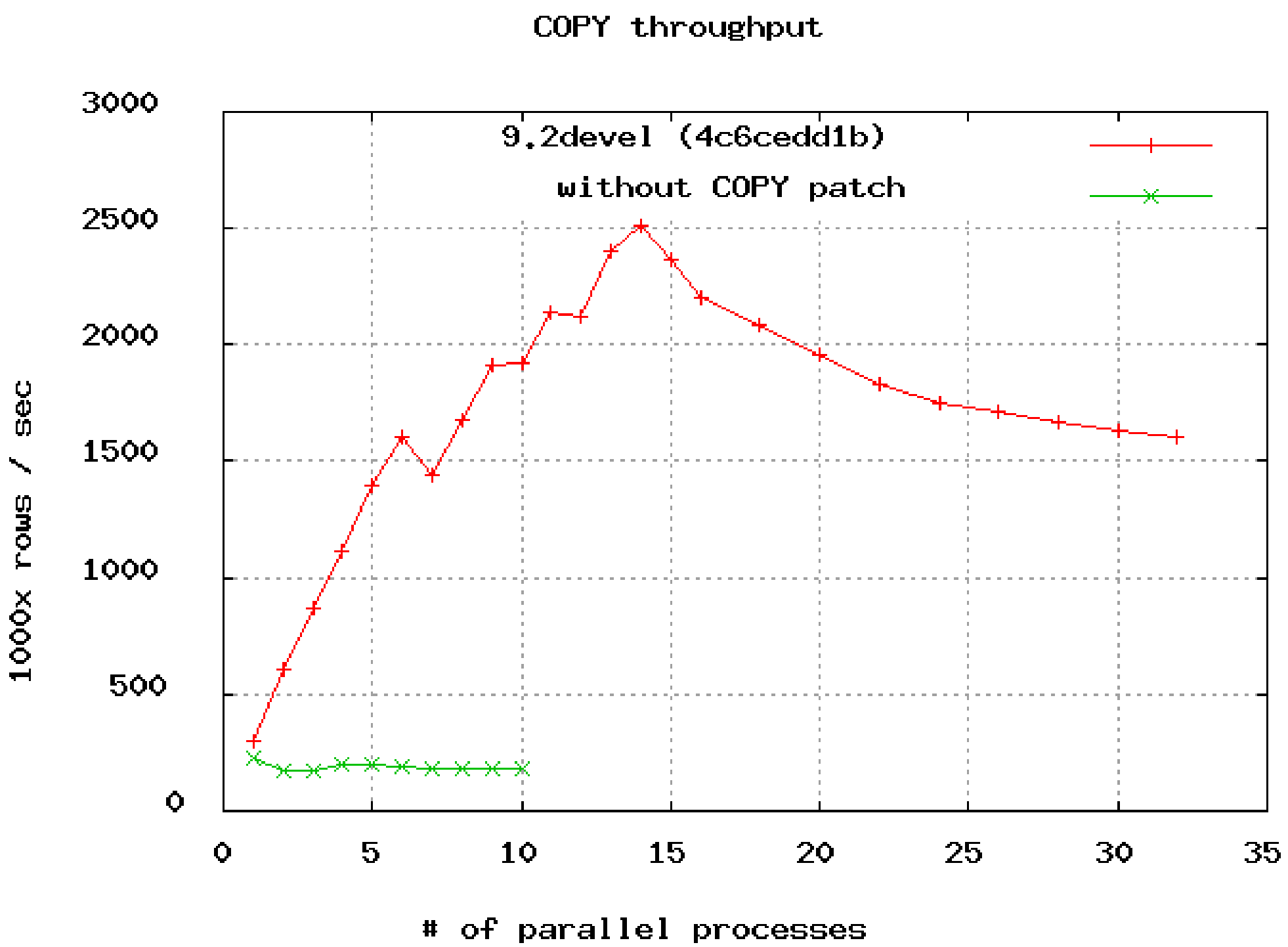
- Bulk loading data with COPY has always scaled well
  - unless it needs to be WAL-logged
    - Which is most of the time

COPY, solved

commit d326d9e8ea1d690cf6d968000efaa5121206d231
Author: Heikki Linnakangas <heikki.linnakangas@iki.fi>
Date: Wed Nov 9 10:54:41 2011 +0200

In COPY, **insert tuples to the heap in batches**.

This greatly reduces the WAL volume, especially when the table is narrow. The overhead of locking the heap page is also reduced. Reduced WAL traffic also makes it scale a lot better, if you run multiple COPY processes at the same time.


COPY throughput
9.2devel (4c6cedd1b)
without COPY patch
1000x rows / sec
# of parallel processes
0
500
1000
1500
2000
2500
3000
0
5
10
15
20
25
30
35

Impact

- Makes bulk loading scale
- Reduces WAL volume
- Caveats:
  - Optimization does not apply if there are BEFORE/AFTER triggers or volatile DEFAULT expressions
  - When loading into a single table, extending the file becomes bottleneck

Next workload: SELECTs

- I said SELECTs already scale well
  - But in 9.1, only if you SELECTed different tables
  - PostgreSQL lock manager is partitioned
  - But when all backends hit the same table, that doesn't help, and the lock manager became a bottleneck

Lock manager, solved

commit 3cba8999b343648c4c528432ab3d51400194e93b
Author: Robert Haas <rhaas@postgresql.org>
Date: Sat May 28 19:52:00 2011 -0400

Create a "fast path" for acquiring weak relation locks.

When an AccessShareLock, RowShareLock, or RowExclusiveLock is requested on an unshared database relation, and we can verify that no conflicting locks can possibly be present, record the lock in a per-backend queue, stored within the PGPROC, rather than in the primary lock table. This eliminates a great deal of contention on the lock manager LWLocks.

...
Review by Jeff Davis.

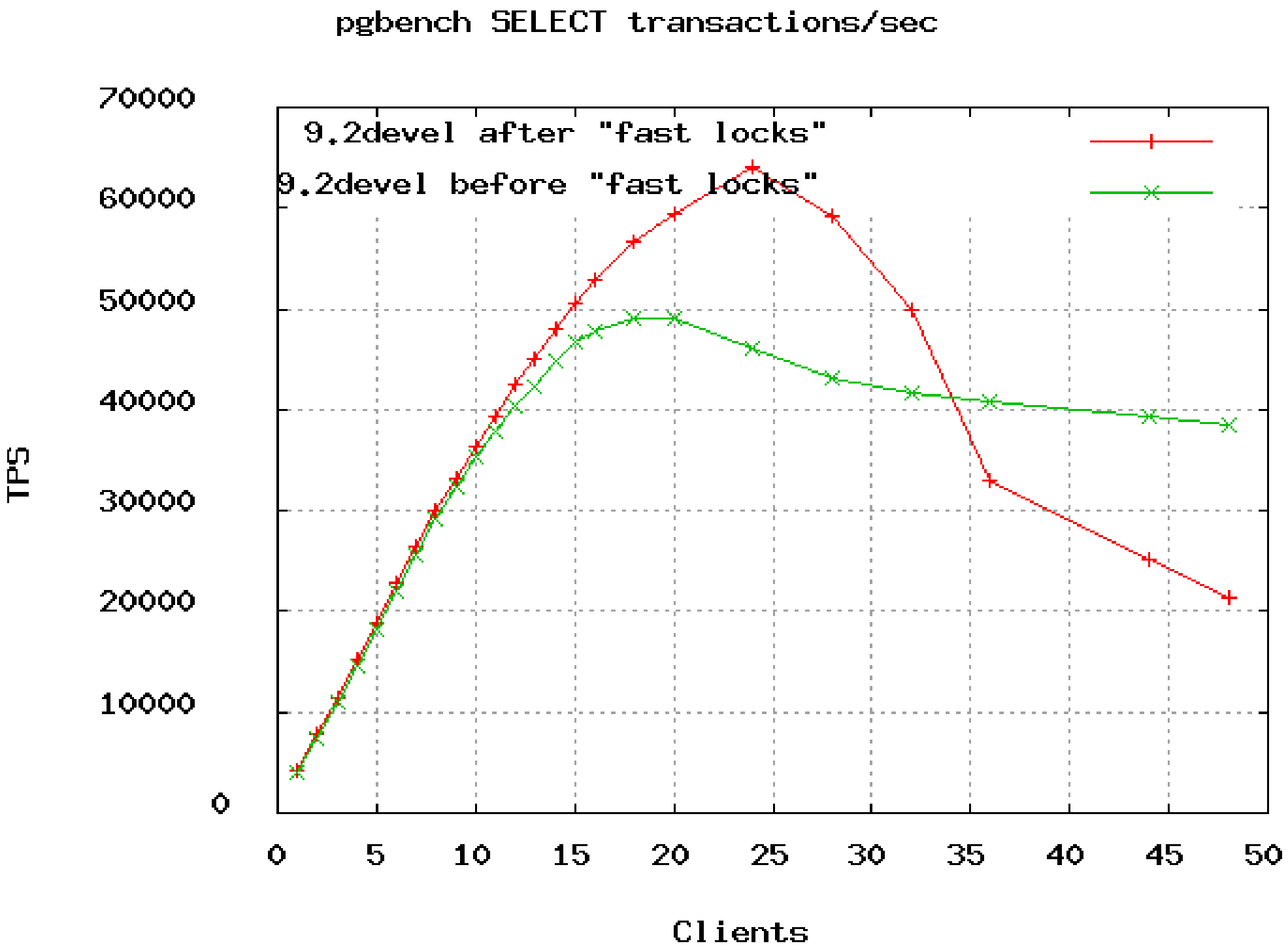
Impact

”Here are the results of alternating runs without
and with the patch on that machine:

tps = **36291**.996228 (including connections establishing)
tps = **129242**.054578 (including connections establishing)
tps = 36704.393055 (including connections establishing)
tps = 128998.648106 (including connections establishing)
tps = 36531.208898 (including connections establishing)
tps = 131341.367344 (including connections establishing)

That's an improvement of about ~3.5x. According to the vmstat output,
when running without the patch, the CPU state was about 40% idle.
With the patch, it dropped down to around 6%.

- Robert Haas, 3 Jun 2011


pgbench SELECT transactions/sec
9.2devel after "fast locks"
9.2devel before "fast locks"
TPS
Clients
0
10000
20000
30000
40000
50000
60000
70000
0
5
10
15
20
25
30
35
40
45
50

SELECTs continued: ProcArrayLock

- Each session has an entry in shared memory, in the ”proc array”
- The proc array is protected by a lock called ProcArrayLock
- It is acquired in shared mode whenever a snapshot is taken (~= at the beginning of each transaction)
- It is acquired in exclusive mode whenever a transaction commits

ProcArrayLock

- Becomes a bottleneck at high transaction rates
  - A problem with OLTP workloads with a lot of small transactions
  - Not a problem with larger transactions that do more stuff per transaction

commit b4fbe392f8ff6ff1a66b488eb7197eef9e1770a4
Author: Robert Haas <rhaas@postgresql.org>
Date: Fri Jul 29 16:46:13 2011 -0400

Reduce sinval synchronization overhead.

Testing shows that the overhead of acquiring and releasing
SInvalReadLock and msgNumLock on high-core count boxes can waste a lot
of CPU time and hurt performance. This patch adds a per-backend flag
that allows us to skip all that locking in most cases. Further
testing shows that this improves performance even when sinval traffic
is very high.

Patch by me. Review and testing by Noah Misch.

commit 84e37126770dd6de903dad88ce150a49b63b5ef9
Author: Robert Haas <rhaas@postgresql.org>
Date: Thu Aug 4 12:38:33 2011 -0400

Create VXID locks "lazily" in the main lock table.

Instead of entering them on transaction startup, we materialize them
only when someone wants to wait, which will occur only during CREATE
INDEX CONCURRENTLY. In Hot Standby mode, the startup process must also
be able to probe for conflicting VXID locks, but the lock need never be
fully materialized, because the startup process does not use the normal
lock wait mechanism. Since most VXID locks never need to touch the
lock manager partition locks, this can significantly reduce blocking
contention on read-heavy workloads.

Patch by me. Review by Jeff Davis.

spinlocks

commit c01c25fbe525869fa81237954727e1eb4b7d4a14
Author: Robert Haas <rhaas@postgresql.org>
Date: Mon Aug 29 10:05:48 2011 -0400

Improve spinlock performance for HP-UX, ia64, non-gcc.

At least on this architecture, it's very important to spin on a
non-atomic instruction and only retry the atomic once it appears
that it will succeed. To fix this, split TAS() into two macros:
TAS(), for trying to grab the lock the first time, and TAS_SPIN(),
for spinning until we get it. TAS_SPIN() defaults to same as TAS(),
but we can override it when we know there's a better way.

It's likely that some of the other cases in s_lock.h require
similar treatment, but this is the only one we've got conclusive
evidence for at present.

commit ed0b409d22346b1b027a4c2099ca66984d94b6dd
Author: Robert Haas <rhaas@postgresql.org>
Date: Fri Nov 25 08:02:10 2011 -0500

Move "hot" members of PGPROC into a separate PGXACT array.

This speeds up snapshot-taking and reduces ProcArrayLock contention.
Also, the PGPROC (and PGXACT) structures used by two-phase commit are
now allocated as part of the main array, rather than in a separate
array, and we keep ProcArray sorted in pointer order. These changes
are intended to minimize the number of cache lines that must be pulled
in to take a snapshot, and testing shows a substantial increase in
performance on both read and write workloads at high concurrencies.

Pavan Deolasee, Heikki Linnakangas, Robert Haas

commit 0d76b60db4684d3487223b003833828fe9655fe2
Author: Robert Haas <rhaas@postgresql.org>
Date: Fri Dec 16 21:44:26 2011 -0500

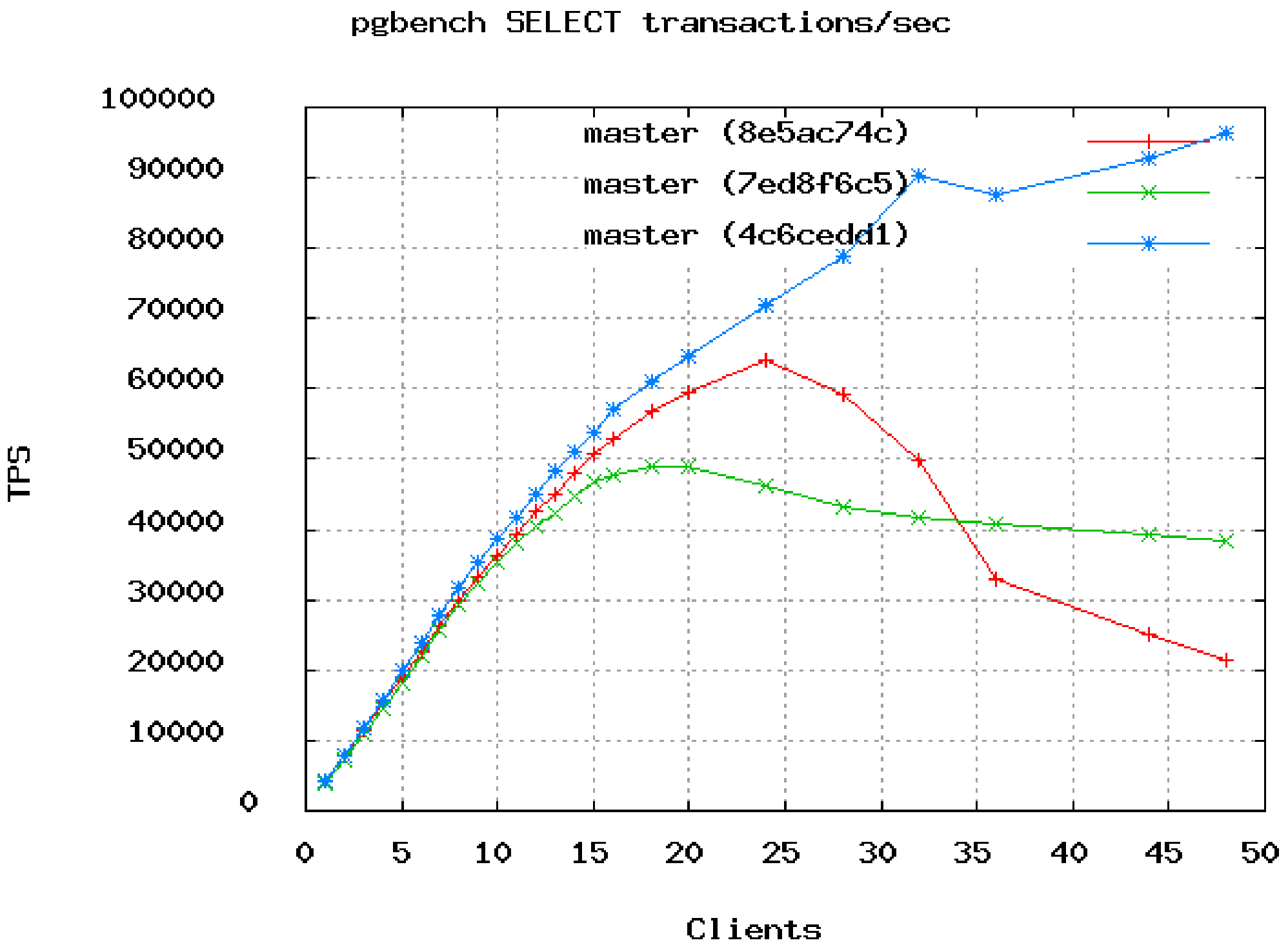
Various micro-optimizations for GetSnapshopData().

Heikki Linnakangas had the idea of rearranging GetSnapshotData to avoid checking for sub-XIDs when no top-level XID is present. This patch does that plus further a bit of further, related rearrangement. Benchmarking show a significant improvement on unlogged tables at higher concurrency levels, and mostly indifferent result on permanent tables (which are presumably bottlenecked elsewhere). Most of the benefit seems to come from using the new NormalTransactionIdPrecedes() macro rather than the function call TransactionIdPrecedes().

commit d573e239f03506920938bf0be56c868d9c3416da
Author: Robert Haas <rhaas@postgresql.org>
Date: Wed Dec 21 09:16:55 2011 -0500

    Take fewer snapshots.

    When a PORTAL_ONE_SELECT query is executed, we can opportunistically
    reuse the parse/plan shot for the execution phase. This cuts down the
    number of snapshots per simple query from 2 to 1 for the simple
    protocol, and 3 to 2 for the extended protocol. Since we are only
    reusing a snapshot taken early in the processing of the same protocol
    message, the change shouldn't be user-visible, except that the remote
    possibility of the planning and execution snapshots being different is
    eliminated.
    ...
    Patch by me; review by Dimitri Fontaine.


pgbench SELECT transactions/sec
master (8e5ac74c)
master (7ed8f6c5)
master (4c6cedd1)
TPS
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
5
10
15
20
25
30
35
40
45
50
Clients

Next bottleneck

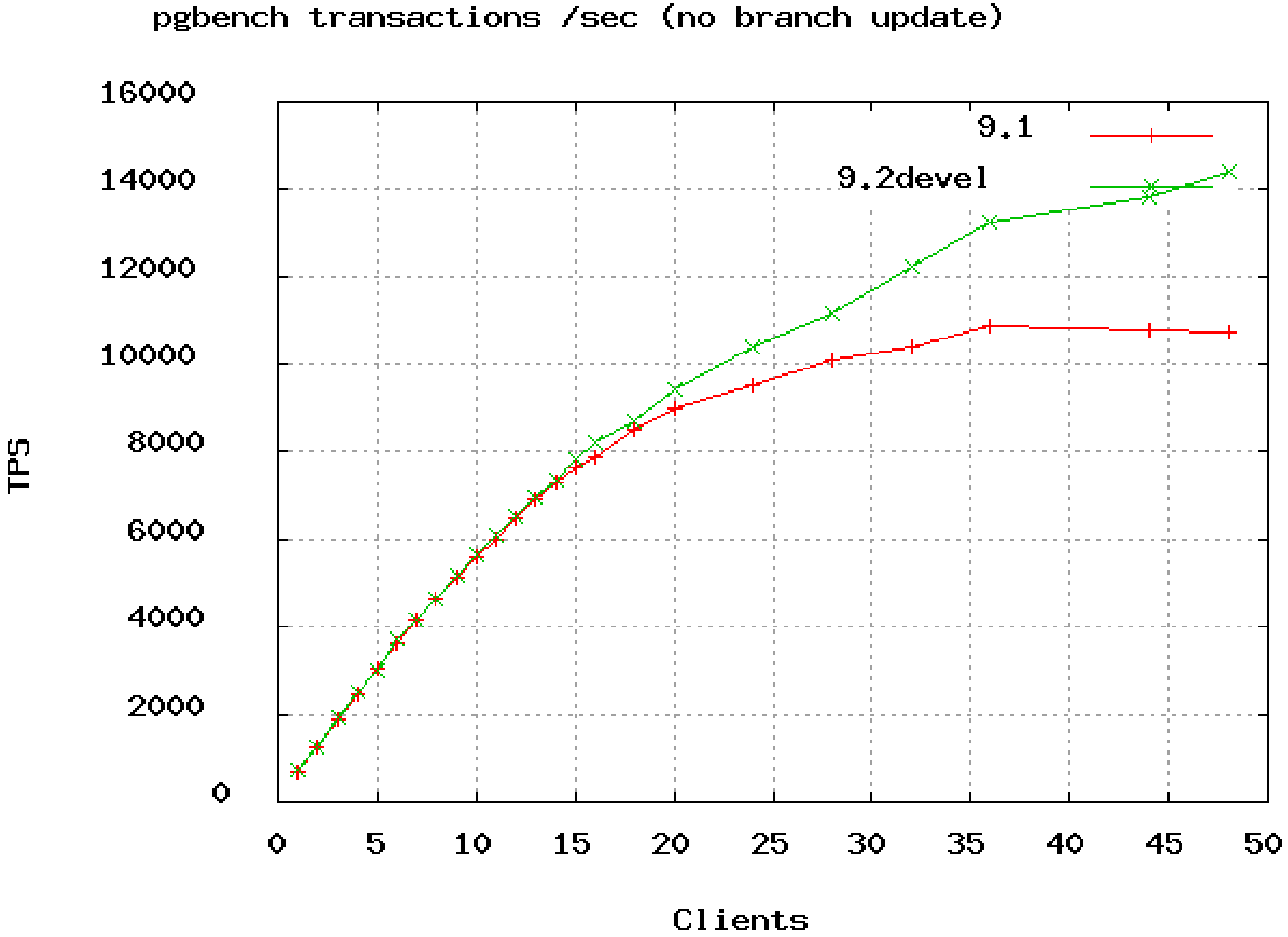
”So I have a new theory: on permanent

tables, *anything* that reduces ProcArrayLock contention causes an approximately equal increase in WALInsertLock contention (or maybe some other lock), and in some cases that increase in contention elsewhere can cost more than the amount we're saving here.”

- Robert Haas, 15 Dec 2011

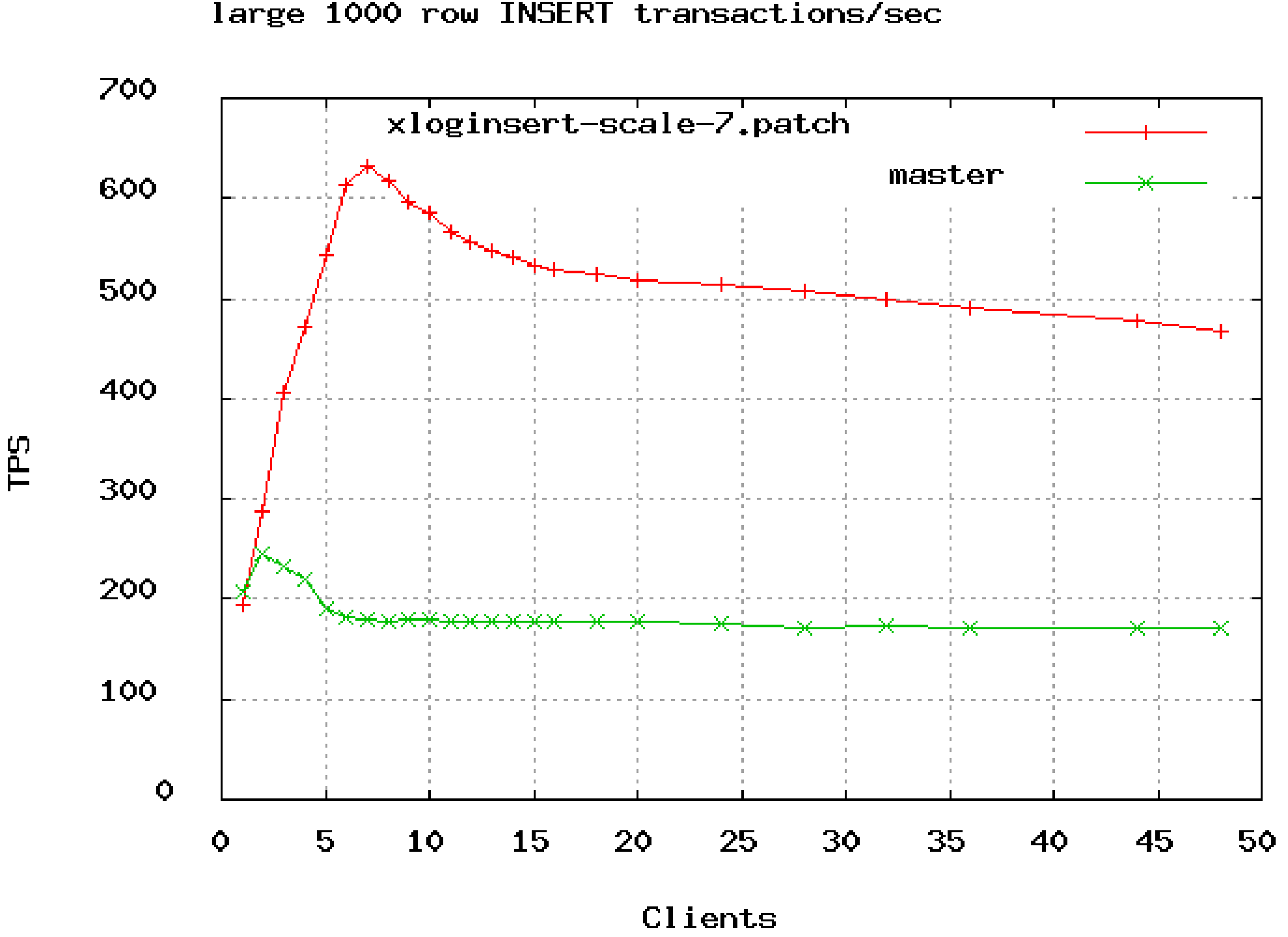
Next workload: read/write

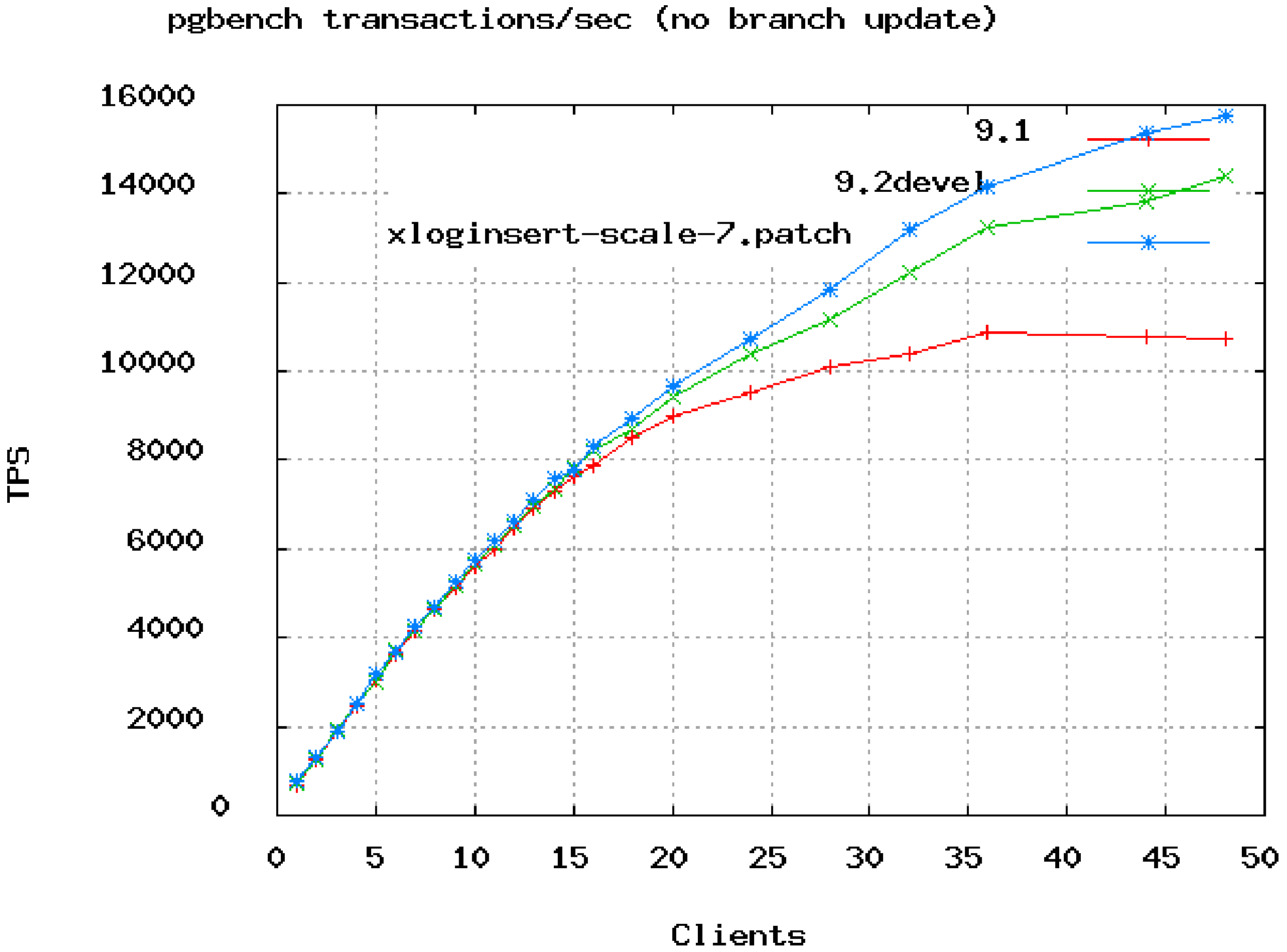
- Reran pgbench, now with writes
  - Excluding branch-table updates, to avoid bottlenecking on the application level


pgbench transactions /sec (no branch update)
16000
14000
12000
10000
8000
6000
4000
2000
0
TPS
0
5
10
15
20
25
30
35
40
45
50
Clients
9.1
9.2devel

WALInsertLock

- Remember the COPY problem?
- The patch to solve that was a special hack targeting just COPY.
- The problem remains for all other inserts/updates/deletes


large 1000 row INSERT transactions/sec
xloginsert-scale-7.patch
master
TPS
Clients
0
100
200
300
400
500
600
700
0
5
10
15
20
25
30
35
40
45
50


pgbench transactions/sec (no branch update)
16000
14000
12000
10000
8000
6000
4000
2000
0
TPS
0
5
10
15
20
25
30
35
40
45
50
Clients
9.1
9.2devel
xloginsert-scale-7.patch

Profiling

- HP-UX has a nice tool called *caliper*
  - Like oprofile, but can include wait times too

```
            34.62          postgres::ProcArrayEndTransaction [51]
            30.77          postgres::XLogInsert [49]
            11.54          postgres::LockBuffer [113]
             7.69          postgres::TransactionIdSetPageStatus [146]
             7.69          postgres::BufferAlloc [72]
             7.69          postgres::GetSnapshotData [246]
[25]    13.6        1.1       12.5         3.85
postgres::LWLockAcquire
            69.23          postgres::PGSemaphoreLock [38]
            26.92          postgres::s_lock [60]
```

Summary

- 9.2 scales much better for many common workloads!
- Future focus
  - Serializable transactions
  - WAL-logging
- Thanks to
  - Nathan boley, for lending a server for benchmarking
  - Greg Smith, for creating pgbench-tools