How Netflix Tunes EC2 Instances for Performance

Brendan Gregg, Performance and OS Engineering Team

November 28, 2017
Netflix performance and operating systems team

- **Evaluate technology**
  - Instance types, Amazon Elastic Compute Cloud (EC2) options

- **Recommendations and best practices**
  - Instance kernel tuning, assist app tuning

- **Develop performance tools**
  - Develop tools for observability and analysis

- **Project support**
  - New database, programming language, software change

- **Incident response**
  - Performance issues, scalability issues
Agenda

1. Instance selection
2. Amazon EC2 features
3. Kernel tuning
4. Methodologies
5. Observability
Warnings

- This is what’s in our medicine cabinet
- Consider these “best before: 2018”
- Take only if prescribed by a performance engineer
1. Instance selection
The Netflix cloud

Many application workloads: Compute, storage, caching...
Netflix AWS environment

- Elastic Load Balancing allows real load testing
  1. Single instance canary, then,
  2. Auto scaling group
- Much better than micro-benchmarking alone, which is error prone
Current generation instances

• Families:
  – **m4**: General purpose
    • Balanced
  – **c5**: Compute-optimized
    • Latest CPUs, lowest price/compute perf
  – **i3, d2**: Storage-optimized
    • SSD large capacity storage
  – **r4, x1**: Memory optimized
    • Lowest cost/Gbyte
  – **p2, g3, f1**: Accelerated computing
    • GPUs, FPGAs...

• Types: Range from medium to 16x large+, depending on family

• Netflix uses over 30 different instance types
Netflix instance type selection

A. Flow chart
B. By-resource
C. Brute force
A. Instance selection flow chart

1. **Start**
2. **Need large disk capacity?**
   - **Y**: **d2**
   - **N**: **Disk I/O bound?**
3. **Disk I/O bound?**
   - **Y**: **Exceptions**
   - **N**: **Can cache?**
4. **Can cache?**
   - **Y**: **Select memory to cache working set**
   - **N**: **Find best balance**
5. **Exceptions**
   - **Trade-off**
     - **Memory**
     - **CPU**
B. By-resource approach

1. Determine bounding resource
   - E.g.: CPU, disk I/O, or network I/O
   - Found using:
     • Estimation (expertise)
     • Resource observability with an existing real workload
     • Resource observability with a benchmark or load test (experimentation)

2. Choose instance type for the bounding resource
   - If disk I/O, consider caching, and a memory-optimized type
   - We have tools to aid this choice: Nomogram Visualization

This focuses on optimizing a given workload
More efficiency can be found by adjusting the workload to suit instance types
Nomogram Visualization tool

1. Select instance families

2. Select resources

3. From any resource, see types and cost

(cost redacted)
C. Brute force choice

1. Run load test on ALL instance types
   - Optionally, different workload configurations as well
2. Measure throughput
   - And check for acceptable latency
3. Calculate price/performance for all types
4. Choose most efficient type
Latency requirements

- Check for an acceptable latency distribution when optimizing for price/performance.
Netflix instance type re-selection

A. Usage
B. Cost
C. Variance
A. Instance usage

- Older instance types can be identified, analyzed, and upgraded to newer types
B. Instance cost

- Also checked regularly. Tuning the price in price/perf.
C. Instance variance

• An instance type may be resource-constrained only occasionally, or after warmup, or a code change

• Continually monitor performance, analyze variance/outliers
2. Amazon EC2 features
## EC2 virtualization

<table>
<thead>
<tr>
<th>Tech</th>
<th>Type</th>
<th>With</th>
<th>CPU</th>
<th>Memory</th>
<th>Network I/O</th>
<th>Remote Storage I/O</th>
<th>Inturpts: Timers</th>
<th>Motherboard: Boot</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>Fully Emulated</td>
<td>VH P P P P</td>
<td>VS</td>
<td>VS</td>
<td>VS</td>
<td>VS</td>
<td>VS</td>
<td>VS</td>
</tr>
<tr>
<td>Old</td>
<td>VM Xen HVM 3.0</td>
<td>P</td>
<td>VH</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>VM Xen HVM 4.0.1</td>
<td>P</td>
<td>VH</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>VM Xen AWS 2013</td>
<td>P</td>
<td>VH</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>VM Xen AWS 2017</td>
<td>P</td>
<td>VH</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>VM AWS Nitro 2017</td>
<td>VH</td>
<td>VH</td>
<td>VH</td>
<td>VH</td>
<td>VH</td>
<td>VH</td>
<td>VS</td>
</tr>
<tr>
<td>New</td>
<td>HW AWS Bare Metal 2017</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

**VM**: Virtual Machine. **HW**: Hardware.
**VS**: Virt. in software. **VH**: Virt. in hardware. **P**: Paravirt. Not all combinations shown.
**SR-IOV(net)**: ixgbe/ena driver. **SR-IOV(storage)**: nvme driver.

---

*Slide updated after talk. see: http://www.brendangregg.com/blog/2017-11-29/aws-ec2-virtualization-2017.html*
Networking SR-IOV

• AWS "enhanced networking"
  - Uses SR-IOV: Single Root I/O Virtualization
  - PCIe device provides virtualized instances
  - Some instance types, VPC only

• "Bare metal" network access
  - Higher network throughput, reduced RTT and jitter
  - ixgbe driver types: Up to 10 Gbps
  - ena driver types: Up to 25 Gbps
Storage SR-IOV

- New in 2017, first used by i3s
- Should be called "enhanced storage"
  - Some instance types only
  - Accesses NVMe attached storage (faster transport than SATA)
  - Uses VT-d for I/O virtualization
- "Bare metal" disk access
  - i3.16xl can exceed 3 million IOPS

3. Kernel tuning
Kernel tuning

• Typically 1-30% wins, for average performance
  - Adds up to significant savings for the Netflix cloud
• Bigger wins when reducing latency outliers
• Deploying tuning:
  - Generic performance tuning is baked into our base AMI
  - Experimental tuning is a package add-on (nflx-kernel-tunables)
  - Workload-specific tuning is configured in application AMIs
  - Remember to tune the workload with the tunables
• We run Ubuntu Linux
Tuning targets

1. CPU scheduler
2. Virtual memory
3. Huge pages
4. NUMA
5. File System
6. Storage I/O
7. Networking
8. Hypervisor (Xen)
1. CPU scheduler

• Tunables:
  - Scheduler class, priorities, migration latency, tasksets...

• Usage:
  - Some apps benefit from reducing migrations using taskset(1), numactl(8), cgroups, and tuning sched_migration_cost_ns
  - Some Java apps have benefited from SCHED_BATCH, to reduce context switching. E.g.:

```
# schedtool -B PID
```
2. Virtual memory

• Tunables:
  – Swappiness, overcommit, OOM behavior…

• Usage:
  – Swappiness is set to zero to disable swapping and favor ditching the file system page cache first to free memory. (This tunable doesn’t make much difference, as swap devices are usually absent.)

```bash
vm.swappiness = 0           # from 60
```
3. Huge pages

- **Tunables:**
  - Explicit huge page usage, transparent huge pages (THPs)
  - Using 2 or 4 Mbytes, instead of 4k, should reduce various CPU overheads and improve MMU page translation cache reach

- **Usage:**
  - THPs (enabled in later Ubuntu kernels) depending on the workload and CPUs, sometimes improve perf on HVM instances (~5% lower CPU), but sometimes hurt perf (~25% higher CPU during %usr, and more during %sys refrag)
  - We switched it back to madvise:

```bash
# echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
```
4. NUMA

• Tunables:
  – NUMA balancing

• Usage:
  – On multi-NUMA systems (largest instances) and earlier kernels (around 3.13), NUMA page rebalance was too aggressive, and could consume 60% CPU alone.
  – We disable it. Will re-enable/tune later.

  kernel.numa_balancing = 0
5. File system

• Tunables:
  - Page cache flushing behavior, file system type and its own tunables (e.g., ZFS on Linux)

• Usage:
  - Page cache flushing is tuned to provide a more even behavior: Background flush earlier, aggressive flush later
  - Access timestamps disabled, and other options depending on the FS

```bash
vm.dirty_ratio = 80                     # from 40
vm.dirty_background_ratio = 5           # from 10
vm.dirty_expire_centisecs = 12000       # from 3000
mount -o defaults,noatime,discard,nobARRIER ...  
```
6. Storage I/O

• Tunables:
  - Read ahead size, number of in-flight requests, I/O scheduler, volume stripe width...

• Usage:
  - Some workloads, e.g., Cassandra, can be sensitive to read ahead size
  - SSDs can perform better with the “noop” scheduler (if not default already)
  - Tuning md chunk size and stripe width to match workload

```
/sys/block/*/queue/rq_affinity 2
/sys/block/*/queue/scheduler noop
/sys/block/*/queue/nr_requests 256
/sys/block/*/queue/read_ahead_kb 256
mdadm --chunk=64 ...
```
7. Networking

- Tunables:
  - TCP buffer sizes, TCP backlog, device backlog, TCP reuse...
- Usage:

  ```
  net.core.somaxconn = 1024
  net.core.netdev_max_backlog = 5000
  net.core.rmem_max = 16777216
  net.core.wmem_max = 16777216
  net.ipv4.tcp_wmem = 4096 12582912 16777216
  net.ipv4.tcp_rmem = 4096 12582912 16777216
  net.ipv4.tcp_max_syn_backlog = 8096
  net.ipv4.tcp_slow_start_after_idle = 0
  net.ipv4.tcp_tw_reuse = 1
  net.ipv4.ip_local_port_range = 10240 65535
  net.ipv4.tcp_abort_on_overflow = 1   # maybe
  ```
8. Hypervisor (Xen)

- **Tunables:**
  - PV/HVM (baked into AMI)
  - Kernel clocksource. From slow to fast: hpет, xen, tsc

- **Usage:**
  - We’ve encountered a Xen clocksource regression in the past (Ubuntu Trusty). Fixed by tuning clocksource to TSC (although beware of clock drift).
  - Best case example (so far): CPU usage reduced by 30%, and average app latency reduced by 43%.

```bash
echo tsc > /sys/devices/system/clocksource/clocksource0/current_clocksource
```
4. Methodologies

Techniques of performance analysis
Checklists: e.g., Netflix perf vitals dashboard

1. RPS, CPU
2. Volume
3. Instances
4. Scaling
5. CPU/RPS
6. Load avg
7. Java heap
8. ParNew
9. Latency
10. 99th tile
Analysis perspectives

Load → Workload analysis

Application → System libraries

System calls → Kernel

Kernel → Devices

Resource analysis
USE Method

• For every hardware and software resource, check:
  1. Utilization
  2. Saturation
  3. Errors

• Resource constraints show as saturation or high utilization
  – Resize or change instance type
  – Investigate tunables for the resource

• The USE Method poses questions to answer
On-CPU and off-CPU analysis

Can be analyzed using:
- On-CPU: Sampling
- Off-CPU: Scheduler tracing
5. Observability

Finding, quantifying, and confirming tunables

Discovering system wins (5-25%’s) and application wins (2-10x’s)
Statistical tools

- vmstat, pidstat, sar, etc., used mostly normally

```
$ sar -n TCP,ETC,DEV 1
Linux 3.2.55 (test-e4f1a80b)       08/18/2014    _x86_64_ (8 CPU)

    09:10:43 PM  IFACE  rxpck/s  txpck/s  rxkB/s  txkB/s  rxcmp/s  txcmp/s  rxmcst/s
    09:10:44 PM   lo       14.00    14.00    1.34     1.34    0.00    0.00      0.00
    09:10:44 PM  eth0    4114.00  4186.00  4537.46 28513.24    0.00    0.00      0.00

          09:10:43 PM  active/s  passive/s  iseg/s  oseg/s
          09:10:44 PM   21.00      4.00   4107.00  22511.00

          09:10:43 PM  atmptf/s  estres/s  retrans/s  isegerr/s  orsts/s
          09:10:44 PM    0.00      0.00  36.00    0.00     1.00

[...]```
Host perf analysis in 60s

1. uptime  ----------------------------------------► load averages
2. dmesg | tail  ----------------------------------------► kernel errors
3. vmstat 1  ----------------------------------------► overall stats by time
4. mpstat -P ALL 1  ----------------------------------------► CPU balance
5. pidstat 1  ----------------------------------------► process usage
6. iostat -xz 1  ----------------------------------------► disk I/O
7. free -m  ----------------------------------------► memory usage
8. sar -n DEV 1  ----------------------------------------► network I/O
9. sar -n TCP,ETCP 1  ----------------------------------------► TCP stats
10. top  ----------------------------------------► check overview

System profilers

- **perf**
  - Standard Linux profiler. In the Linux source tree.
  - Interval sampling, CPU performance counter events.
  - User and kernel static and dynamic tracing.

- **perf CPU flame graphs:**

  ```
  # git clone https://github.com/brendangregg/FlameGraph
  # cd FlameGraph
  # perf record -F 49 -ag -- sleep 30
  # perf script | ./stackcollapse-perf.pl | ./flamegraph.pl > perf.svg
  ```

Java (Broken stacks: No frame pointer)

AWS re:Invent 2014

Kernel (C)

JVM (C++)

perf Flame Graph
Tracing Tools: ftrace

• Part of the Linux kernel
  – First added in 2.6.27 (2008), and enhanced in later releases
  – Already available in all Netflix Linux instances

• Front-end tools aid usage: perf-tools collection
  – https://github.com/brendangregg/perf-tools
  – Unsupported hacks: see WARNINGs
  – Also see the trace-cmd front-end, as well as perf
**ftrace tool: iosnoop**

```bash
# /apps/perf-tools/bin/iosnoop -ts
Tracing block I/O. Ctrl-C to end.

<table>
<thead>
<tr>
<th>STARTs</th>
<th>ENDS</th>
<th>COMM</th>
<th>PID</th>
<th>TYPE</th>
<th>DEV</th>
<th>BLOCK</th>
<th>BYTES</th>
<th>LATms</th>
</tr>
</thead>
<tbody>
<tr>
<td>5982800.302061</td>
<td>5982800.302679</td>
<td>supervise</td>
<td>1809</td>
<td>W</td>
<td>202,1</td>
<td>17039600</td>
<td>4096</td>
<td>0.62</td>
</tr>
<tr>
<td>5982800.302423</td>
<td>5982800.302842</td>
<td>supervise</td>
<td>1809</td>
<td>W</td>
<td>202,1</td>
<td>17039608</td>
<td>4096</td>
<td>0.42</td>
</tr>
<tr>
<td>5982800.304962</td>
<td>5982800.305446</td>
<td>supervise</td>
<td>1801</td>
<td>W</td>
<td>202,1</td>
<td>17039616</td>
<td>4096</td>
<td>0.48</td>
</tr>
<tr>
<td>5982800.305250</td>
<td>5982800.305676</td>
<td>supervise</td>
<td>1801</td>
<td>W</td>
<td>202,1</td>
<td>17039624</td>
<td>4096</td>
<td>0.43</td>
</tr>
</tbody>
</table>
```

```
# /apps/perf-tools/bin/iosnoop -h
USAGE: iosnoop [-hQst] [-d device] [-i iotype] [-p PID] [-n name] [duration]

- d device       # device string (eg, "202,1")
- i iotype       # match type (eg, '*R*' for all reads)
- n name         # process name to match on I/O issue
- p PID          # PID to match on I/O issue
- Q              # include queueing time in LATms
- s              # include start time of I/O (s)
- t              # include completion time of I/O (s)
```

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.
Tracing tools: perf

```
# perf record -e skb:consume_skb -ag -- sleep 10
# perf report

[...]

74.42%  swapper  [kernel.kallsyms]  [k] consume_skb

--- consume_skb
    arp_process
    arp_rcv
    __netif_receive_skb_core
    __netif_receive_skb
    netif_receive_skb
    virtnet_poll
    net_rx_action
    __do_softirq
    irq_exit
    do_IRQ
    ret_from_intr
```

Summarizing stack traces for a tracepoint

perf can do many things, it is hard to pick just one example
Tracing tools: BPF

- Enhanced Berkeley Packet Filter (BPF, aka eBPF)
  - Safe, efficient, advanced, production tracing. Best on Linux 4.9+.
# /usr/share/bcc/tools/tcplife

<table>
<thead>
<tr>
<th>PID</th>
<th>COMM</th>
<th>LADDR</th>
<th>LPORT</th>
<th>RADDR</th>
<th>RPORT</th>
<th>TX_KB</th>
<th>RX_KB</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2509</td>
<td>java</td>
<td>100.82.34.63</td>
<td>8078</td>
<td>100.82.130.159</td>
<td>12410</td>
<td>0</td>
<td>0</td>
<td>5.44</td>
</tr>
<tr>
<td>2509</td>
<td>java</td>
<td>100.82.34.63</td>
<td>8078</td>
<td>100.82.78.215</td>
<td>55564</td>
<td>0</td>
<td>0</td>
<td>135.32</td>
</tr>
<tr>
<td>2509</td>
<td>java</td>
<td>100.82.34.63</td>
<td>60778</td>
<td>100.82.207.252</td>
<td>7001</td>
<td>0</td>
<td>0</td>
<td>13 15126.87</td>
</tr>
<tr>
<td>2509</td>
<td>java</td>
<td>100.82.34.63</td>
<td>38884</td>
<td>100.82.208.178</td>
<td>7001</td>
<td>0</td>
<td>0</td>
<td>15568.25</td>
</tr>
<tr>
<td>2509</td>
<td>java</td>
<td>127.0.0.1</td>
<td>4243</td>
<td>127.0.0.1</td>
<td>42166</td>
<td>0</td>
<td>0</td>
<td>0.61</td>
</tr>
<tr>
<td>12030</td>
<td>upload-mes</td>
<td>127.0.0.1</td>
<td>34020</td>
<td>127.0.0.1</td>
<td>8078</td>
<td>11</td>
<td>0</td>
<td>3.38</td>
</tr>
<tr>
<td>12030</td>
<td>upload-mes</td>
<td>127.0.0.1</td>
<td>21196</td>
<td>127.0.0.1</td>
<td>7101</td>
<td>0</td>
<td>0</td>
<td>12.61</td>
</tr>
<tr>
<td>3964</td>
<td>mesos-slav</td>
<td>127.0.0.1</td>
<td>7101</td>
<td>127.0.0.1</td>
<td>21196</td>
<td>0</td>
<td>0</td>
<td>12.64</td>
</tr>
<tr>
<td>12021</td>
<td>upload-sys</td>
<td>127.0.0.1</td>
<td>34022</td>
<td>127.0.0.1</td>
<td>8078</td>
<td>372</td>
<td>0</td>
<td>15.28</td>
</tr>
<tr>
<td>2509</td>
<td>java</td>
<td>127.0.0.1</td>
<td>8078</td>
<td>127.0.0.1</td>
<td>34022</td>
<td>0</td>
<td>372</td>
<td>15.31</td>
</tr>
<tr>
<td>2235</td>
<td>dockerd</td>
<td>100.82.34.63</td>
<td>13730</td>
<td>100.82.136.233</td>
<td>7002</td>
<td>0</td>
<td>4</td>
<td>18.50</td>
</tr>
<tr>
<td>2235</td>
<td>dockerd</td>
<td>100.82.34.63</td>
<td>34314</td>
<td>100.82.64.53</td>
<td>7002</td>
<td>0</td>
<td>8</td>
<td>56.73</td>
</tr>
</tbody>
</table>

Dynamic tracing of TCP set state only; does not trace send/receive
https://github.com/iovisor/bcc includes other TCP tools
Hardware counters

- Model Specific Registers (MSRs)
  - Basic details: Timestamp clock, temperature, power
  - Some are available in Amazon EC2

- Performance Monitoring Counters (PMCs)
  - Advanced details: Cycles, stall cycles, cache misses...
  - Availability depends on instance type: either none, some, or all

- Root cause CPU usage at the cycle level
  - E.g., higher CPU usage due to more memory stall cycles
MSRs

• Can be used to verify real CPU clock rate
  - Can vary with turboboost. Important to know for perf comparisons.
  - Tool from https://github.com/brendangregg/msr-cloud-tools:

```
ec2-guest# ./showboost
CPU MHz     : 2500
Turbo MHz   : 2900 (10 active)
Turbo Ratio : 116% (10 active)
CPU 0 summary every 5 seconds...

TIME       C0_MCYC      C0_ACYC        UTIL  RATIO    MHz
06:11:35   6428553166   7457384521      51%   116%   2900
06:11:40   6349881107   7365764152      50%   115%   2899
06:11:45   6240610655   7239046277      49%   115%   2899
[...]
```

Real CPU MHz
**PMCs: Architectural**

- Some instance types (e.g., m4.16xl) support the PMC architectural set:

<table>
<thead>
<tr>
<th>Event Name</th>
<th>UMask</th>
<th>Event Select</th>
<th>Example Event Mask Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnHalted Core Cycles</td>
<td>00H</td>
<td>3CH</td>
<td>CPU_CLK_UNHALTED.THREAD_P</td>
</tr>
<tr>
<td>Instruction Retired</td>
<td>00H</td>
<td>C0H</td>
<td>INST RETIRED.ANY_P</td>
</tr>
<tr>
<td>UnHalted Reference Cycles</td>
<td>01H</td>
<td>3CH</td>
<td>CPU_CLK_THREAD_UNHALTED.REF_XCLK</td>
</tr>
<tr>
<td>LLC Reference</td>
<td>4FH</td>
<td>2EH</td>
<td>LONGEST_LAT_CACHE.REFERENCE</td>
</tr>
<tr>
<td>LLC Misses</td>
<td>41H</td>
<td>2EH</td>
<td>LONGEST_LAT_CACHE.MISS</td>
</tr>
<tr>
<td>Branch Instruction Retired</td>
<td>00H</td>
<td>C4H</td>
<td>BR_INST_RETIRED.ALL_BRANCHES</td>
</tr>
<tr>
<td>Branch Misses Retired</td>
<td>00H</td>
<td>C5H</td>
<td>BR_MISP_RETIRED.ALL_BRANCHES</td>
</tr>
</tbody>
</table>

http://www.brendangregg.com/blog/2017-05-04/the-pmcs-of-ec2.html
PMCs: All

• All PMCs are available on this c5.18xlarge:

```bash
# perf stat -d -a -- sleep 5

Performance counter stats for 'system wide':

360195.435103 cpu-clock (msec) # 71.454 CPUs utilized
38,733 context-switches # 0.108 K/sec
504 cpu-migrations # 0.001 K/sec
861,393 page-faults # 0.002 M/sec
2,275,234,239 cycles # 0.006 GHz
191,859,050,716 instructions # 84.32 insn per cycle
38,989,119,249 branches # 108.244 M/sec
152,913,791 branch-misses # 0.39% of all branches
40,262,604,776 L1-dcache-loads # 111.780 M/sec
283,924,939 L1-dcache-load-misses # 0.71% of all L1-dcache hits

[...]
```
Netflix Atlas

- Cloud-wide and instance monitoring:

  ![Atlas Dashboard]

  - **Application**
  - **Metrics**
  - **Presentation**
  - **Region**
  - **Interactive graph**
  - **Summary statistics**
  - **Time range**
Netflix Atlas

- All metrics in one system
- System metrics:
  - CPU usage, disk I/O, memory...
- Application metrics:
  - Requests completed, latency percentiles, errors...
- Filters/breakdowns by region, application, ASG, metric, instance
Netflix Vector

- Real-time per-second instance metrics:
  - Utilization
  - Per-device
  - Saturation
  - Errors
  - Breakdowns
Vector on-demand flame graphs
Vector

- Given an instance, analyze low-level performance
- On-demand flame graphs
  - CPU, off-CPU, context switch, IPC, page fault, disk I/O
  - These use perf or BPF
- Quick
  - GUI-driven root cause analysis
- Scalable
  - Other teams can use it easily
Summary

1. Instance selection
2. Amazon EC2 features
3. Kernel tuning
4. Methodologies
5. Observability
References & links

- Amazon EC2:
  - https://www.slideshare.net/AmazonWebServices/cmp402-amazon-ec2-instances-deep-dive

- Netflix on EC2:
  - http://www.slideshare.net/cpwatson/cpn302-yourlinuxamioptimizationandperformance
  - http://techblog.cloudperf.net/2016/05/2-million-packets-per-second-on-public.html
  - http://techblog.cloudperf.net/2017/04/3-million-storage-iops-on-aws-cloud.html

- Performance Analysis:
  - https://www.slideshare.net/brendangregg/velocity-2015-linux-perf-tools
  - http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html#Java
  - https://github.com/brendangregg/FlameGraph
Netflix talks @ re:Invent

Monday
10:45am ARC208: Walking the tightrope: Balancing Innovation, Reliability, Security, and Efficiency (Venetian)
12:15pm SID206: Best Practices for Managing Security on AWS (MGM)

Tuesday
10:45am ARC209: A Day in the Life of a Netflix Engineer (Venetian)
11:30am CMP325: How Netflix Tunes EC2 Instances for Performance (Venetian)

Wednesday
11:30am MCL317: Orchestrating ML Training for Netflix Recommendations (Venetian)
12:15pm NET303: A day in the life of a Cloud Network Engineer at Netflix (Venetian)
1:00pm ARC312: Why Regional Reservations are a Game Changer for Netflix (Venetian)
1:00pm SID304: SecOps 2021 Today: Using AWS Services to Deliver SecOps (MGM)
1:45pm DEV334: Performing Chaos at Netflix Scale (Venetian)
4:45pm SID316: Using Access Advisor to Strike the Balance Between Security and Usability (MGM)

Thursday
12:15pm CMP311: Auto Scaling Made Easy: How Target Tracking Scaling Policies Hit the Bullseye (Palazzo)
12:15pm DAT308: Codex: Conditional Modules Strike Back (Venetian)
12:55pm CMP309: How Netflix Encodes at Scale (Venetian)
5:00pm ABD401: How Netflix Monitors Applications Real Time with Kinesis (Aria)

Friday
8:30am ABD319: Tooling Up For Efficiency: DIY Solutions @ Netflix (Aria)
10:00am ABD401: Netflix Keystone SPaaS - Real-time Stream Processing as a Service (Aria)
Thank you!

Brendan Gregg, Netflix Performance and Operating Systems Team
@brendangregg