Mental models for modern program tuning

Andi Kleen
Intel Corporation

Jun 2016
How can we see program performance?

High level vs Army of ants

Important to get the common ants fast
“Preliminary optimization is the root of all evil”

– Donald Knuth, 1974, “Structured programming with gotos”
“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%”
Flat profiles

- Is it only for 3%?
  - Often not true for complex workloads with flat performance profile
Strategy

• Have to be aware of performance when writing code
  – At least for critical paths
  – Otherwise it needs too many changes later to get fast
  – Also need a model to understand results
Mental model

vs

vs
## Latency numbers every programmer should know

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
<td></td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 ns</td>
<td></td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
<td>14x L1 cache</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 ns</td>
<td></td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
<td>20x L2 cache, 200x L1</td>
</tr>
<tr>
<td>Compress 1K bytes with snappy</td>
<td>3000 ns</td>
<td></td>
</tr>
<tr>
<td>Send 1K bytes over 1Gbps network</td>
<td>10,000 ns</td>
<td>0.01 ms</td>
</tr>
<tr>
<td>Read 4K randomly from SSD</td>
<td>150,000 ns</td>
<td>0.15 ms</td>
</tr>
<tr>
<td>Read 1MB sequentially from memory</td>
<td>250,000 ns</td>
<td>0.25 ms</td>
</tr>
<tr>
<td>Round trip within same data center</td>
<td>500,000 ns</td>
<td>0.5 ms</td>
</tr>
<tr>
<td>Read 1MB sequentially from SSD</td>
<td>1,000,000 ns</td>
<td>1ms  4x memory</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
<td>10 ms  20x data center r-t</td>
</tr>
<tr>
<td>Read 1MB sequentially from disk</td>
<td>20,000,000 ns</td>
<td>20 ms  20x SSD</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000 ns</td>
<td>150 ms</td>
</tr>
</tbody>
</table>

Originally from Peter Norvig. Generic numbers. Does not represent a particular part.
Critical bottleneck

CPU bound

Could be bound on other things:

IO
Network
GPU

For finding bottlenecks in other resources see Brendan Gregg's talks
What is CPU bound?

- Computation
- Accessing memory
- Communicating with other cores
What versus where

• Counting
  – What accurately but not where
  – Look at ratios

• Sampling
  – Loosely where, but not always what
  – Need more than just cycles

• Tracing
  – Where very accurately, but lots of data
  – Not necessarily what
Sampling skid

- Sampling may not find the exact place
- Use Precise Event Based Sampling (:pp) to minimize skid
  - Does not work in VMs

```
perf record -e cycles:pp ..
```

More PEBS events available, see ocperf list output
Slow sampling does not accurately measure fast events

PEBS Interrupt less sampling allows fast sampling at reasonable overhead

Feature in recent Linux perf 4.1+
perf record -c 25000 -no-time ...
Ratios can mislead

% perf stat -e cache-references,cache-misses ./fib

fib 100000 = 13120482006059434805

Performance counter stats for './fib':

23,283  cache-references
8,483   cache-misses   # 36.434 % of all cache refs

0.000535555 seconds time elapsed

But Fibonacci is not using memory much

Use TopDown instead to find real bottleneck
TopDown methodology

Use performance counter measurement to find bottleneck in CPU pipeline
### TopDown output

<table>
<thead>
<tr>
<th>Level</th>
<th>BE/Mem</th>
<th>Backend_Bound</th>
<th>Memory_Bound</th>
<th>Bound</th>
<th>Percent</th>
<th>[Percent]</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td></td>
<td>Backend_Bound: 98.97%</td>
<td>Memory_Bound: 96.09%</td>
<td>Bound:</td>
<td>100.00%</td>
<td></td>
</tr>
<tr>
<td>C0-T0</td>
<td></td>
<td>Backend_Bound.Memory_Bound: 96.07%</td>
<td>Memory_Bound.MEM_Bound: 96.07%</td>
<td>Bound:</td>
<td>100.00%</td>
<td></td>
</tr>
</tbody>
</table>

- This metric estimates how often the CPU was stalled on accesses to external memory (DRAM) by loads...
- Sampling events: mem_load_uops_retired.l3_miss:pp

<table>
<thead>
<tr>
<th>C0-T0</th>
<th>BOTTLENECK Backend_Bound.Memory_Bound.MEM_Bound 96.07%</th>
<th>Bound:</th>
<th>100.00%</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0-T1</td>
<td>BE/Mem</td>
<td>Backend_Bound.Memory_Bound.L1_Bound: 18.54%</td>
<td>Bound:</td>
</tr>
</tbody>
</table>

- This metric estimates how often the CPU was stalled without loads missing the L1 data cache...
- Sampling events: mem_load_uops_retired.l1_hit:pp mem_load_uops_retired.hit_lfb:pp

| C0-T1 | BOTTLENECK Backend_Bound.Memory_Bound.L1 Bound 18.54% | Bound: | 100.00% |

---

toplev tool in [http://github.com/andikleen/pmu-tools](http://github.com/andikleen/pmu-tools)
Basic performance unit

- Less than that is not cheaper

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU cache lines</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Pages</td>
<td>4K / 2MB (1GB)</td>
</tr>
<tr>
<td>Network IO packets</td>
<td>Hundreds of bytes</td>
</tr>
<tr>
<td>Network IO connection</td>
<td>Thousands of bytes</td>
</tr>
<tr>
<td>Block IO</td>
<td>Several MB</td>
</tr>
</tbody>
</table>
Simplified cache model

- 64byte cache lines
- Local
  - Temporal
  - Predictible
  - Cold
- Communication
  - Shared read-only
  - Bouncing
Temporal

- What we recently used
- What fits in caches of our working set

- Best performance if working set fits
Cold

- Not temporal, large or thrashing
- Can be quite slow: several orders of magnitude
  - Some optimizations possible: NUMA locality or interleaving using libnuma

- When data is not reused can use non temporal loads/stores to avoid polluting caches
  - But only for very large data sets
Predictible

- Large, not hot
- But predictable access pattern
- Cache can prefetch in background and hide latency

- Arrays are usually good, pointers can be bad
Prefetchers 101

- Only work for larger amounts of data
  - Need training
- Support strides and forward/backward stream
- Multiple streams supported, but only small number
Kernel software prefetching

- Event loop

  offset += pread(shared_fd, buffer, size, offset)

  vs

- read(private_fd, buffer, size);
Hot loop model

• We understand everything about the hot loop

• Can use cache blocking and other tricks to optimize temporal locality for everything
  – Compiler may even be able to do automatically

• Unfortunately a lot of software is not like that
Library model

- Called by (random) other code which has own cache footprint

- Tradeoff: more computation or more caching/table lookup
  - Tragedy of the commons: if all together use too much everyone suffers

- Non composable problem
  - Everything depends on everything

Longer essay on the topic: http://halobates.de/blog/p/227
Cache friendly libraries

- May need to optimize for cache cold
  - Minimize foot print
  - But only if memory is not cheaper
- Cluster hot fields in data structures together.
- Support big and small versions
  - Add knobs for foot print
  - Could use automatic tuning
Cache coloring

- Caches can store addresses at specific multiplies of 64 only in limited number of positions
Inefficient use of cache

- Metadata always at same offset in data page
  - Only uses fraction of the cache for pointers
  - Can use separate packed metadata instead
How to find cache issues

- First check TopDown Memory Bound
- Count cache misses
  - `perf stat -e cache-references,cache-misses ...
- Sample L3 cache misses:
  - `ocperf.py record -e \mem_load_uops_l3_miss_retired.local_dram:pp ...
- Sample addresses
  - `perf mem record ...

http://github.com/andikleen/pmu-tools
for ocperf
Automated icache tuning

- Some cache problems (like instruction cache) can be automatically optimized
  - Indicated by FrontendBound in TopDown
  - Split hot and cold code using compiler profile feedback
  - Can be done automatically from profiles without instrumentation

http://github.com/google/autofdo with gcc 5.x+

gcc 5 compile performance with profile feedback
Noisy neighbour problem

- Caches can be shared resources
- Other processes/threads/VMs cause slowdowns

- Co-locate processes with different characteristics
  - For example low IPC and high IPC

- New hardware capabilities:
  - Cache QoS monitoring
  - Cache QoS enforcement
Measure L3 cache occupancy for a process/container/VM

Gives us the upper consumption and avoids co-placement problems

Requires Intel Xeon E5 v3 and Linux 4.1+

multichase L3 occupancy
Communication

- Communication happens when a core accesses a cache line written by another core
- Like message passing in a fast network
Finding Communication

- **Check for TopDown Contested_Accesses**

- **Find with:**
  
  ocperf.py stat -e \
  mem_load_uops_l3_hit_retired.xsnp_hitm ...

  ocperf.py record -g -e \
  mem_load_uops_l3_hit_retired.xsnp_hitm:pp ...
  - Or similar event name
Latency levels

- SMT thread<->other thread
  - Really fast
- Core<->other Core in socket
- Other socket
  - Latency depends on home address
  - Possibly multiple hops

Tool to find latencies:
Intel Latency Checker

http://www.intel.com/software/mlc
Cache queueing effects

- Cache communication is message passing
  - Has ordering requirements

- Can cause conflicts and queue delays
  - Gets worse with more and more cores

Time to do 250k increments on shared counter
Cache messaging optimizations

- Avoid unnecessary round trips
- Avoid false sharing
- Prefer nearby thread/core/socket
- Design for load, use backoffs
- Avoid thundering herds
Example

- global_flag = true;

- vs

- if (!global_flag)
  - global_flag = true;
Classic Lock optimization

- Start with big lock per subsystem
- Push down to fine grained data objects to lower contention
- End goal: lock per cache line?
Lock overhead

• Locks have overhead even when cache line is not contended
  − Can depend on nearby operations (sometimes slow)
  − Can be 3 orders of magnitude delta even when not contended

• Too many locks can be costly
  − “Locks as red tape”
Another problem with small locks:

/*
 * Lock ordering in mm:
 * 
 * inode->i_mutex (while writing or truncating, not reading or faulting)
 * mm->mmap_sem
 * page->flags PG_locked (lock_page)
 * hugetlbfs_i_mmap_rwsem_key (in huge_pmd_share)
 * mapping->i_mmap_rwsem
 * anon_vma->rwsem
 * mm->page_table_lock or pte_lock
 * zone->lru_lock (in mark_page_accessed, isolate_lru_page)
 * swap_lock (in swap_duplicate, swap_info_get)
 * mmlist_lock (in mmput, drain_mmlist and others)
 * mapping->private_lock (in __set_page_dirty_buffers)
 * mem_cgroup_[begin,end]_page_stat (memcg->move_lock)
 * mapping->tree_lock (widely used)
 * inode->i_lock (in set_page_dirty's __mark_inode_dirty)
 * bdi.wb->list_lock (in set_page_dirty's __mark_inode_dirty)
 * sb_lock (within inode_lock in fs/fs-writeback.c)
 * mapping->tree_lock (widely used, in set_page_dirty,
 * in arch-dependent flush_dcache_mmap_lock,
 * within bdi.wb->list_lock in __sync_single_inode)
 * anon_vma->rwsem,mapping->i_mutex (memory_failure, collect_procs_anon)
 * ->tasklist_lock
 * pte map lock
*/
Lock region size versus work

Lock acquisition needs to amortize communication
Requires doing enough work inside lock
Lock stability

- Short lock regions can be unstable
  - Small timing variations can cause big performance changes
  - Due to timing and queueing effects on the locks and data
- Rule of thumb: critical section at least 200us for stable behavior
  - That is a lot of instructions!

In depth paper:
Library design for lock batching

- Design interfaces to process multiple operations in the same call
  - Allows doing more work in lock regions
  - Improves temporal locality
Lock elision

• Can use hardware transactional memory support with TSX to do lock elision

• Use coarse lock and let the hardware figure it out

• Lock cache line stays shared
  – Only uses real lock on real conflict
  – Still need to minimize common data conflicts

Available on Intel Xeon v4 or Xeon E7 v3
http://www.intel.com/software/tsx

Measure:  perf stat -T ...
Find aborts:  perf record -e tx-aborts:pp ....
Microbenchmarks are difficult

- CPU Frequency can change
- Accurate timing is tricky
- Compilers are good at optimizing them away
- Caching effects are difficult to reproduce
  - For example calling library with always the same input always hits cache
  - Calling always with different input thrashes caches
  - Both is unrealistic
Automatic micro benchmarking

- Using timed last branch records in Skylake CPUs
- Sample real workload and get cycle count of 32 program blocks per sample

```
$ perf record -b ...
$ perf report
```

Needs Linux perf 4.3+

Indepth articles:
http://lwn.net/Articles/680985/
http://lwn.net/Articles/680996
Tracing

• Using Processor Trace (PT)
  – Hardware feature in Broadwell/Skylake
  – Supported in Linux perf since Linux 4.1
  – Fine grained execution trace with time stamps

$ perf record -e intel_pt// ...
$ perf script --ns --itrace=cr \-F time,event,callindent,addr,sym

For older kernels
http://github.com/andikleen/simple-pt
Assembler Tracing

- Using Processor Trace (PT)
- Timing cycle accurate to ~4 basic blocks on Skylake

$ perf record -e intel_pt/ ...$
$ perf script --ns --itrace=i0ns \ -F time,pid,comm,ip,asm

Requires patched perf for disassembler

Indepth article on PT:
http://lwn.net/Articles/648154/
Summary

• Focus on critical bottlenecks
• Remember the order of magnitudes
• Cache communication is message passing
• Lock coarsely
• Measure properly

• http://github.com/andikleen/pmu-tools
• http://halobates.de
Backup
Autotuning

- Add knobs to size tables and algorithms and use an auto tuner to find best trade off for whole program
  - Can adapt to changing circumstances
  - Use generic optimization frameworks

ATLAS: Automatically tuned linear algebra kernels

http://github.com/jansel/opentuner
Cache events

- Communication:
  - `mem_load_uops_l3_hit_retired.xsnp_hit`
  - `mem_load_uops_l3_hit_retired.xsnp_hitm`
  - `mem_load_uops_l3_hit_retired.xsnp_miss`

- Locality:
  - `mem_load_uops_retired.l1_miss` / hit
  - `mem_load_uops_retired.l2_miss` / hit
  - `mem_load_uops_retired.l3_miss` / hit

- Can be counted or sampled with ocperf in pmu-tools
Linked lists versus ropes

A → B → C → D → E → F

VS

A B C D → D E F