PostgreSQL on FreeBSD

Some news, observations and speculation

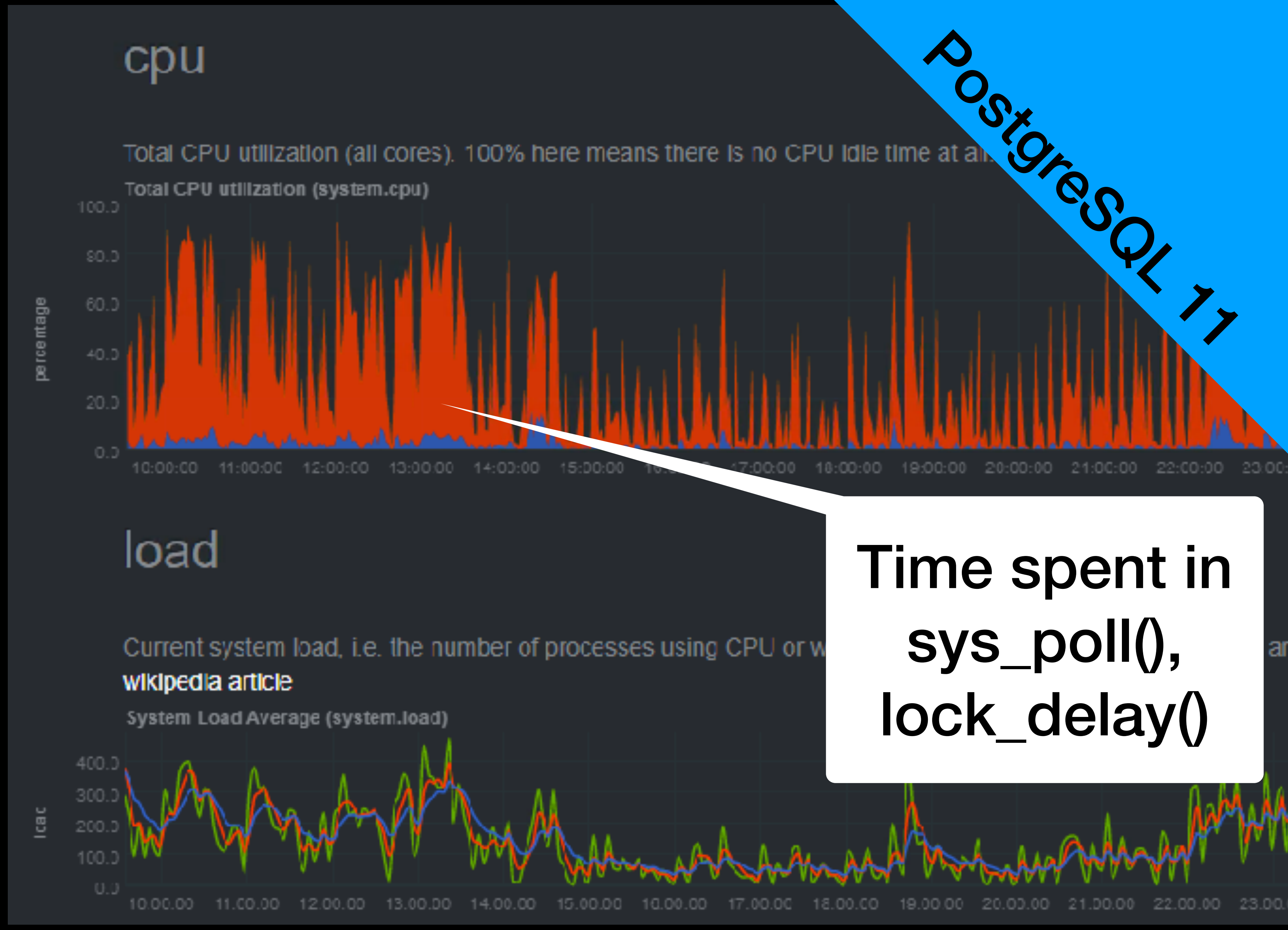
Thomas Munro, BSDCan 2020
thomas.munro@microsoft.com
tmunro@postgresql.org
tmunro@freebsd.org

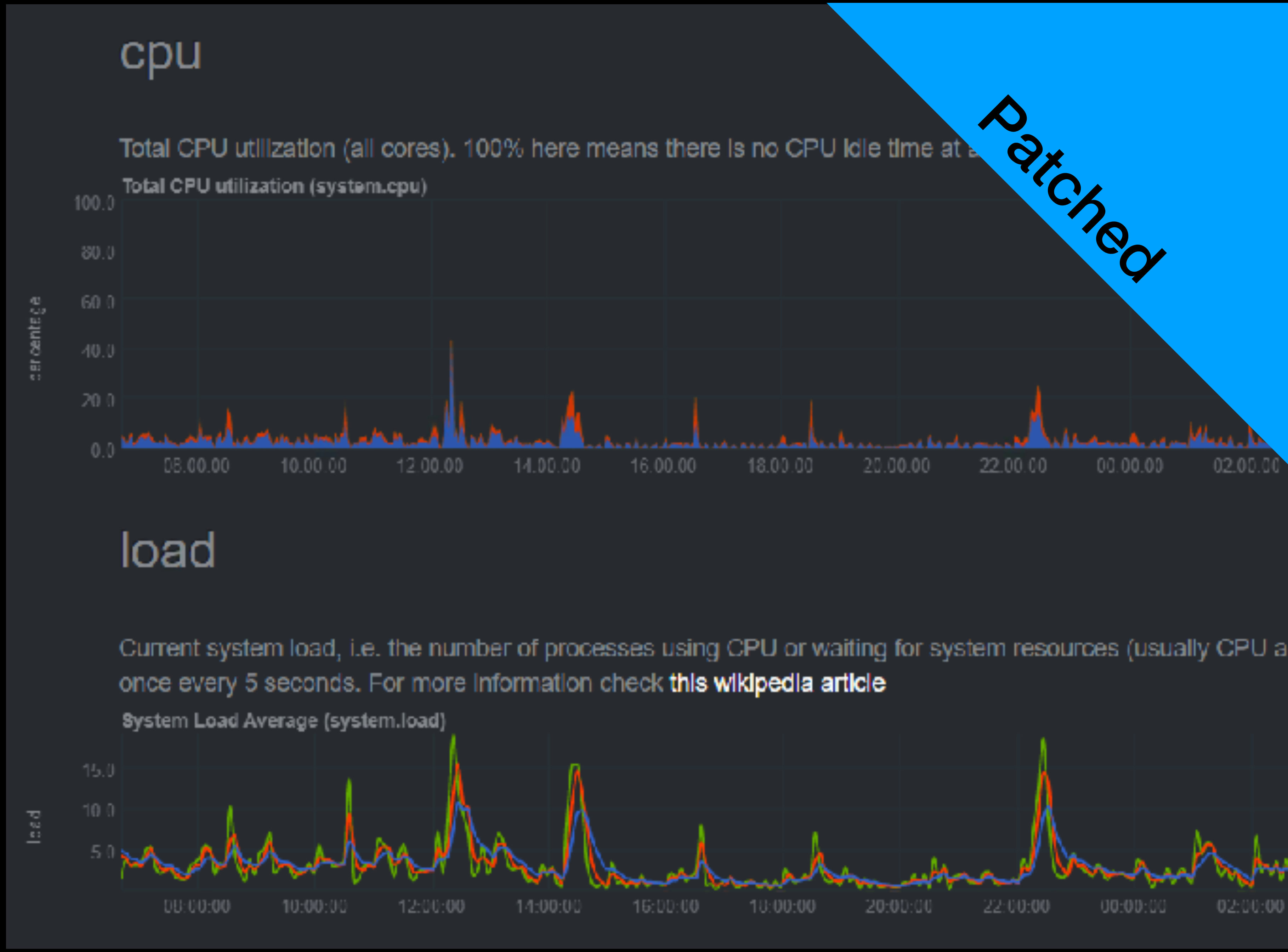
Show & Tell — Recent(-ish) work

PostgreSQL 13: kqueue(2)

Replaces poll(2) for multiplexing waits

- Like similar improvements from adopting epoll(2) on Linux.
- Dramatically reduces system time on high core count systems with many active connections (= processes).
- Mostly due to contention on a pipe inherited by every process, used to control emergency shutdown on postmaster (parent process) exit without leaking processes everywhere.
- Very useful also for future work on shared queries that multiplex large numbers of sockets (and maybe more).





Graphs courtesy of Rui DeSousa; hw.ncpu=88

PostgreSQL 12, FreeBSD 11.2: PROC_PDEATHSIG_CTL

- We also have non-waiting CPU-bound code paths that want timely notification that the database cluster is shutting down unexpectedly: mainly the crash recovery loop, also used on read-only streaming servers.
- Previously we regularly called `read()` on that contended pipe these CPU bound loops! System calls aren't getting cheaper… this turned out to be wasting up to 15% of crash recovery time. We could probably improve that by simply polling less frequently, but… better idea:
- Brief history of mechanisms to get a signal when your parent process exits:
  - IRIX: `prctl(PR_TERMCHILD)` requests `SIGHUP`
  - Linux 2.1.57: `prctl(PR_SET_PDEATHSIG, signo)`
  - FreeBSD 11.2: `procctl(P_PID, 0, PROC_PDEATHSIG_CTL, &signo)`

PostgreSQL 12, FreeBSD 12: setproctitle_fast(3)

- PostgreSQL updates the process title frequently to show what it's doing; administrators like it, and expect it to be enabled when coming from other OSes.
- This was always a performance problem on FreeBSD but not other systems, because FreeBSD's `setproctitle(3)` made 2 system calls. The port turns it off by default.
- `setproctitle_fast(3)` restored an ancient BSD code path that did it just by overwriting process memory (like most (all?) other OSes), requiring the read side `ps(1)`, `top(1)` etc to do more work to copy it out.
- Reduces the number of system calls a busy database servers makes, for measurable performance boost (depending; I have seen ~10% improvement of pgbench TPS).

```
kernel
`- /sbin/init
   `- /usr/local/bin/postgres -D /data
      `- postgres: walsender replication 192.168.1.103(45243) streaming CC/2B717610
      `- postgres: tmunro salesdb [local] SELECT
      `- postgres: tmunro salesdb [local] UPDATE
      `- postgres: tmunro salesdb [local] COMMIT
      `- postgres: tmunro salesdb [local] idle
      `- postgres: checkpoint
      `- postgres: background writer
```

PostgreSQL 12: Change WAL file behaviour for ZFS

- PostgreSQL traditionally “reycles” 16MB write ahead log files, by renaming old ones into place. If it can’t to that, it zero-fills new files up front.
- Joyent reported that both of these things make no sense on ZFS, which overwrites anyway, so you might as well have a fresh inode rather than one you haven’t touched for ages that might not even be in memory. They proposed new settings `wal_init_zero` and `wal_recycle` to control that.
- This does indeed seem to improve performance, at least on high latency media (like my home lab spinning disk arrays).

FreeBSD 11.1: fdatasync(2) for UFS

- Like `fsync(2)`, but without the need to write “meta-data”. This means things like modified time can be lost after a crash, but we save an I/O. Thanks, kib@. This gives measurable `pgbench` TPS increases (10%+ on simple single threaded SSD test; `wal_sync_method=fsync|fdatasync`).
- Proposed for FreeBSD 13: `open(O_DSYNC)`; saving a system call for some patterns. D25090.
- Proposed for FreeBSD 13: `aio_fsync(O_DSYNC)`, the asynchronous cousin of `fdatasync(2)`. D25071.

IPC changes

- PostgreSQL 12 introduced option `shared_memory_type=sysv` option, because SysV memory + `kern.ipc.shm_use_phys=1` *might* still provide slightly very slightly higher performance on some benchmarks than anonymous shared memory (default since 9.3). (Thanks to kib@ for work done back in 2014 to close the gap; see references at end).
- FreeBSD 11 gained proper isolation of SysV shared memory between jails (even when not using `shared_memory_type=sysv`, we use a tiny SysV segment as interlocking prevent multiple servers clobbering each other). Thanks, jamie@.
- PostgreSQL 10 switched to POSIX unnamed semaphores instead of SysV semaphores, which can be cache line padded and don't require frobbing sysctls. (Other BSDs don't seem to support shared memory unnamed semaphores yet; we currently only do this for Linux and FreeBSD.)

FreeBSD 11: Unicode collations

- PostgreSQL makes heavy use of `strcoll_l(3)`.
- Previously, UTF-8 encoded text was not sorted correctly according to national norms. Thanks to bapt@ and others for merging illumos and DragonFlyBSD code for this into FreeBSD.
- This is pretty old news by now, but I wanted a chance to highlight it so I could get a chance to say that I think it would be really neat if other BSDs and macOS adopted this code!
- (I have some more wish list items in this area — see end of talk.)

I/O — Some ideas

POSIX_FADV_WILLNEED

Poor man's aio_read(2)

- Sometimes PostgreSQL calls `posix_fadvise(POSIX_FADV_WILLNEED)` to tell the kernel about future `pread()` calls. The setting `effective_io_concurrency` controls the maximum number of overlapping advice/read sequences generated by a single query. Hopefully this avoids stalls and gets concurrent I/O happening.
- Currently this is used to improve Bitmap Heap Scans and some maintenance tasks; more users of these features are in development, for example to avoid stalls in recovery (think: something like Joyent's pg_prefaulter, but built-in).
- Unfortunately this system call only works on Linux and NetBSD to my knowledge. Entirely absent: macOS, OpenBSD, Windows. No-op stub function: Solaris/illumos. Present, other hint supported but `POSIX_FADV_WILLNEED` ignored: FreeBSD, AIX. Unknown: HP-UX.

POSIX_FADV_WILLNEED for FreeBSD?

- The existing kernel code paths used to prefetch blocks when sequential access is detected can be easily hooked up to `posix_fadvise()`:
  - UFS: define `ffs_advise()`, call `breada()` for the range of blocks? Possibly a bit too naive, need to interact with `vfs_cluster.c` code to generate larger reads?
  - NFS: define `nfs_advise()`, refactor the existing prefetching code in `ncl_bioread()` into its own function `ncl_bioprefetch()`, and then call it from both places?
  - ZFS: define `zfs_advise()`, pass through to `dmu_prefetch()`. Need Linux and FreeBSD implementations via OpenZFS, devil in the details (memory mapped files, automated test). A start: https://github.com/openzfs/zfs/pull/9807

sync_file_range(2)? Bugzilla #203891

A Linux system call that is more flexible than fdatasync(2)

- Allows write back of a range of a buffered file, waiting optional. Used by Redis, MongoDB, Hadoop, PostgreSQL, …
- This allows user space to control the write back rate, distinguishing between temporary data files that don't need to be flushed to disk for data integrity purposes, and those that we know we're going to call `fsync()` on as part of a checkpoint.
- It sounds like `posix_fadvise(POSIX_FADV_DONTNEED)` should work for this purpose, but that also drops the data from kernel buffers which isn't necessarily a side effect we want. Perhaps we need a new thing… `UNPOSIX_FADV_WILLSYNC`?
- I don't know enough about ZFS to know if this makes any sense there; I think it probably makes sense for UFS and NFS.

Power loss atomicity

- PostgreSQL dumps complete images of 8KB* data page into the WAL, the first time each page is touched after each checkpoint (5 minute, 30 minutes, …). This avoids a problem with “torn pages” (another solution is the one used by MySQL: it double writes every data page with a sync in between, so only one can be torn in a power loss; different trade-offs).
- You can turn this off with `full_page_writes=off`, but that’s only safe if you know that the filesystem’s power loss atomicity is a multiple of PostgreSQL’s page size.
- It’d be really nice if you could ask the kernel!

I/O — Future direction

Real asynchronous and direct I/O

Early investigative work to modernise our I/O layer

- Current thinking is that we should use io_uring on Linux, POSIX AIO on BSDs/HPUX/AIX (all systems that uses async I/O down to the driver or kernel threads, but not systems that use user threads like Linux and Solaris, due to PostgreSQL process-based architecture), and Windows native.
- One open question is whether there is any advantage to using kqueue for completion notification; it only seems to be supported on FreeBSD, so we'll need to support signal based notification anyway.
- Early prototyping on Linux is very promising for performance; finally achieving device speed, skipping layers of buffering and system calls. POSIX version not yet started.

Comparative observations

UFS read-ahead heuristics vs Parallel Sequential Scan

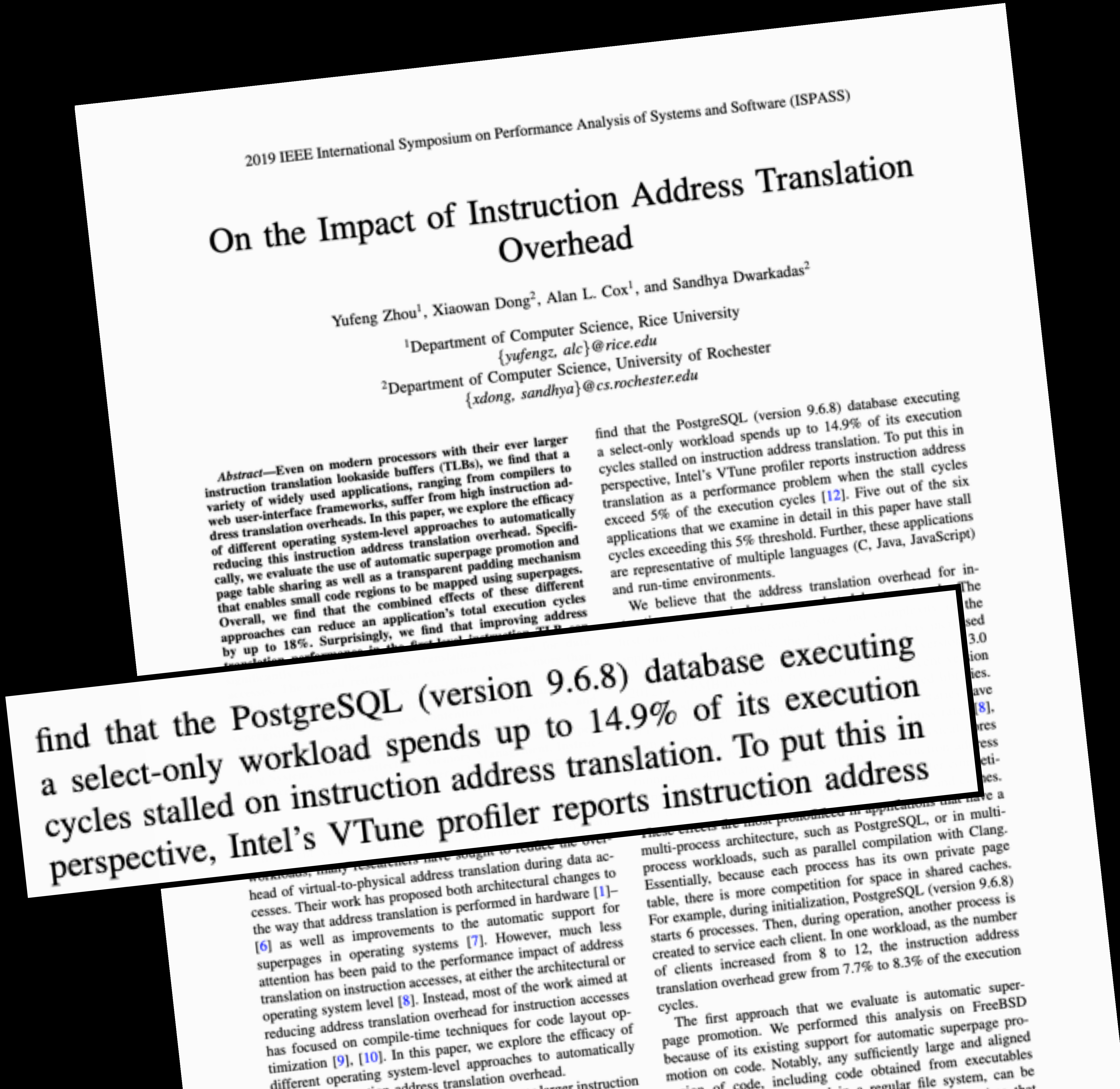
- Parallel Sequential Scan's goal is to divide tuple processing work up by handing out sequential blocks to parallel worker processes.
- UFS doesn't recognise this access pattern as sequential. Linux does, due to read-ahead "window". ZFS apparently does too.
- Maybe PostgreSQL is the only software to have come up with this diabolical access pattern, due to its process model and lack of AIO and direct I/O (for now).
- Related: mixing two streams, read and write, in the same file. D25024 to fix that.

```
create table t as
select generate_series(1, 2000000000)::int i;

set max_parallel_workers_per_gather = 0;
select count(*) from t;
-> 14.5s, ~4000 IOPS, ~128kB/t = ~500MB/s

23080: pread(6,<data>,8192,0x10160000)
23080: pread(6,<data>,8192,0x10162000)
23080: pread(6,<data>,8192,0x10164000)
23080: pread(6,<data>,8192,0x10166000)

set max_parallel_workers_per_gather = 1;
select count(*) from t;
-> 35.6s, ~6000 IOPS, ~33kB/t = ~180MB/s

23080: pread(6,<data>,8192,0x10160000)
23081: pread(9,<data>,8192,0x10162000)
23080: pread(6,<data>,8192,0x10164000)
23081: pread(9,<data>,8192,0x10166000)
```

FreeBSD is good at putting stuff in super pages

```
$ sudo procstat -v 91751 | grep -E ' (FLAG|..S..) '
  PID              START                END PRT  RES PRES REF SHD FLAG  TP PATH
91751           0x4b0000           0x8e5000 r-x 1037 1690  24   1 CNS-- vn /usr/local/bin/postgres
91751        0x801ecb000        0x886703000 rw- 23810 23810  10   0 --S-- df
```

- Databases touch a lot of data and code randomly, and benefit enormously from super (huge, large) pages.
- Getting these things to work on many popular operating systems requires extra configuration and hoop jumping (Linux: libhugetlbfs, remounting /dev/shm). This causes FreeBSD to do 5-20% better at certain kinds of large random memory access tasks without tuning. Example: Parallel Hash Join, in shm_open() memory.
- Interesting new research on text segment that happens to use PostgreSQL (among other applications) on FreeBSD.



2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)

On the Impact of Instruction Address Translation Overhead

Yufeng Zhou[1], Xiaowan Dong[2], Alan L. Cox[1], and Sandhya Dwarkadas[2]
[1]Department of Computer Science, Rice University
{yufengz, alc}@rice.edu
[2]Department of Computer Science, University of Rochester
{xdong, sandhya}@cs.rochester.edu

***Abstract*—Even on modern processors with their ever larger instruction translation lookaside buffers (TLBs), we find that a variety of widely used applications, ranging from compilers to web user-interface frameworks, suffer from high instruction address translation overheads. In this paper, we explore the efficacy of different operating system-level approaches to automatically reducing this instruction address translation overhead. Specifically, we evaluate the use of automatic superpage promotion and page table sharing as well as a transparent padding mechanism that enables small code regions to be mapped using superpages. Overall, we find that the combined effects of these different approaches can reduce an application's total execution cycles by up to 18%. Surprisingly, we find that improving address**

find that the PostgreSQL (version 9.6.8) database executing a select-only workload spends up to 14.9% of its execution cycles stalled on instruction address translation. To put this in perspective, Intel's VTune profiler reports instruction address translation as a performance problem when the stall cycles exceed 5% of the execution cycles [12]. Five out of the six applications that we examine in detail in this paper have stall cycles exceeding this 5% threshold. Further, these applications are representative of multiple languages (C, Java, JavaScript) and run-time environments.

We believe that the address translation overhead for in-

head of virtual-to-physical address translation during data accesses. Their work has proposed both architectural changes to the way that address translation is performed in hardware [1]–[6] as well as improvements to the automatic support for superpages in operating systems [7]. However, much less attention has been paid to the performance impact of address translation on instruction accesses, at either the architectural or operating system level [8]. Instead, most of the work aimed at reducing address translation overhead for instruction accesses has focused on compile-time techniques for code layout optimization [9], [10]. In this paper, we explore the efficacy of different operating system-level approaches to automatically address translation overhead.

multi-process architecture, such as PostgreSQL, or in multi-process workloads, such as parallel compilation with Clang. Essentially, because each process has its own private page table, there is more competition for space in shared caches. For example, during initialization, PostgreSQL (version 9.6.8) starts 6 processes. Then, during operation, another process is created to service each client. In one workload, as the number of clients increased from 8 to 12, the instruction address translation overhead grew from 7.7% to 8.3% of the execution cycles.

The first approach that we evaluate is automatic superpage promotion. We performed this analysis on FreeBSD because of its existing support for automatic superpage promotion on code. Notably, any sufficiently large and aligned code, including code obtained from executables

find that the PostgreSQL (version 9.6.8) database executing a select-only workload spends up to 14.9% of its execution cycles stalled on instruction address translation. To put this in perspective, Intel's VTune profiler reports instruction address

FreeBSD doesn’t drop cached data on write back failure

- Historically, PostgreSQL (and MySQL, MongoDB, …) believed it was meaningful to be able to “try again” if `fsync(2)` fails during a checkpoint. Now we panic immediately. On some other systems, but not FreeBSD, if `fsync(2)` fails then dirty data is evicted. That allows future calls to `fsync(2)` to succeed despite doing nothing. Example of transient failure: `ENOSPC` reported at `fsync(2)` time, or `EIO` reported on some kind of virtualised storage that recovers.
- Historically, PostgreSQL believed that it was safe to write data, close the descriptor, reopen it later and then call `fsync(2)`. This seems to be true on some systems including FreeBSD, but not true on systems that (1) throw away dirty pages due to error during asynchronous write back and (2) might evict the only record of an error before it’s reported to user space.
- You see why we’re interested in moving to direct I/O, and owning buffering completely…

Wish list

- Text comparison
    - I would like libc to be able to report collation versions, so we could avoid index corruption when the sorting rules change (Win32 does this, IBM ICU does this, POSIX should do it too!); D17166.
    - I wish strcoll_l() didn't internally expand every string to wide character format, extra malloc() + free().
    - I wish there were a way to do strcoll_l() with non-NUL-terminated strings, so you could avoid the need to copy such strings just to terminate them; strncoll_l()?
    - If we knew that strxfrm_l() were bullet-proof we could speed up sorting considerably.
- Port
    - I wish we could install postgresql11 and postgres12 at the same time (different install paths).
    - I wish libpq5 (the client library) were separate from postgresql12-client.
    - I wish we had a port of Debian's postgresql-common, so you could easily start, stop, copy, create, destroy PostgreSQL instances; it could probably be extended to support some ZFS magic like fast cloning too.

Thanks for listening!

Some links and references

- On the Impact of Address Translation Overhead (Zhou, Dong, alc@, Dwarkadas)
  https://www.cs.rochester.edu/u/xdong/ispass-19-final.pdf

- kib@'s 2014 report on PostgreSQL/FreeBSD performance:
  http://kib.kiev.ua/kib/pgsql_perf.pdf

- Attempts to understand what different kernels do with write-back errors:
  https://wiki.postgresql.org/wiki/Fsync_Errors

- A kind of cross-project to-do list:
  https://wiki.postgresql.org/wiki/FreeBSD