Mechanical Sympathy for Elephants

Reducing I/O and memory stalls

Thomas Munro, PGCon 2020
thomas.munro@microsoft.com
tmunro@postgresql.org
tmunro@freebsd.org

Talk structure

- I/O
  - Prefetching opportunities
  - Proposal: Prefetching in recovery
- Memory
  - Partitioning vs cache size
  - Experimental work: Prefetching in hash joins

I/O

Three kinds of predictions about future access

1. You’ll probably want recently and frequently accessed data again soon; that’s why we have caches
2. If you’re accessing blocks in physically sequential order, you’ll probably keep doing that
   - Larger read/write sizes possible
   - I/O can be completed before we need it
   - Automatic prefetching exists at many levels
3. More complex access patterns typically require case-specific magic with high level knowledge of pointers within the data

Limited I/O prediction used by PostgreSQL today

- Sequential scans rely on kernel read-ahead for good performance
  - To support direct I/O we'll have to do that explicitly one of these days
- Bitmap Heap Scan issues explicit hints
  - Used for brin and bloom indexes and AND/OR multi-index scans
  - Calls `PrefetchBuffer()` up to `effective_io_concurrency` blocks ahead of `ReadBuffer()` using the bitmap of interesting blocks
- VACUUM issues some explicit hints
  - Calls `PrefetchBuffer()` for up to `maintenance_io_concurrency` blocks
- Linux only: we control write back rate with `sync_file_range()`

Side note: posix_fadvise() v true async I/O

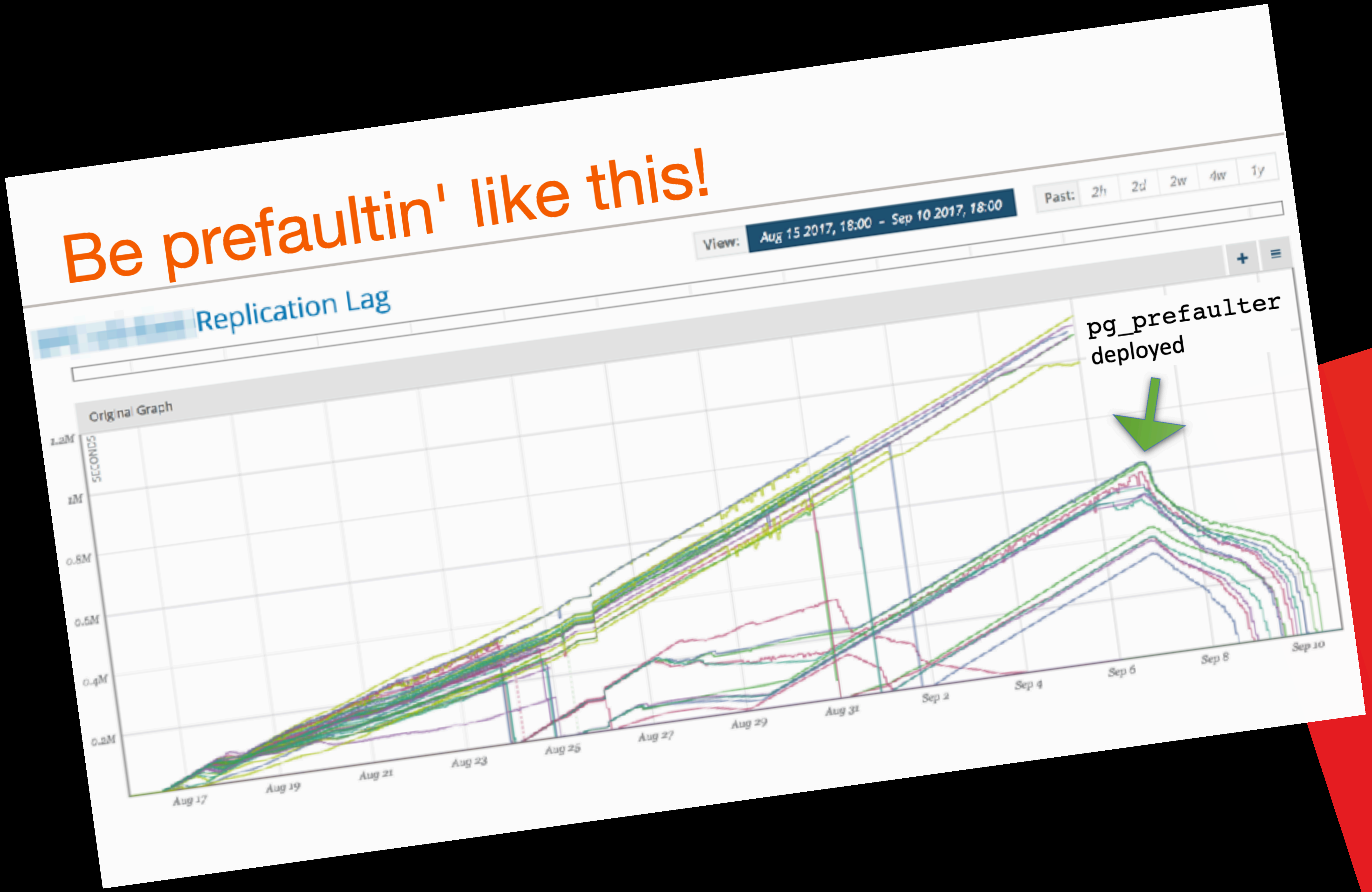
- `PrefetchBuffer()` currently calls `posix_fadvise(POSIX_FADV_WILLNEED)` as a hint to the kernel that you will soon be reading a certain range of a file, that it can use to prefetch the relevant data asyncronously so that a future `pread()` call hopefully doesn't block.
- As far as I know, it only actually does something on Linux and NetBSD today. Even there, it doesn't work on ZFS (yet).
- Work is being done to introduce real asynchronous I/O to PostgreSQL. For more on that, see Andres Freund's PGCon 2020 talk.
- `PrefetchBuffer()` or a similar function will probably still be called to initiate that, it'll just that the data will travel all the way into PostgreSQL's buffers, not just kernel buffers. So the case-specific logic to know *when* to call `PrefetchBuffer()` is mostly orthogonal still needs to be done either way.

More opportunities to predict I/O

- Sometimes the kernel heuristics don’t detect sequential access:
  - 1GB segment file boundaries (seq scan, spill files for hash, sort, CTE, …)
  - Interleaving reads and writes to the same fd (VACUUM, hint bit writeback)
  - Parallel Sequential Scan (multiple processes stepping through a file)
- While scanning btree, gin, gist without a Bitmap Heap Scan
  - Next btree page, referenced heap pages, visibility map
- Future keys in a nested loop join (“block nest loop join” with prefetch)
- While replaying the WAL on a streaming replica or after a crash, we know exactly which blocks we’ll be accessing: it’s in the WAL

Inspiration: pg_prefaulter

Presented by Sean Chittenden, PGCon 2018



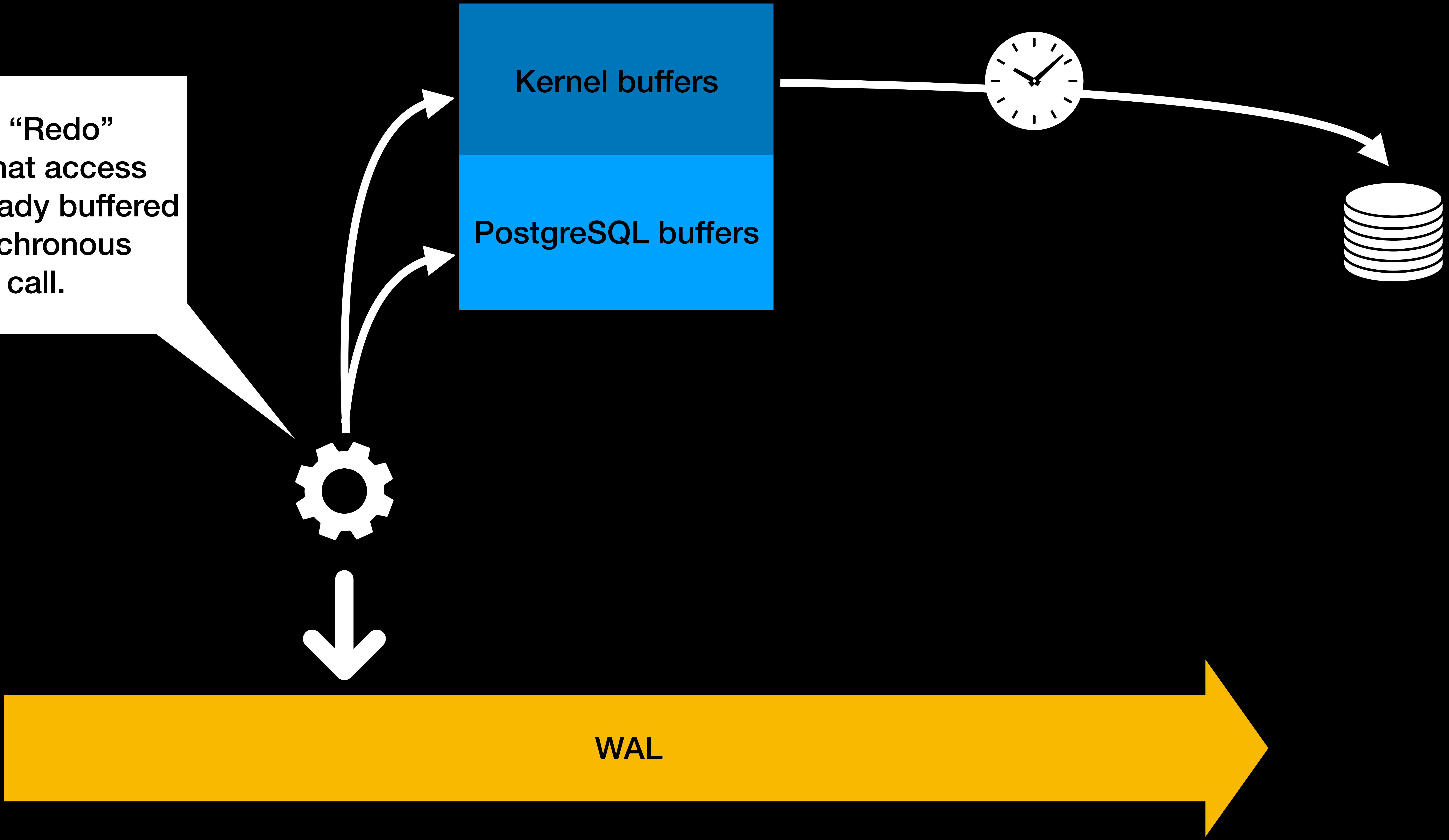
Scaling WAL Performance
Eliminate replication lag and reduce startup times with pg_prefaulter
Joyent

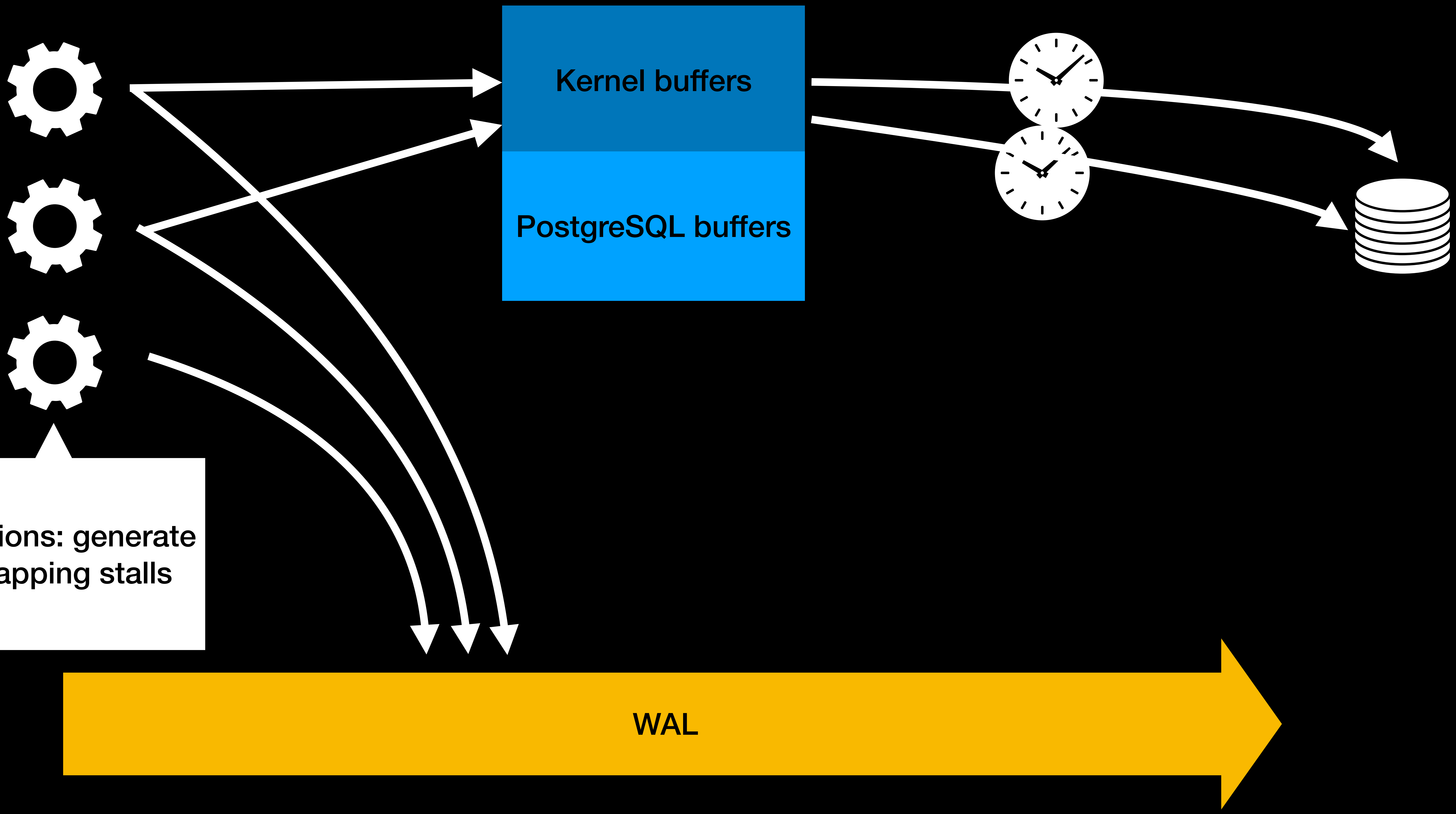
“Physiological” logging

Logical changes within pages, but physical references to pages

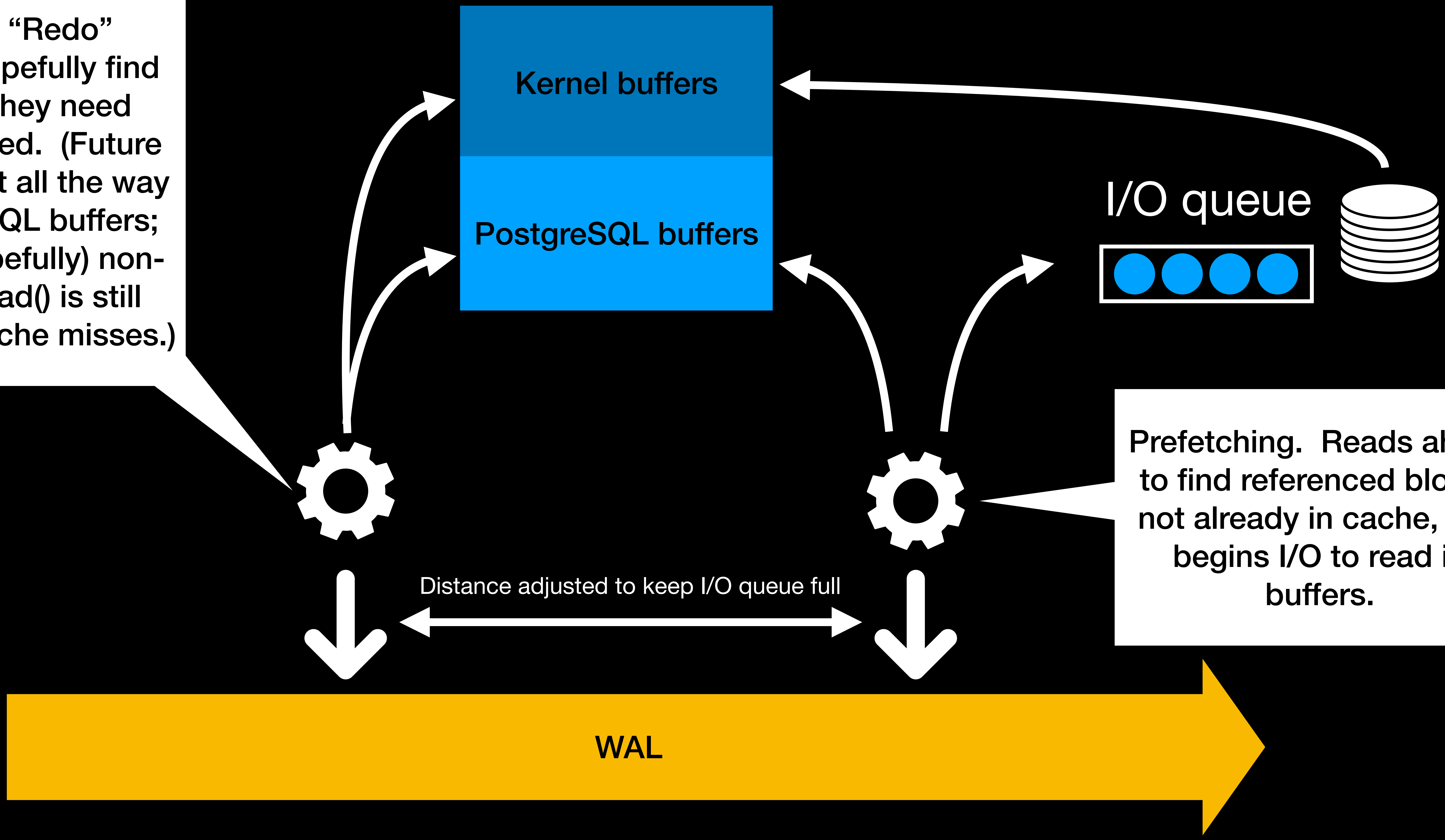
```
postgres=# insert into t values (1234), (4321);
INSERT 0 2

$ pg_waldump pgdata/pg_wal/000000010000000000000001
[output abridged]
rmgr: Heap        lsn: 0/015B8F48 desc: INSERT off 5 flags 0x00,  blkref #0: rel 1663/12923/24587 blk 0
rmgr: Btree       lsn: 0/015B8F88 desc: INSERT_LEAF off 4,        blkref #0: rel 1663/12923/24590 blk 1
rmgr: Heap        lsn: 0/015B8FC8 desc: INSERT off 6 flags 0x00,  blkref #0: rel 1663/12923/24587 blk 0
rmgr: Btree       lsn: 0/015B9008 desc: INSERT_LEAF off 5,        blkref #0: rel 1663/12923/24590 blk 1
rmgr: Transaction lsn: 0/015B9048 desc: COMMIT
```

Recovery. "Redo" operations that access blocks not already buffered make a synchronous pread() call.





Primary sessions: generate many overlapping stalls



Recovery. "Redo" operations hopefully find everything they need already buffered. (Future plans will get it all the way into PostgreSQL buffers; for now a (hopefully) non-sleeping pread() is still required for cache misses.)

Prefetching. Reads ahead to find referenced blocks not already in cache, and begins I/O to read in buffers.

User interface

As of most recent patch — details likely to change!

- `maintenance_io_concurrency`: defaulting to 10
- `max_recovery_prefetch_distance`: defaulting to 256kB (-1 = disable)

```
postgres=# select * from pg_stat_prefetch_recovery ;
-[ RECORD 1 ]---+------------------------------
stats_reset     | 2020-05-21 21:13:30.950423+12
prefetch        | 46091
skip_hit        | 154285
skip_new        | 995
skip_fpw        | 58445
skip_seq        | 10686
distance        | 144200
queue_depth     | 10
avg_distance    | 62077.297
avg_queue_depth | 5.2426248
```

Blocks prefetched so far

Blocks not prefetched (various reasons)

Current number of prefetches in flight

pgbench time

Scale 2000, 16GB RAM, 5000 IOPS cloud storage, -c16 -j16

```
iostat -x:   r/s     rkB/s    aqu-sz
====================================
Primary:    3466  34088.00     16.80
Replica:     250   2216.00      1.09 -> falls behind

maintenance_io_concurrency settings:
iostat -x:   r/s     rkB/s    aqu-sz
====================================
Replica-10: 1143   6088.00      6.80
Replica-20: 2170  17816.00     12.83
Replica-50: 4887  40024.00     33.00 -> keeps up
```

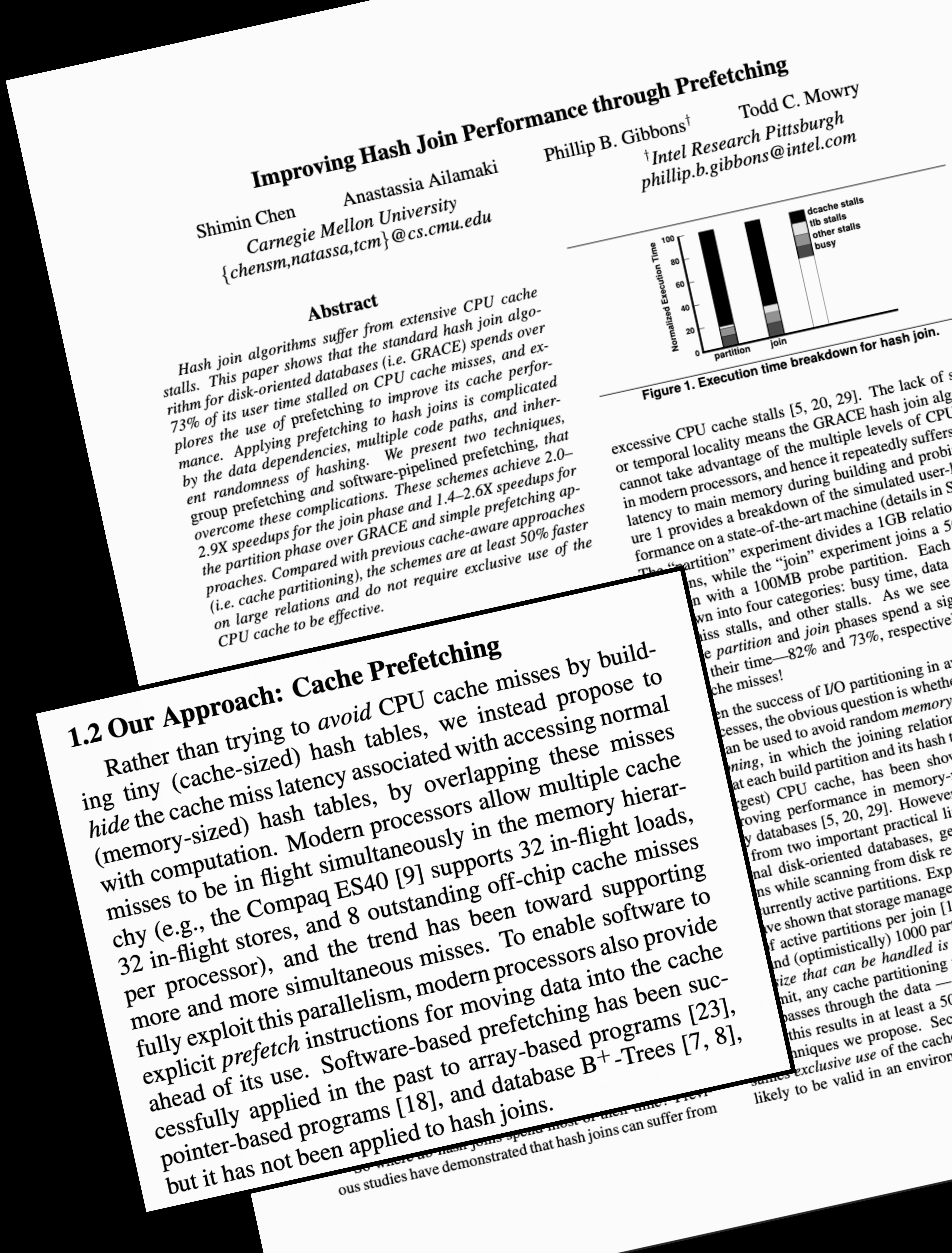
Problems

- Works best with full_page_writes=off, because FPW avoids the need for reads!
- Also works with FPWs, with infrequent checkpoints (fewer FPWs).
- Also works well for systems with storage page size > PostgreSQL's (Joyent's large ZFS records), even with FPW, due to read-before-write.
- Would be useful for FPW if we adopted an idea proposed on pgsql-hackers to read and trust pages whose checksum passes (consider them non-torn); such pages may have a high LSN and allow us to skip applying a bunch of WAL.
- Currently reads and decodes records an extra time while prefetching. Also probes the buffer mapping table an extra time. Fixable.

Memory

Prefetching hash joins

- Hash joins produce high rates of data cache misses while building and probing large hash tables.
- “Improving hash Join Performance through Prefetching” claims up to 73% of time is spent in data cache stalls.
- PostgreSQL suffers from this effect quite measurably.



Improving Hash Join Performance through Prefetching

Shimin Chen Anastassia Ailamaki Phillip B. Gibbons[†] Todd C. Mowry
Carnegie Mellon University [†]*Intel Research Pittsburgh*
{*chensm,natassa,tcm*}*@cs.cmu.edu* *phillip.b.gibbons@intel.com*

Abstract

Hash join algorithms suffer from extensive CPU cache stalls. This paper shows that the standard hash join algorithm for disk-oriented databases (i.e. GRACE) spends over 73% of its user time stalled on CPU cache misses, and explores the use of prefetching to improve its cache performance. Applying prefetching to hash joins is complicated by the data dependencies, multiple code paths, and inherent randomness of hashing. We present two techniques, group prefetching and software-pipelined prefetching, that overcome these complications. These schemes achieve 2.0–2.9X speedups for the join phase and 1.4–2.6X speedups for the partition phase over GRACE and simple prefetching approaches. Compared with previous cache-aware approaches (i.e. cache partitioning), the schemes are at least 50% faster on large relations and do not require exclusive use of the CPU cache to be effective.

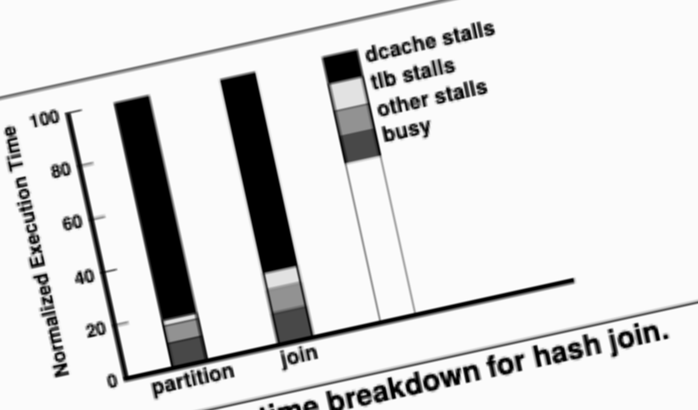


Figure 1. Execution time breakdown for hash join.

excessive CPU cache stalls [5, 20, 29]. The lack of s
or temporal locality means the GRACE hash join alg
cannot take advantage of the multiple levels of CPU
in modern processors, and hence it repeatedly suffers
latency to main memory during building and probi
ure 1 provides a breakdown of the simulated user-l
formance on a state-of-the-art machine (details in S
The “partition” experiment divides a 1GB relatio
ns, while the “join” experiment joins a 5
n with a 100MB probe partition. Each
wn into four categories: busy time, data
iss stalls, and other stalls. As we see
e *partition* and *join* phases spend a sig
their time—82% and 73%, respectivel
che misses!

en the success of I/O partitioning in a
cesses, the obvious question is whethe
an be used to avoid random *memory*
ning, in which the joining relation
at each build partition and its hash t
rgest) CPU cache, has been show
roving performance in memory-
y databases [5, 20, 29]. However
from two important practical li
nal disk-oriented databases, ge
ns while scanning from disk re
urrently active partitions. Exp
ve shown that storage manage
f active partitions per join [1
nd (optimistically) 1000 par
size that can be handled is
mit, any cache partitioning
passes through the data —
this results in at least a 50
hniques we propose. Sec
sumes *exclusive use* of the cach
likely to be valid in an environ

ous studies have demonstrated that hash joins can suffer from

1.2 Our Approach: Cache Prefetching

Rather than trying to *avoid* CPU cache misses by building tiny (cache-sized) hash tables, we instead propose to *hide* the cache miss latency associated with accessing normal (memory-sized) hash tables, by overlapping these misses with computation. Modern processors allow multiple cache misses to be in flight simultaneously in the memory hierarchy (e.g., the Compaq ES40 [9] supports 32 in-flight loads, 32 in-flight stores, and 8 outstanding off-chip cache misses per processor), and the trend has been toward supporting more and more simultaneous misses. To enable software to fully exploit this parallelism, modern processors also provide explicit *prefetch* instructions for moving data into the cache ahead of its use. Software-based prefetching has been successfully applied in the past to array-based programs [23], pointer-based programs [18], and database B$^+$-Trees [7, 8], but it has not been applied to hash joins.

Hash table vs cache hierarchy

- Partitioning the hash table so that it fits in L3 cache helps avoid cache misses, but…
- L3 cache is shared with other cores that could be doing unrelated work, and other executor nodes in our own plan!
- Cache-limited hash table means potentially large numbers of partitions, whose buffers become too large and random at some point.

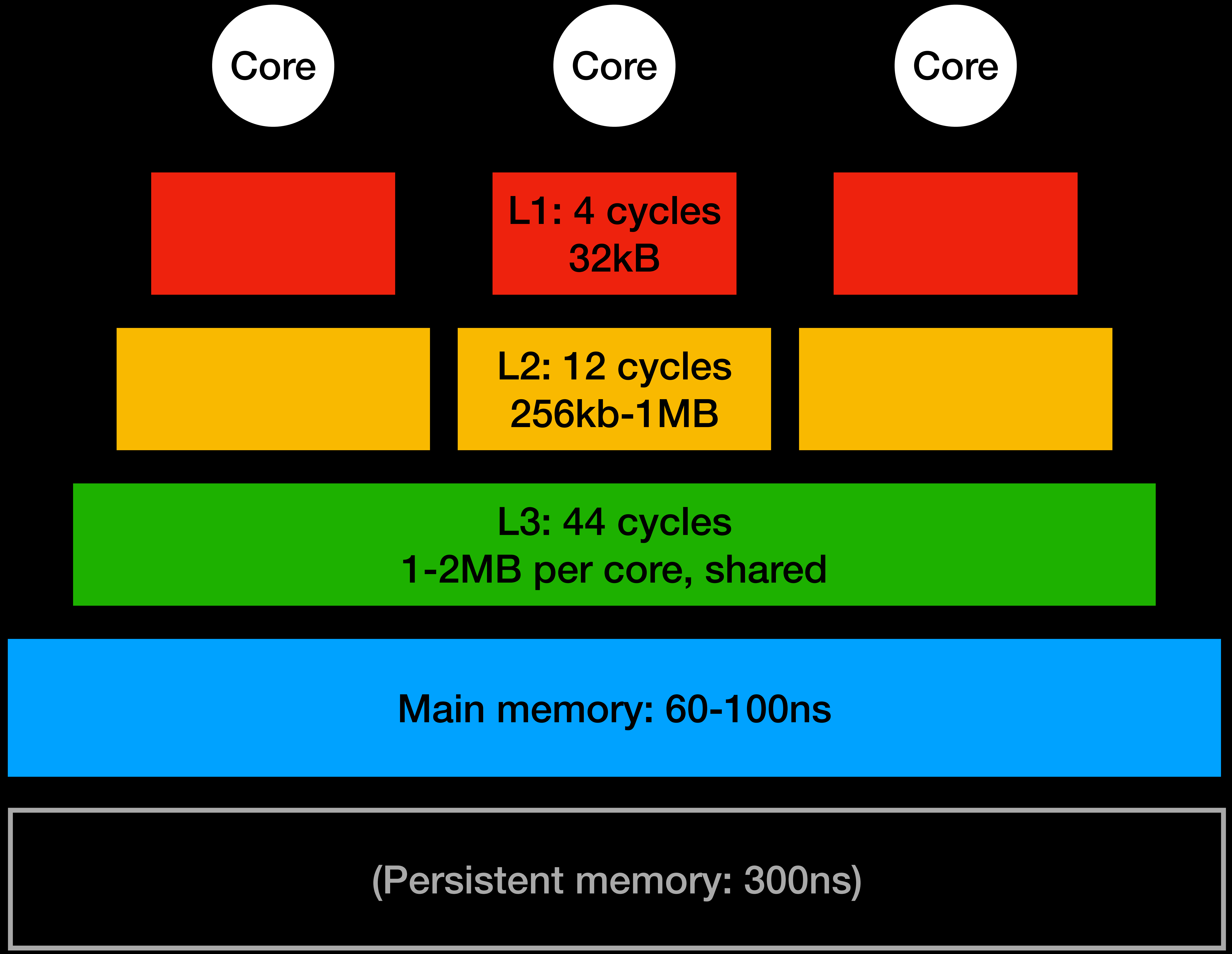


*illustration only, actual details vary enormously

Software prefetching

- Modern ISAs have some kind of PREFETCH instruction that initiates a load of a cache line at a given address into the L1 cache. (Compare “hardware” prefetching, based on sequential access heuristics, and much more complex voodoo for instructions.)
- Sprinkling it around simple pointer-chasing scenarios where you can’t get far enough ahead is a bad plan. See Linux experience (link at end), which concluded: “prefetches are absolutely toxic, even if the NULL ones are excluded”
- Can we get far enough ahead of a hash join insertion? Yes!
- Can we get far enough ahead of a hash join probe? Also yes! But with more architectural struggle.

Hash table vs L3 cache

- We can see the L3 cache size friendliness, when running in isolation.
- Software prefetching can avoid (“hide”) these misses through parallelism.
- Note: 4.2->4.0, even with similar LLC misses! Due to nearer caches + code reordering.

```
create table t as select generate_series(1, 10000000)::int i;
select pg_prewarm('t');
set max_parallel_workers_per_gather = 0;

set work_mem = '4MB';
select count(*) from t t1 join t t2 using (i);

  Buckets: 131072  Batches: 256  Memory Usage: 2400kB
  master:  Time: 4242.639 ms (00:04.243),  6,149,869 LLC-misses
  patched: Time: 4033.288 ms (00:04.033),  6,270,607 LLC-misses

set work_mem = '1GB';
select count(*) from t t1 join t t2 using (i);

  Buckets: 16777216  Batches: 1  Memory Usage: 482635kB
  master:  Time: 5879.607 ms (00:05.880), 28,380,743 LLC-misses
  patched: Time: 2728.749 ms (00:02.729),  2,487,565 LLC-misses
```

Algorithm changes

- Build phase
    - Push pointers to tuples + bucket number into an “insert buffer”, rather than inserting directly.
    - When the buffer is full, PREFETCH all the buckets, and then insert all the tuples.
    - Small gain even with the PREFETCH disabled, just from giving the CPU more leeway to reorder execution.
- Probe phase
    - Copy a small number of outer tuples into a “probe buffer” of extra slots. Refill when empty. These will be used for probing. It would be nice if there were a cheap way to “move” tuples without materialising them; the memory management problems involved look a bit tricky.
    - Calculated hash values for all the tuples in one go, then PREFETCH the hash buckets, then PREFETCH the first tuples.
    - While scanning buckets, fetch the next item in the chain (but no NULL) before we emit a tuple.
    - Other strategies are possible (something more pipelined and less batched might reduce competition for cache lines in nested hash joins).

References

- Patches for prefetching in recovery:
  https://commitfest.postgresql.org/28/2410/
- Thread about hash join prefetching :
  https://www.postgresql.org/message-id/flat/CAEepm%3D2y9HM9QP%2BHhRZdQ3pU6FShSMyu%3DV1uHXhQ5gG-dketHg%40mail.gmail.com
- Sean Chittenden's pg_prefaulter talk:
  https://www.pgcon.org/2018/schedule/track/Case%20Studies/1204.en.html
- Improving Hash Join Performance through Prefetching (Chen, Ailamaki, Gibbons, Mowry):
  https://www.cs.cmu.edu/~chensm/papers/hashjoin_icde04.pdf
- The Problem with Prefetch [in certain Linux macros]:
  https://lwn.net/Articles/444336/
- Martin Thompson's blog (inspiration for this talk's title):
  https://mechanical-sympathy.blogspot.com/