Linux kernel scaling

Andi Kleen and Tim Chen

Intel Corporation

Sep 2011

Case study:
MOSBENCH Exim Mail Server Workload

- Configure exim to use tmpfs for all mutable files - spool files, log files, and user mail files. No file system / IO test.
- Clients run on the same machine as exim. Each repeatedly opens an SMTP connection to the mail server.
- Sends 10 separate 20-byte messages to a local user.
- Running on a 4S 40Core/80Thread system.

Exim initial profile on 2.6.38

Baseline

```
-     52.82%          exim  [kernel.kallsyms]             [k] do_raw_spin_lock
   - do_raw_spin_lock
      - 99.87% _raw_spin_lock
         + 39.61% dput
         + 38.61% dget
         + 18.68% nameidata_drop_rcu
         + 0.65% nameidata_drop_rcu_last
         + 0.63% __do_fault
+     11.14%          exim  [kernel.kallsyms]             [k] vfsmount_lock_local_lock
+      4.10%          exim  [kernel.kallsyms]             [k] vfsmount_lock_global_lock_
```

File System – Fix absolute path names #1

- Slow path of directory entry (dentry) lookup requires updating the reference count of all the dentries in the directory path. Cache line bouncing on reference counts.
- 2.6.38 introduces RCU path walk. Per dentry seqlock detects dentry modificatoins
- Absolute paths always dropped out of RCU because of incorrect seqlock initialization of root.
- Fix merged 2.6.39.

Exim Profile (after fix #1)

Throughput at 197% relative to baseline

```
-    29.47%        exim  [kernel.kallsyms]        [k] do_raw_spin_lock
   - do_raw_spin_lock
      - 99.49% _raw_spin_lock
         + 40.42% dput
         + 20.38% dget
         + 17.91% nameidata_drop_rcu
         + 11.63% __do_fault
         + 3.07% __d_lookup
         + 2.51% anon_vma_lock.clone.11
         + 0.75% nameidata_drop_rcu_last
         + 0.54% unlink_file_vma
-     9.98%        exim  [kernel.kallsyms]        [k] vfsmount_lock_local_lock
+     5.80%        exim  [kernel.kallsyms]        [k] filemap_fault
+     2.54%        exim  [kernel.kallsyms]        [k] vfsmount_lock_global_lock_online
+     2.26%        exim  [kernel.kallsyms]        [k] page_fault
```

File System – Path Walk Speedup #2

- RCU path walk still keeps failing after we've fixed the initialization of seq_number in the seqlock.
- LSM layer (inode_exec_permission) unconditionally drops out of RCU path walk
- Fix the security code to support RCU path walk.
- Fix merged 2.6.39.

File System - Mount Lock (#3)

- mntput_no_expire and lookup_mnt separate short term and long term counts. Short term is per cpu, long term is global.
- vfsmount “put” had to sum up all short term counters, unless there is a long term mount that pins the entry.
- pipe_fs and other internal file systems always triggered the short term mount case because they weren't mounted, but still used
- Fix merged 3.0.

Profile after Fix #2 & #3

Throughput at 256% relative to baseline

```
-    17.00%      exim  [kernel.kallsyms]       [k] filemap_fault
   - filemap_fault
      + 99.88% __do_fault
-    12.45%      exim  [kernel.kallsyms]       [k] do_raw_spin_lock
   - do_raw_spin_lock
      - 98.34% _raw_spin_lock
         + 78.45% __do_fault
         + 8.83% anon_vma_lock.clone.11
         + 1.90% unlink_file_vma
         + 1.27% dup_mm
         ...
+     3.14%      exim  [kernel.kallsyms]       [k] page_fault
+     2.49%      exim  [kernel.kallsyms]       [k] clear_page_c
+     2.24%      exim  [kernel.kallsyms]       [k] unmap_vmas
```

File Readahead - Cache Line Bouncing (#4)

- File map page faults of memory mapped files are taking a lot of time.
- The read ahead parameters ra->mmap_miss and ra->prev_pos caused a lot of cache line bouncing when they were updated frequently.
- In our tests, many of our test files are stored in tmpfs within the memory for speed, which makes readahead of these files unnecessary.
- Turn off readahead and update of readahead parameters for tmpfs.
- This could still be an issue for file system which are fast, but still need readahead.
- Fix merged 2.6.39.

Profile after Fix #4

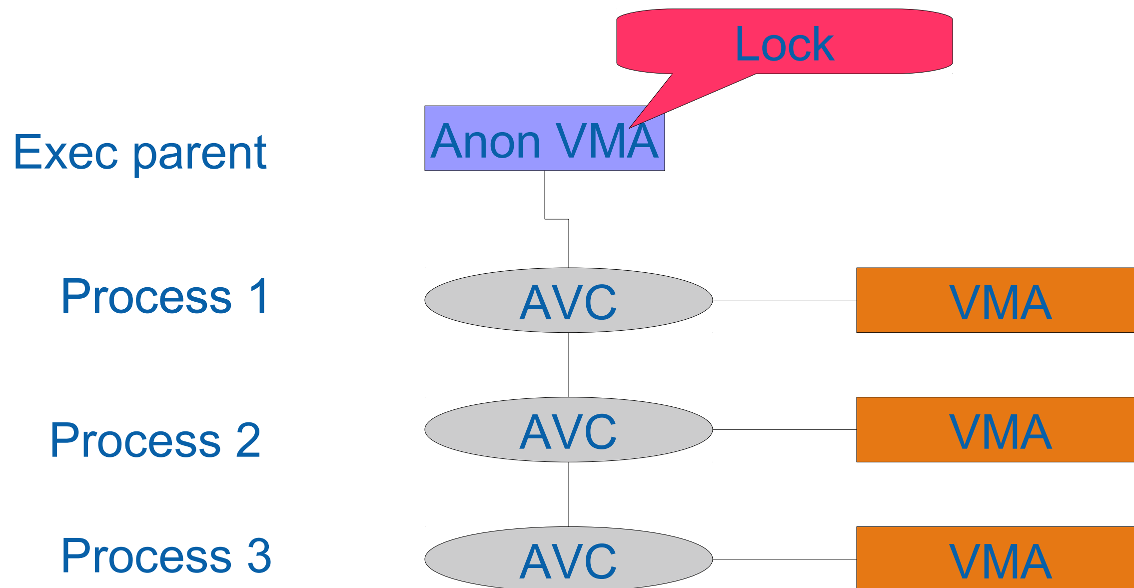
Throughput at 290% relative to baseline

```
-    24.41%      exim  [kernel.kallsyms]      [k] do_raw_spin_lock
   - do_raw_spin_lock
      - 99.22% _raw_spin_lock
         + 77.96% anon_vma_lock.clone.11
         + 14.85% vma_adjust
         + 1.06% unlink_file_vma
         + 0.57% __pte_alloc
         + 0.54% dup_mm
+     3.45%      exim  [kernel.kallsyms]      [k] page_fault
+     2.64%      exim  [kernel.kallsyms]      [k] clear_page_c
+     2.24%      exim  [kernel.kallsyms]      [k] unmap_vmas
+     1.67%      exim  [kernel.kallsyms]      [k] page_cache_get_speculative
```

Memory - Reduce Anon VMA Lock Contention #5

- Exim forks many child processes to handle incoming email.
- The initial virtual memory areas for child processes are cloned from parents and shares lock with parent process's vma.
- Aggressive merging of child processes' new vmas with the cloned vmas will introduce contention on the parent process anon_vma lock (even though vmas are local).
- Avoiding the merging of child processes' vmas with the cloned vmas greatly reduces the contentions.
- When we insert a new memory area to vma and change only vma->end, anon_vma locking is unnecessary. Remove that.
- Fixes merged 2.6.39.

Anon vma chains



Profile after Fix #5

Throughput at 381% relative to baseline

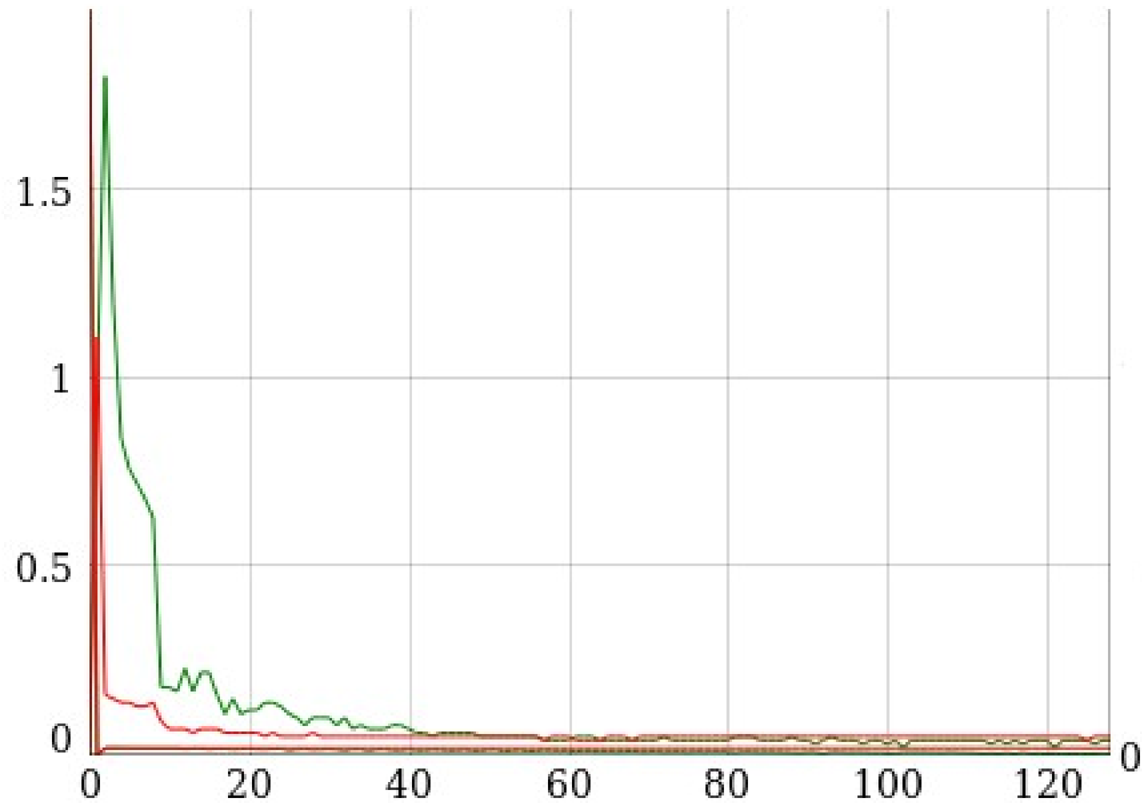
```
-    4.80%      exim  [kernel.kallsyms]      [k] do_raw_spin_lock
   - do_raw_spin_lock
      - 94.94% _raw_spin_lock
         + 51.42% anon_vma_lock.clone.11
         + 6.08% unlink_file_vma
             ...

+    4.48%      exim  [kernel.kallsyms]      [k] page_fault
+    3.59%      exim  [kernel.kallsyms]      [k] clear_page_c
+    2.84%      exim  [kernel.kallsyms]      [k] unmap_vmas
```

Memory - Reduce Anon VMA Lock Contention #6

- With frequent forks/exits, there are a lot of chaining and de-chaining of child processes' anon_vmas, needing frequent acquisition of root anon_vma lock.
- By doing batch chaining of the anon_vmas, we can do more work per acquisition of the anon_vma lock, and reduce contention.
- Regression originally from 2.6.35 caused by a correctness change: always lock the chain head.
- Batch chaining adopted in v3.0.
- Still slower than 2.6.35.

Problem Visible in Micro-benchmark



Profile after Fix #6

Throughput at 397% relative to baseline

```
+   4.61%   exim  [kernel.kallsyms]   [k] page_fault
+   3.64%   exim  [kernel.kallsyms]   [k] clear_page_c
+   3.17%   exim  [kernel.kallsyms]   [k] do_raw_spin_lock
+   2.92%   exim  [kernel.kallsyms]   [k] unmap_vmas
+   2.22%   exim  [kernel.kallsyms]   [k] page_cache_get_speculative
+   1.85%   exim  [kernel.kallsyms]   [k] copy_page_c
+   1.47%   exim  [kernel.kallsyms]   [k] __list_del_entry
-   1.47%   exim  [kernel.kallsyms]   [k] format_decode
   - format_decode
      - 94.57% vsnprintf
         - 98.51% seq_printf
              show_cpuinfo
              seq_read
              proc_reg_read
              vfs_read
```

libc – Inefficient Functions (#7)

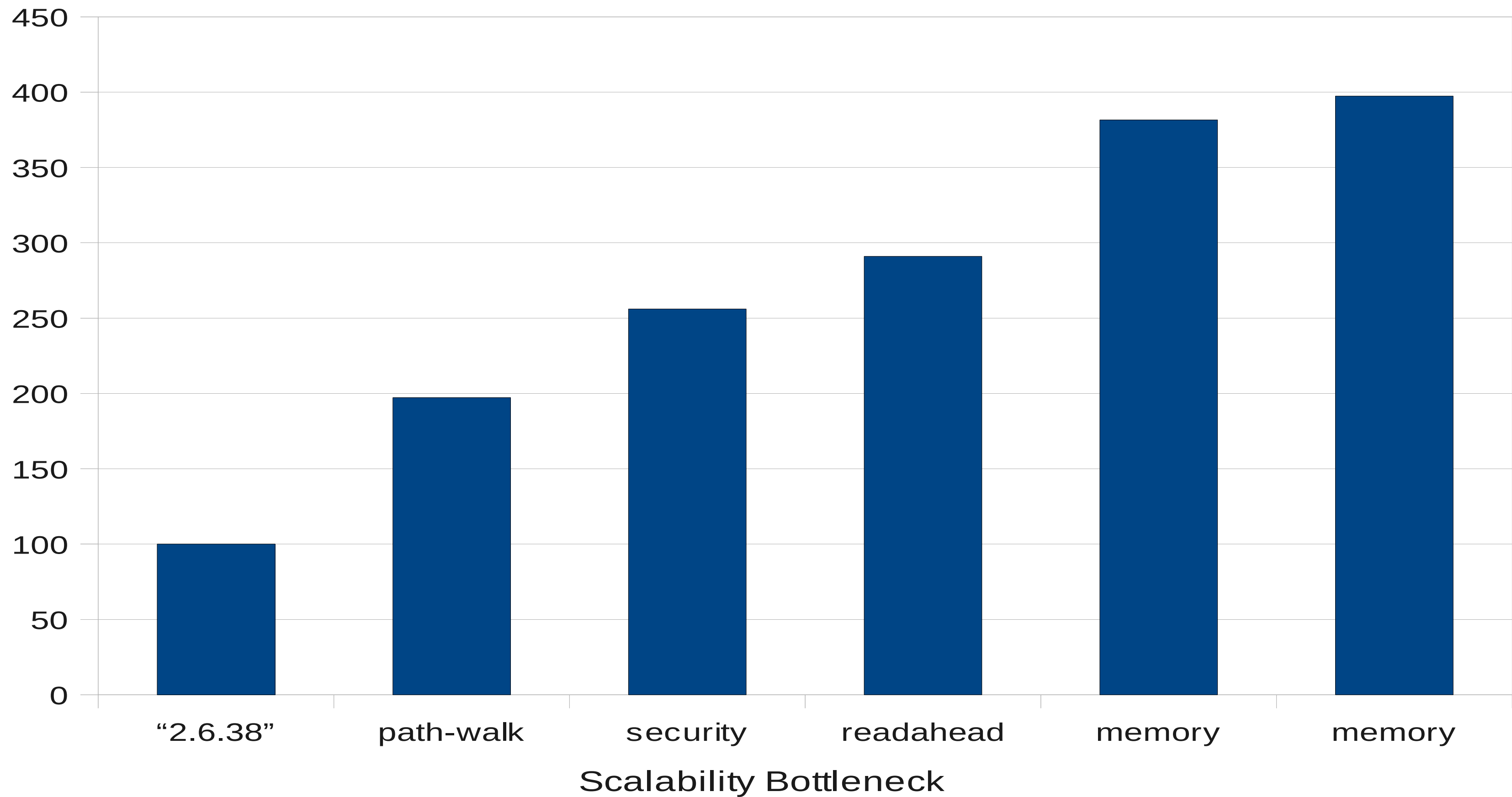
- Exim makes use of Berkeley DB library for data management. Frequent dbfn_open calls for new exim threads.
- dbfn_open calls glibc's sysconf() to get the number of CPUs to tune its locking.
- Reads /proc/stat which is very expensive.
- Switch libc to use a direct system call to obtain the number of cpus.
- Patches not added due to disagreement between libc/kernel. But you can use http://halobates.de/smallsrc/sysconf.c as LD_PRELOAD.

Profile after Fix #7

370.4 msgs/sec/core (+18.3 msgs/sec/core)

```
+     4.84%      exim  [kernel.kallsyms]      [k] page_fault
+     3.83%      exim  [kernel.kallsyms]      [k] clear_page_c
-     3.25%      exim  [kernel.kallsyms]      [k] do_raw_spin_lock
   - do_raw_spin_lock
      - 91.86% _raw_spin_lock
         + 14.16% unlink_anon_vmas
         + 12.54% unlink_file_vma
         + 7.30% anon_vma_clone_batch
         + 6.17% dup_mm
         + 5.77% __do_fault
         + 5.77% __pte_alloc
         ...
+     3.22%      exim  [kernel.kallsyms]      [k] unmap_vmas
+     2.27%      exim  [kernel.kallsyms]      [k] page_cache_get_speculative
+     2.02%      exim  [kernel.kallsyms]      [k] copy_page_c
+     1.63%      exim  [kernel.kallsyms]      [k] __list_del_entry
```

Summary: Scalability Bottlenecks in 2.6.38



But the next regression hit shortly after:

2.6.39(vanilla) 100.0%

2.6.39+readahead-fix 166.7% (+66.7%)

Anon VMA lock change in 3.0 (spin lock -> mutex)

3.0-rc2(vanilla) 68.0% (-32%)

After a lot of tweaking from Linus and others:

3.0-rc2+fixes 140.3% (+40.3%)

(anon_vma clone + unlink + chain_alloc_tweak)

- **Lost 26% again compared to 2.6.39+rafix**

Summary Exim:

- Relatively simple workload exposed lots of scalability problems in the kernel
- Mutexes and anon vma are still a serious problem
- Looking for other good workloads with similar properties
- Anyone have any?

Network stack

- Testing MOSBENCH memcached workload over Ethernet.
- Load generator talking to 4S server.

Neighbour cache scalability

When no other TCP connection between load generator/server.
Reference count changes in neighbor structure is expensive when it is done for every message.

```
-    27.06%    memcached  [kernel.kallsyms]         [k] atomic_add_unless.clone.34
     - atomic_add_unless.clone.34
     - neigh_lookup
         __neigh_lookup_errno.clone.17
         arp_bind_neighbour
         rt_intern_hash
         __ip_route_output_key
         ip_route_output_flow
         udp_sendmsg

-    13.33%    memcached  [kernel.kallsyms]         [k] atomic_dec_and_test
     - atomic_dec_and_test
     - dst_destroy
       - dst_release
       - skb_dst_drop.clone.55
```

Avoid Neighbour Cache by establishing TCP Connection: Now routing cache ref count

```
20.54%    memcached  [kernel.kallsyms]    [k] _atomic_dec_and_lock
          |   <+> _atomic_dec_and_lock
          |   [.] inet_putpeer
          |   [.] ipv4_dst_destroy
          |   [.] dst_destroy
          |   [.] dst_release
12.48%    memcached  [kernel.kallsyms]    [k] inet_getpeer
          |   [.] inet_getpeer
          |   [.] inet_getpeer_v4
          |   [.] rt_set_nexthop.clone.30
          |   [.] __ip_route_output_key
          |   [.] ip_route_output_flow
          |   [.] udp_sendmsg
          |   [.] inet_sendmsg
          |   [.] __sock_sendmsg
          |   [.] sock_sendmsg
          |   [.] __sys_sendmsg
          |   [.] sys_sendmsg
          |   [.] system_call_fastpath
          |   [.] __sendmsg
11.80%    memcached  [kernel.kallsyms]    [k] addr_compare
 3.09%    memcached  [kernel.kallsyms]    [k] do_raw_spin_lock
```

Routing Cache: What to Do?

- Feedback from network maintainers: disable routing cache.
- echo 0 > /proc/sys/net/ipv4/rt_cache_rebuild_count

(bonus price for most obscure way to do important tunable)

Now INET PEER shows up (route cache disabled, TCP connection)

```
15.97%  memcached  [kernel.kallsyms]    [k] _raw_spin_lock
            |
            --- _raw_spin_lock
               |
               |---- _atomic_dec_and_lock
               |          inet_putpeer
               |          ipv4_dst_destroy
               |          dst_destroy
               |          dst_release
               |          dev_hard_start_xmit
               |          dev_queue_xmit
               |          neigh_resolve_output
               |          ip_finish_output2
10.97%  memcached  [kernel.kallsyms]    [k] _raw_spin_lock_bh
            |
            --- _raw_spin_lock_bh
               |
               |---- inet_getpeer
               |          rt_set_nexthop
               |          __ip_route_output_key
               |          ip_route_output_flow
               |          udp_sendmsg
```

INET PEER

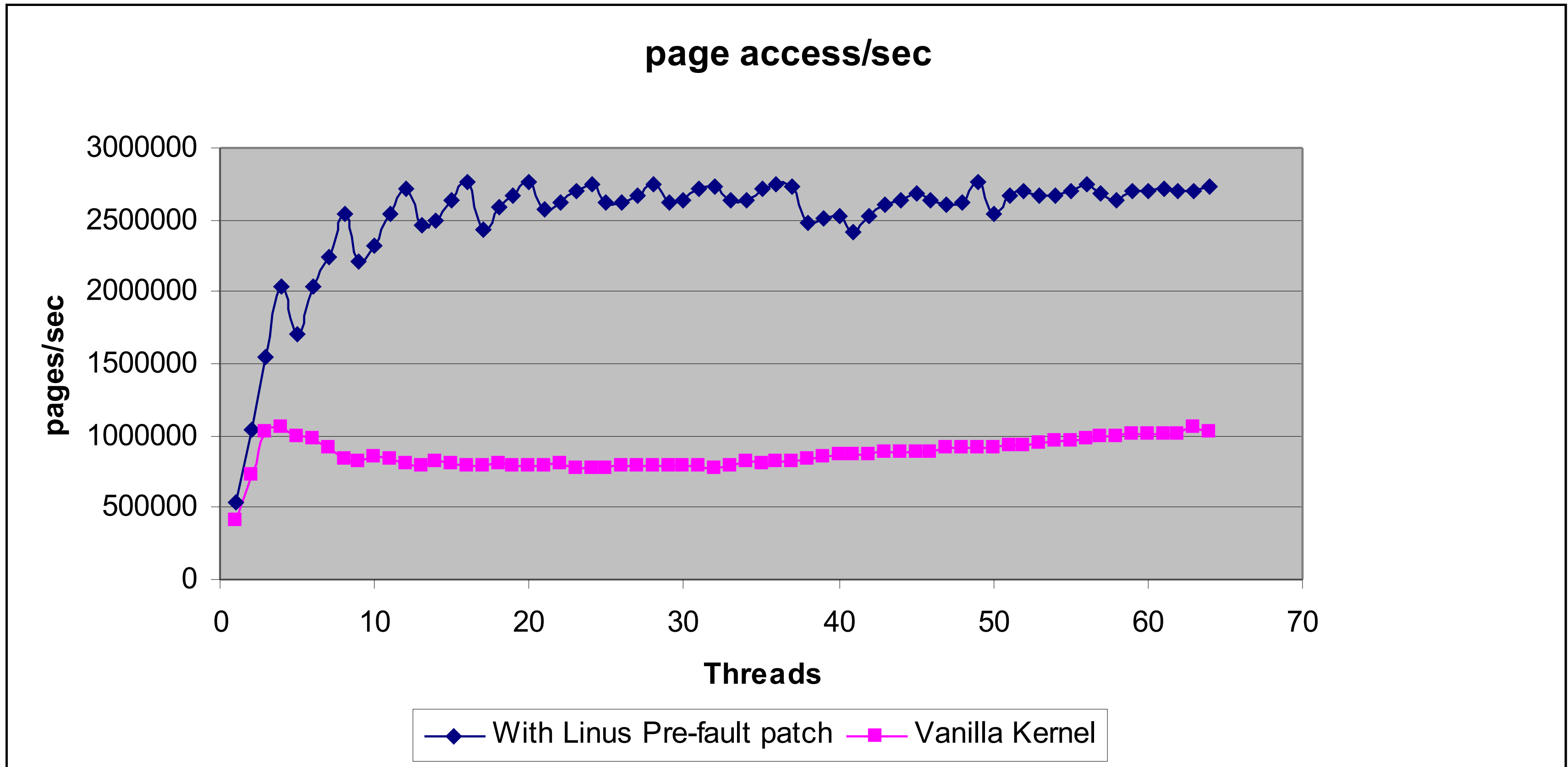
- Used to cache information for destination IP addresses.
- insert/remove peers from unused_peers.list, contending on unused_peers spin lock.
- Constantly flip peers refcnt between 0 and 1.
- Solution was to remove the unused_peers list and perform garbage correction on-the-fly at lookup time (by Eric Dumazet).

Summary networking

- Biggest problems are various reference counts
- Some workarounds/tunings are unexpected
  - “open ssh connection” avoids neighbor cache ref count
- Routing cache is a big problem
  - But you can turn it off
- Defaults out of the box don't scale well

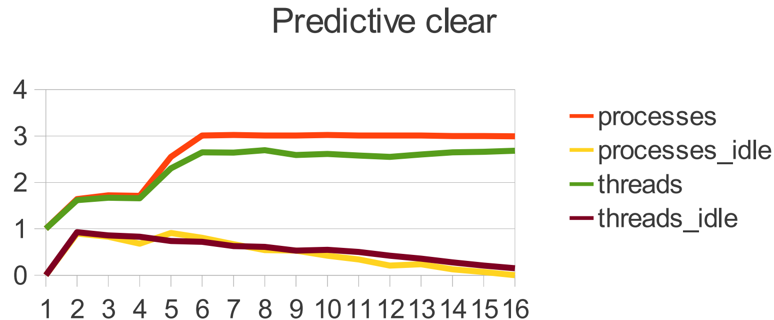
VM

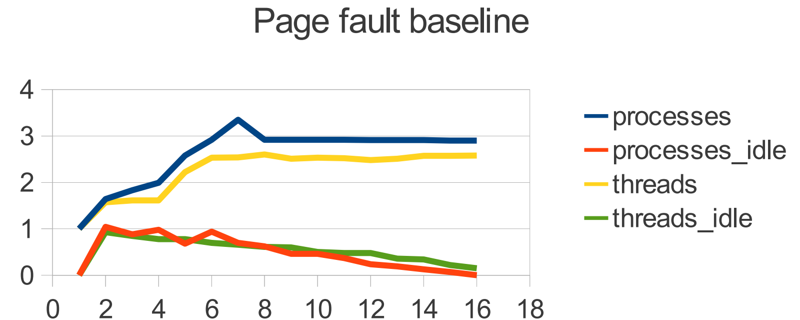
mmap_sem per process



Experimental pre-fault patch improves baseline, but does not give real scalability

Predictive page clearing outside lock





- Biggest cost inside lock is usually page clearing (for anonymous).
- Idea: move clearing "predictively" outside.
- Increases thread scaling to same as process scaling

Page fault profile: processes

```
-    45.63%  page_fault1_pro  [kernel.kallsyms]     [k] clear_page_c
   + clear_page_c
   + __alloc_pages_nodemask
-     7.43%  page_fault1_pro  [kernel.kallsyms]     [k] _raw_spin_lock
   - _raw_spin_lock
      + 47.95% handle_pte_fault
      + 28.47% free_pcppages_bulk
      + 20.05% get_page_from_freelist
```

- Limited by page table lock, zone lock
- Transparent huge pages are also costly (disabled here)
- Thread case still limited by mm_sem

Zone LRU Lock

- Does not scale well. Problem is too many cores on a node now.
- One example was workload where activate_page() is frequently used, such as read on mmaped sparse file shared between processes
  - Activate pages in batches. This approach was merged in v3.0
- Acquired also when adding pages to a zone's lru_cache and getting pages from freelist in a zone. For page fault tests by multiple processes, we're spending 40% of cpu time contending this lock.
- No general fix available so far. Do more batching?

Conclusion

- Scalability is like an onion:
  - one bottleneck fixed exposes the next
- This was just a few selected problems.
- Many more problems in the kernel.
- Still it scales reasonably for many workloads: but there are always more problems to fix.
- Interested in similar scalability problems you encounter.

Backup

Scaling Macro Benchmark Suites

- Multicore Operating System Benchmarks - MOSBENCH
  - Macro Benchmark suite
    - Exim – mail server benchmark
    - Memcache – object cache used frequently by web servers
    - Apache – web server
    - Postgres SQL – SQL database
    - Gmake – parallel build of kernel
    - Psearchy – parallel text indexer

Scaling Micro Benchmark Suites

- Will it Scale?
  - Suite of micro benchmarks with parallel execution of processes or threads, exercising basic system calls or operations concurrently
  - Originally from IBM OzLabs
  - Vary the number of processes/threads from 1 to number of cpus
  - Workload includes
    - Memory - brk, malloc/free, mmap/munmap, page fault,
    - Scheduling - context switch, sched_yield
    - Locking - futex, pthread mutex, posix semaphores
    - Files - file write, file lseek, file open/close, socket read/write, poll of fds, eventfd read/write

Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

All dates provided are subject to change without notice.

Intel is a trademark of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2011, Intel Corporation. All rights are protected.