PostgreSQL/FreeBSD performance and scalability on a 40-core machine* v1.0

Konstantin Belousov <kib@FreeBSD.org>

July 13, 2014

SVN revision 138

Contents

1 Overview **2**

1.1 Measuring contention . . . 3

2 Page Fault Handler **3**

2.1 Fast Path . . . 5

2.2 Phys backing for shared area . . . 6

2.3 Pre-populating the shared area . . . 6

2.4 DragonFlyBSD shared page tables . . . 6

3 Buffer Cache **7**

3.1 nblock . . . 7

3.2 bqclean . . . 7

4 Page Queues **8**

5 Semaphores **8**

6 sched_ule cpu_search() **10**

7 Miscellaneous **10**

7.1 hwpmc(4) on Westmere-EX . . . 10

7.2 mfi(4) unmapped I/O . . . 10

7.3 tmpfs(5) shared locking . . . 10

7.4 Process titles . . . 10

8 Results **11**

9 Appendix: Patch use **13**

*This work is sponsored by The FreeBSD Foundation.

1 Overview

This report details my investigation into reports of poor scalability of the PostgreSQL database on FreeBSD, and describes system changes and tuning to significantly improve performance and scalability.

Testing was done using two machines, `pig1` and `zoo` in the FreeBSD *Test-ClusterOne* laboratory hosted at Sentex Communications.

Machine	Usage
`pig1`	Device under test
`zoo`	Test cluster infrastructure: build and netboot host

Table 1: Machines used.

The `pig1` configuration:

- Four Westmere-EX E7-4850 CPUs at 2.0GHz, 10 cores with HTT each, for a total of 40 cores, and 80 hardware threads;
- 1TB of RAM;
- LSI ThunderBolt MegaRAID disk controller;
- 3 Samsung SSD 840 in single-volume RAID-1 configuration.

The machine is netbooted from `zoo`. The software configuration on `pig1`:

Software	Revisions
FreeBSD	HEAD at r267575, amd64, stripped down and modularized kernel config
PostgreSQL	9.3.4 + patches, built from ports
pgbench	9.3.4 from port `databases/postgresql-contrib`

Table 2: Software versions.

The investigation started with rumors of abysmal PostgreSQL performance on many-core machines. Evidence was provided in [1] that the issue is an enormous performance drop in the pgbench benchmark when the parallelism increases. Attempts to use even half of the cores on the machine made PostgreSQL perform even worse than the single-threaded test.

The nature of the problem makes lock contention an immediate suspect. Other causes are also possible, like excessive cache sharing, or algorithmic issues. Still, a check for locking problems is due. It was decided to measure contention using the `LOCK_PROFILING(9)` facility of the FreeBSD kernel. `LOCK_PROFILING(9)` provides essential data for analyzing lock contention, but requires a custom kernel configuration and introduces higher system overheads.

1.1 Measuring contention

For measurement, a production kernel config was created. In particular, all debugging options which slow down the kernel were removed.

```
option LOCK_PROFILING
```

was added to the kernel config, and the machine was booted with the new kernel.

Profiling was done with the following script, which assumes that the caches are already hot:

```
#!/bin/sh
sudo sysctl debug.lock.prof.reset=1
sudo sysctl debug.lock.prof.enable=1
pgbench -j 40 -c 40 -T 180 -S -h /tmp -U bench bench
sudo sysctl debug.lock.prof.enable=0
sudo sysctl debug.lock.prof.stats >prof.1.txt
(head -2 prof.1.txt; sort -k 4 -nr <prof.1.txt) >prof_sorted.1.txt
```

After execution, `prof_sorted.1.txt` contains the profiling results sorted by lock wait time. See table 3 for the first six lines of the report on the unpatched kernel.

2 Page Fault Handler

After initial lock profile data was obtained from the sample run for 40 clients, the immediate outstanding lock congestion appears to be the `VM object(9)` lock in the page fault handler `vm_fault_hold(9)` function. The first three lines in table 3 provide irrefutable evidence for this claim. There are three lines, but the affected code path is the same. The profile data is split due to the fault handler dropping and re-acquiring the object lock during execution. Cumulative wait time for these three places is larger by an order of magnitude than the next appearance in the profile output.

Stating only the details of the FreeBSD algorithm for page fault handling which are relevant to the current discussion, fault processing is:

- the low-level machine-dependent (MD) handler calls `vm_fault(9)`, which is machine-independent (MI) VM code.
- `vm_fault_hold(9)` locks the `vmspace` for the current process and looks up `vm_map_entry` which describes the faulting address.
- `vm_object`, which backs the found `vm_map_entry`, is locked exclusively.

The tested load on the PostgreSQL server side is comprised of server processes accessing the shared data area, which was created as a large shared anonymous memory region. The region is implemented by our VM in the process-private address spaces as process-unique VM map entries which all point to the same backing

max	wait_max	total	wait_total	count	avg	wait_avg	cnt_hold	cnt_lock	name
375	465018	120484291	2595677997	4328075	27	599	0	4187237	vm/vm_fault.c:1056 (rw:vm object)
9868	495017	33096852	1982263541	2226088	14	890	0	2113302	vm/vm_fault.c:303 (rw:vm object)
9304	465838	24335805	1864397319	2147294	11	868	0	1984397	vm/vm_fault.c:909 (rw:vm object)
61	1743	70250978	601931512	10998523	6	54	0	9968930	kern/subr_turnstile.c:551 (spin mutex:turnstile)
40	440034	886032	70222211	78794	11	891	0	74764	vm/vm_page.c:900 (rw:vm object)
939787	433240	47110362	49641604	2245668	20	22	0	65106	vm/vm_object.c:513 (rw:vm object)

Table 3: debug.lock.prof.stats sample output

vm_object. Both the object's pages and page table entries are populated on demand. Prefaulting is performed by the VM in a limited form, but is not significant since the connection processes are started from a cold shared region state.

Threads which access a non-mapped page in the shared region congest on the backing object lock, as outlined above. For the 40-client test, the wait time on the object lock was an order of magnitude higher than on the next most congested lock. Explained below are two changes which completely eliminated the vm_object lock from the top lock profiles.

2.1 Fast Path

The final goal of the vm_fault(9) function is to install the proper page table entry (or fill TLB) for the faulted address by calling pmap_enter(9). To do the job, the function must:

- ensure that the address is valid.
- find the corresponding page from the hierarchy of backing objects.
- if the page does not exist, allocate it.
- populate the page with valid content, either from the pager or from the lower object for COW.

There is a lot of work to do, and it happens mostly under the top vm_object's lock taken in exclusive mode. This explains the lock profiling results.

Now, assume that there is

- no backing object chain (our mapping is shared and anonymous);
- the object is of swap (or phys, see 2.2) type;
- the page to map already exists in the object.

In this situation, which is exactly the case for page faults on the shared region, most of the work performed by vm_fault_hold(9) is reduced to checking the conditions and deciding that no action is required. Still, a lot of code must execute just to check that we are ready to call pmap_enter(9). Moreover, the vm_object is not modified during the fault, because the page already exists and does not need to be allocated. This means that exclusive object locking is excessive.

I added code to vm_fault_hold(9) which takes the vm_object lock in shared mode immediately after the object is resolved, and checks whether the conditions above hold. If true, we can call pmap_enter(9) right away instead of going through the long path for the common case.

One complicated issue with the fast path is interaction between the shared mode object lock and page busying. Right now, there is no code in FreeBSD which would busy a page while the owner object is shared locked, as done in the patch. Busying is needed to call pmap_enter(9). It is not clear at present whether this mode of operation should be put into mainline, since shared object lock owners must be ready for the page busy state changing underneath them.

2.2 Phys backing for shared area

Typical mappings in the process address space are *managed*. For a physical page participating in the mapping, the VM may ask `pmap(9)` to revoke all access (see `pmap_remove_all(9)`) or downgrade all access to read-only, which is done by `pmap_remove_write(9)`. This is needed, for example, for paging out or for setting up the page for write.

It is up to `pmap(9)` to implement manageability, but it is not too wrong to assume that it must be able to enumerate all page mappings to perform the operations. The typical FreeBSD `pmap(9)` implements it by creating `pv entries` for each page mapping, which describes address space and address of the mapping, and linking them together in the MD part of `vm_page`. `pmap_enter(9)` needs to allocate `pv entry`. If the page participates in shared mappings in a large number of address spaces, the page's `pv_list` becomes large.

For many years, the FreeBSD implementation of *System V Shared Memory* has provided the tunable `sysctl(2) kern.ipc.shm_use_phys`, which allows using a `PHYS`-type VM object to back shared memory segments. The end result is that the shared memory is not pageable, and pv entries are not needed to track the mappings.

I implemented a similar tunable `vm.shared_anon_use_phys`, which forces the `PHYS`-type for the objects backing the anonymous shared memory. This way, the PostgreSQL shared area avoids the overhead of the pv entries maintenance.

A more fine-grained approach is probably needed there, allowing the application to request `PHYS` mappings instead of enforcing it on all shared users. A new `mmap(2)` flag, perhaps `MMAP_NON_SWAP`, could be introduced.

2.3 Pre-populating the shared area

Even after the removal of pv entries for the shared area, and the fast page fault handler path, the first access to the shared area page still causes exclusive `vm_object` locking to allocate the physical page. Exclusive locking blocks other threads' page fault handlers, which could have proceeded by the fast path in the shared lock mode.

A new tunable, `sysctl(2) vm.shm_anon_phys_preload`, is added. It requests that the object backing the shared anonymous area is populated with the pages during object creation.

With the fast fault handler path, physical backing for the shared area and prepopulation of it, all page faults are handled by the fast path. The object lock is shared-locked, eliminating contention in `vm_fault_hold(9)`.

Future work should consider applying `RLIMIT_MEMLOCK` when populating the object.

2.4 DragonFlyBSD shared page tables

The DragonFlyBSD approach to fixing the performance issue of contention on parallel page faults to the shared area was to allow sharing page tables. The change would require major rework of the `pmap(9)` layer for each machine architecture,

together with complicated interaction with the fine-grained `pmap(9)` locking. It is also not clear how parallel initial population of the page tables for the shared region could be achieved. This matters for real-world PostgreSQL performance, as opposed to a purely isolated test setup.

Due to the doubts outlined above, I decided to implement the fast path for the page fault handler and physical backing of the shared regions.

3 Buffer Cache

After the page fault handler contention was sorted out, albeit in a hackish way, the next outstanding locks in the profile appeared to be related to the buffer cache, at least when the PostgreSQL data files were located on a `UFS(5)` filesystem.

On FreeBSD, the file data is cached in pages kept in the `vm_object` and connected to the corresponding `vnode`. Nonetheless, to perform the I/O, UFS instantiates the `buf(9)` buffers. These buffers mimic the old UNIX buffer cache, but reference the physical memory pages containing the data. The profile indicated that congestion points are the buffer reclamation, done by `bqrelse(9)` and `brelse(9)` functions, and insertion of the clean buffers into the global queues from where they can be reused.

3.1 nblock

The `nblock` mutex serves two purposes:

1. it protects the `needsbuffer` variable which keeps flags indicating the buffer request types which cannot be satisfied now;

2. it protects against missed wakeups from the `brelse(9)` when `getnewbuf(9)` blocks due to reasons specified in `needsbuffer`.

Despite `getnewbuf(9)` rarely blocking in the testing load, the need to protect `needsbuffer` in `bufspacewakeup()` and `bufcountadd()` in the `brelse()` execution path created very significant congestion on `nblock`.

I moved the `needsbuffer` updates to `atomic(9)` operations. The wakeup protection is ensured by changing `nblock` to rw lock. The update code takes `nblock` in read mode, and `getnewbuf_bufd_help()` checks `needsbuffer` and goes to sleep under write-locked `nblock`. Contention on `nblock` for `brelse()` was eliminated.

The changes are committed as r267255. A followup fix for the gcc build was committed as r267264.

3.2 bqclean

The buffer cache keeps unused (unlocked) clean buffers on several lists. These lists group buffers with similar characteristics with regard to the attached resources. The buffer may be returned to use, or it may be reclaimed for reuse if another buffer

must be created. The lists are scanned for a suitable buffer for reuse. Note that the `struct buf`s are type-stable.

The clean lists themselves, and the scan procedure to select the buffer for reuse, are protected by the `bqclean` mutex. After the other changes, it appeared high in the lock profile. The lock blocks both `brelse()`, when the buffer is put on a clean queue, and scan.

I redefined `bqclean` to only protect the queue modifications. Scan code now walks the queues without taking the `bqclean` lock. The queue structure could be changed under the iterator. That would be tolerable since buffers are type-stable, and we always get the next pointer pointing to the live buffer, although possibly no longer from the iterated queue. The next buffer is locked and its location on the queue is rechecked afterward. If we are dropped from the queue, the scan is restarted.

A queue generation counter prevents infinite waits which could occur if buffers are placed on the queue before the iterator.

4 Page Queues

The inactive queue lock appeared to be highly congested in the test load due to the buffer release.

When the buffer is formed, it wires the pages which constitute it. On buffer reclamation, the pages are unwired. Wired pages belong to the wire owner. Unwired pages are put on the active or inactive queues, where they can be found by the `pagedaemon(9)`. In particular, the corresponding queue must be locked when the page is wired or unwired. When unwiring, buffer cache always requests deactivation of the page, which causes the inactive queue lock to be held for page insertion.

During the multithreaded pagedaemon work, the page queues were split into queues of pages owned by a corresponding NUMA proximity domain. For the 4-socket `pig1`, this meant that the inactive queue was in fact split into four queues. There is no intrinsic relation between NUMA domain and VM page domain, and nothing prevents splitting the NUMA domains into more fine-grained sets of pages with a corresponding split of the queues and page queue locks.

The tunable `vm.ndomain_split_factor` is introduced, which additionally splits each NUMA domain into the specified number of sub-domains. This reduces the contention on the (inactive) page queue locks.

5 Semaphores

Traditionally, PostgreSQL uses *System V semaphores* for inter-process synchronization. The FreeBSD implementation of semaphores uses a mutex pool lock for protection of kernel data describing the semaphore set. This mutex is highly congested when many parallel threads operate on the single set.

The PostgreSQL code allows using either SysV semaphores or *Posix semaphores* (see `sem_open(3)`). The userspace Posix semaphore implementation defers to

usem(2) for congested operations. After switching the PostgreSQL server to Posix semaphores, the kernel lock profile was cleaned from the inter-processing synchronization locks. The change was done as a (trivial) ports(5) patch for the databases/postgresql93-server port.

6 sched_ule cpu_search()

Rough profiling using the top mode from pmcstat(8) for the

instructions-retired

event demonstrated that up to 25% of the samples gathered are for the cpu_search() function of the sched_ule(9) scheduler.

Apparently, the clang(1) compiler neither inlines the function nor uses constant propagation here, resulting in a convoluted code path with many conditional branches executed for each context switch. An experiment with compiling kern/sched_ule.c with gcc(1) version 4.9.0 reduced the sample rate to 8–9%.

7 Miscellaneous

7.1 hwpmc(4) on Westmere-EX

Trying to use hwpmc(4) on the test machine, it appeared non-functional, causing general protection faults when trying to access non-existent uncore PMC MSRs. This is due to the uncore PMC subsystem being significantly different from that used in desktop-class CPUs and Xeon 7500 (Nehalem-EX) and E7 (Westmere-EX) variants. To get a working hwpmc(4), although without uncore monitoring, the Core i7 and Westmere support in the driver was restructured to avoid touching uncore for EX microarchitectures, see r267062.

7.2 mfi(4) unmapped I/O

The driver for mfi(4) controllers on pig1 did not support unmapped I/O. On an 80-thread machine, IPI overhead from TLB shootdowns is enormous. As the first step, before doing any measurements, I added support for unmapped I/O to mfi(4), committed as r266848.

7.3 tmpfs(5) shared locking

Running the load with the PostgreSQL data files on tmpfs(5) (to exclude the UFS(5) and buffer cache overhead), I noted that tmpfs(5) enforces exclusive mode for its vnode lock acquisitions. tmpfs(5) should allow shared vnode locks for non-modifying vnode accesses, but this was overlooked during the porting effort. The glitch was corrected in r267060.

7.4 Process titles

FreeBSD provides a facility for the running process to change the displayed command in ps(1) output, see setproctitle(3). The operation is performed using sysctl(2) KERN_PROC_ARGS.

By default, PostgreSQL updates the title of the connection handler whenever a new SQL command is received. This can cause high-frequency calling

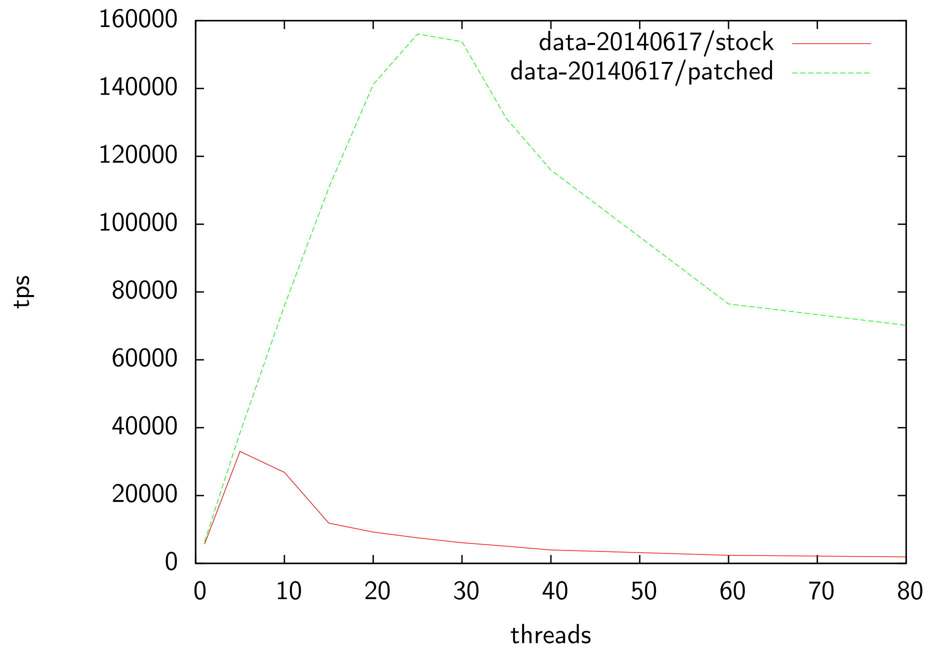
of `setproctitle(3)`, which is CPU-intensive and causes contention on the `sysctl(9)` internal locks. It is important to disable title update in the PostgreSQL configuration to avoid unnecessary CPU load.

8 Results

Measurements were obtained using methodology similar to that described in [2]. The results are presented in the figure below. The 'stock' graph corresponds to the unpatched kernel from head at r267575, while the 'patched' graph is for a patched kernel with the same configuration.

num. of threads	stock kernel tps	patched kernel tps
1	5698.41	6378.57
5	33026.72	38513.55
10	26854.85	75916.02
15	11874.05	110696.75
20	9232.19	141171.54
25	7530.54	156066.39
30	6086.78	153771.79
35	5050.99	131141.87
40	3969.86	115945.73
60	2389.97	76474.32
80	1944.68	70179.30

Table 4: Transactions per second vs. number of threads.


160000
140000
120000
100000
80000
60000
40000
20000
0
tps
0 10 20 30 40 50 60 70 80
threads
data-20140617/stock
data-20140617/patched

9 Appendix: Patch use

1. Apply the kernel patch.

2. Apply the patch to the PostgreSQL server by placing the `patch-2` file into `databases/postgresql93-server/files` and rebuilding the port.

3. Ensure that the multithreaded pagedaemon is enabled with the

   ```
   options MAXMEMDOM=N
   ```

   kernel config setting. Set N to the number of sockets multiplied by the domain split factor. The latter is set with the `vm.ndomain_split_factor` loader tunable. Set the split factor so that N is at least greater than the number of cores.[1]

4. Set the `sysctl(8)`s

   - `vm.shared_anon_use_phys=1`
   - `vm.shm_anon_phys_preload=1`

   before starting the PostgreSQLserver.

5. Ensure that `postgresql.conf` disables updates to the server process titles by using the

   ```
   update_process_title = off
   ```

   directive.

[1] For `pig1`1, I used MAXMEMDOM=128 and vm.ndomain_split_factor=8, which gives $4(\text{sockets}) * 8(\text{splitfactor}) = 32$ domains. There is no overhead from setting MAXMEMDOM higher than the actual number of domains. However, a current artificial limitation is that the number of physical segments must be less than or equal to the number of bits in the long type, 64 for amd64, and each domain consumes a physical segment.

References

[1] http://lists.dragonflybsd.org/pipermail/users/attachments/20140310/4250b961/attachment-0002.pdf

[2] http://lists.dragonflybsd.org/pipermail/users/attachments/20121010/7996ff88/attachment-0002.pdf

The author thanks The FreeBSD Foundation for allowing him to work on the issues, and to Ed Maste, Warren Block and Dimitry Andric for providing suggestions and editorial help with the text.