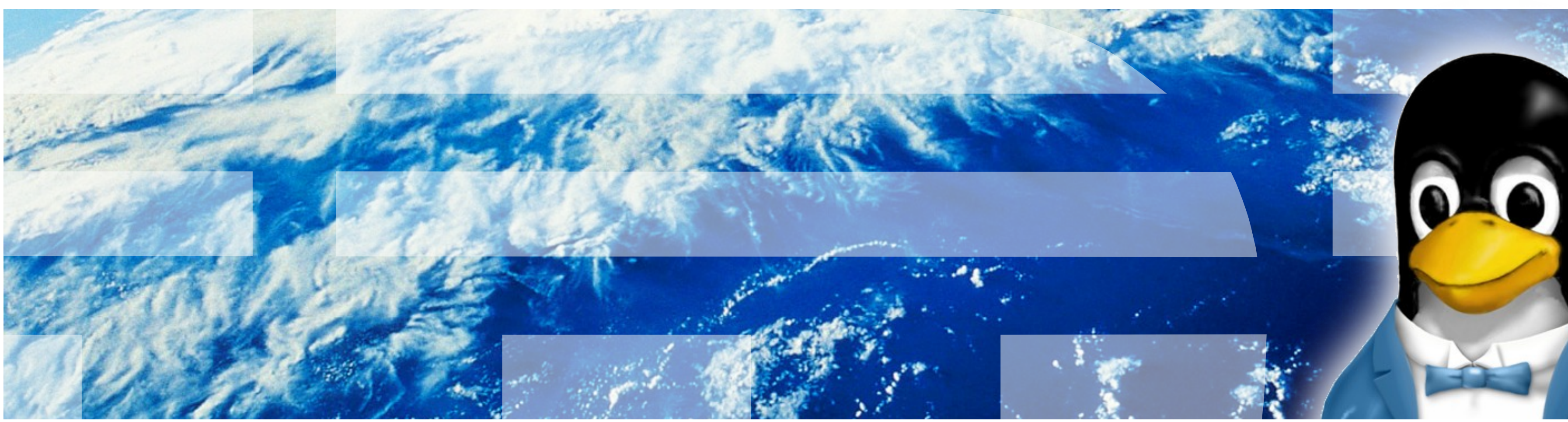
Improving the Out-of-Box KVM Performance



Andrew Theurer, IBM

atheurer@us.ibm.com

Topics

- Current performance and public benchmarks
- Example of “out of box” performance
- Some analysis of performance
- Improving performance with NUMA aware VM balancer
- Before/After test results
- Future work items
- Kernel or User?

Current KVM Performance

- **With Industry Standard Benchmarks – it is fantastic!**
  - SPECvirt_sc2010:
    - More per-core #1 results than any other hypervisor (12, 16, 20, 40, 64, 80)[1]
    - KVM results now from multiple vendors
    - KVM scaling to biggest x86_64 servers
    - As with almost any public benchmark, there is a lot of tuning to get the best result
- **Out of Box (ad-hoc testing, PoC's, user workloads) – not quite as good as above**
  - Performance analysis & tuning is generally not done here
    - Important that the hypervisor provide the best settings automatically
  - Performance can be impacted by not choosing the best options
    - Much better now with libvirt, virt-install (defaulting to virtio when possible)
    - User may not be experienced with best settings, assumes bigger is better (why have 2 vCPUs when I can have 16!!!)
    - Some of the highest performing configurations require special hardware and special configuration (does the user really know they have to enable virtual functions for that "SR-IOV" thingy?)
  - Performance is impacted by lack of NUMA optimizations for VMs
    - *This is the focus of this presentation*

[1] For all details on SPECvirt_sc2010, see spec.org

Example of Out-of-the-box Performance

- **Let's take a relatively simple test case: 40 VMs (4-way, 2 GB) and have them run Dbench (in tmpfs) at the same time on a 4 x Westmere-EX server (40 cores)**
- **Use sensible configurations (para-virtualized IO), no special optimizations**
- **Compare to “Mystery X86 Hypervisor” (MXH) with default configuration**
- **Aggregate Dbench throughput:**
  - KVM: 14541 MB/sec
  - MXH: 22919 MB/sec (58% better!?!)

Analysis – What went wrong?

- **Host CPU stats**
  - Guest: 97% Host: 3%
    - Hypervisor overhead is probably not the primary issue
- **NUMA optimization**
  - /proc/<pid>/numa_maps -where is our memory?

```
[vg-db0040(26824)]
node:[0]  pages:[0228984]  MiB:[00894]  percent[050.48]
node:[1]  pages:[0013569]  MiB:[00053]  percent[002.99]
node:[2]  pages:[0182557]  MiB:[00713]  percent[040.25]
node:[3]  pages:[0028473]  MiB:[00111]  percent[006.28]
[vg-db0039(26872)]
node:[0]  pages:[0095351]  MiB:[00372]  percent[021.05]
node:[1]  pages:[0114915]  MiB:[00448]  percent[025.37]
node:[2]  pages:[0025176]  MiB:[00098]  percent[005.56]
node:[3]  pages:[0217497]  MiB:[00849]  percent[048.02]
[vg-db0038(26913)]
node:[0]  pages:[0130070]  MiB:[00508]  percent[028.65]
node:[1]  pages:[0026870]  MiB:[00104]  percent[005.92]
node:[2]  pages:[0264026]  MiB:[01031]  percent[058.16]
node:[3]  pages:[0033010]  MiB:[00128]  percent[007.27]
[vg-db0037(26948)]
node:[0]  pages:[0078001]  MiB:[00304]  percent[017.10]
node:[1]  pages:[0078063]  MiB:[00304]  percent[017.12]
node:[2]  pages:[0073302]  MiB:[00286]  percent[016.07]
node:[3]  pages:[0226674]  MiB:[00885]  percent[049.70]
[vg-db0036(26986)]
node:[0]  pages:[0189318]  MiB:[00739]  percent[041.84]
node:[1]  pages:[0138542]  MiB:[00541]  percent[030.62]
node:[2]  pages:[0009930]  MiB:[00038]  percent[002.19]
node:[3]  pages:[0114656]  MiB:[00447]  percent[025.34]
[vg-db0035(27029)]
node:[0]  pages:[0035075]  MiB:[00137]  percent[007.73]
node:[1]  pages:[0266316]  MiB:[01040]  percent[058.66]
node:[2]  pages:[0020798]  MiB:[00081]  percent[004.58]
node:[3]  pages:[0131779]  MiB:[00514]  percent[029.03]
```

Memory scattered across nodes for all VMs

```
[vg-db0034(27062)]
node:[0]  pages:[0173804]  MiB:[00678]  percent[038.37]
node:[1]  pages:[0093313]  MiB:[00364]  percent[020.60]
node:[2]  pages:[0030831]  MiB:[00120]  percent[006.81]
node:[3]  pages:[0155011]  MiB:[00605]  percent[034.22]
[vg-db0033(27100)]
node:[0]  pages:[0265909]  MiB:[01038]  percent[058.71]
node:[1]  pages:[0062230]  MiB:[00243]  percent[013.74]
node:[2]  pages:[0044257]  MiB:[00172]  percent[009.77]
node:[3]  pages:[0080547]  MiB:[00314]  percent[017.78]
[vg-db0032(27138)]
node:[0]  pages:[0025163]  MiB:[00098]  percent[005.52]
node:[1]  pages:[0113478]  MiB:[00443]  percent[024.91]
node:[2]  pages:[0127552]  MiB:[00498]  percent[028.00]
node:[3]  pages:[0189330]  MiB:[00739]  percent[041.56]
[vg-db0031(27182)]
node:[0]  pages:[0011550]  MiB:[00045]  percent[002.55]
node:[1]  pages:[0083236]  MiB:[00325]  percent[018.40]
node:[2]  pages:[0100223]  MiB:[00391]  percent[022.15]
node:[3]  pages:[0257437]  MiB:[01005]  percent[056.90]
[vg-db0030(27215)]
node:[0]  pages:[0144517]  MiB:[00564]  percent[031.87]
node:[1]  pages:[0056723]  MiB:[00221]  percent[012.51]
node:[2]  pages:[0080227]  MiB:[00313]  percent[017.69]
node:[3]  pages:[0171986]  MiB:[00671]  percent[037.93]
[vg-db0029(27253)]
node:[0]  pages:[0052847]  MiB:[00206]  percent[011.65]
node:[1]  pages:[0097325]  MiB:[00380]  percent[021.46]
node:[2]  pages:[0051285]  MiB:[00200]  percent[011.31]
node:[3]  pages:[0251995]  MiB:[00984]  percent[055.57]
```

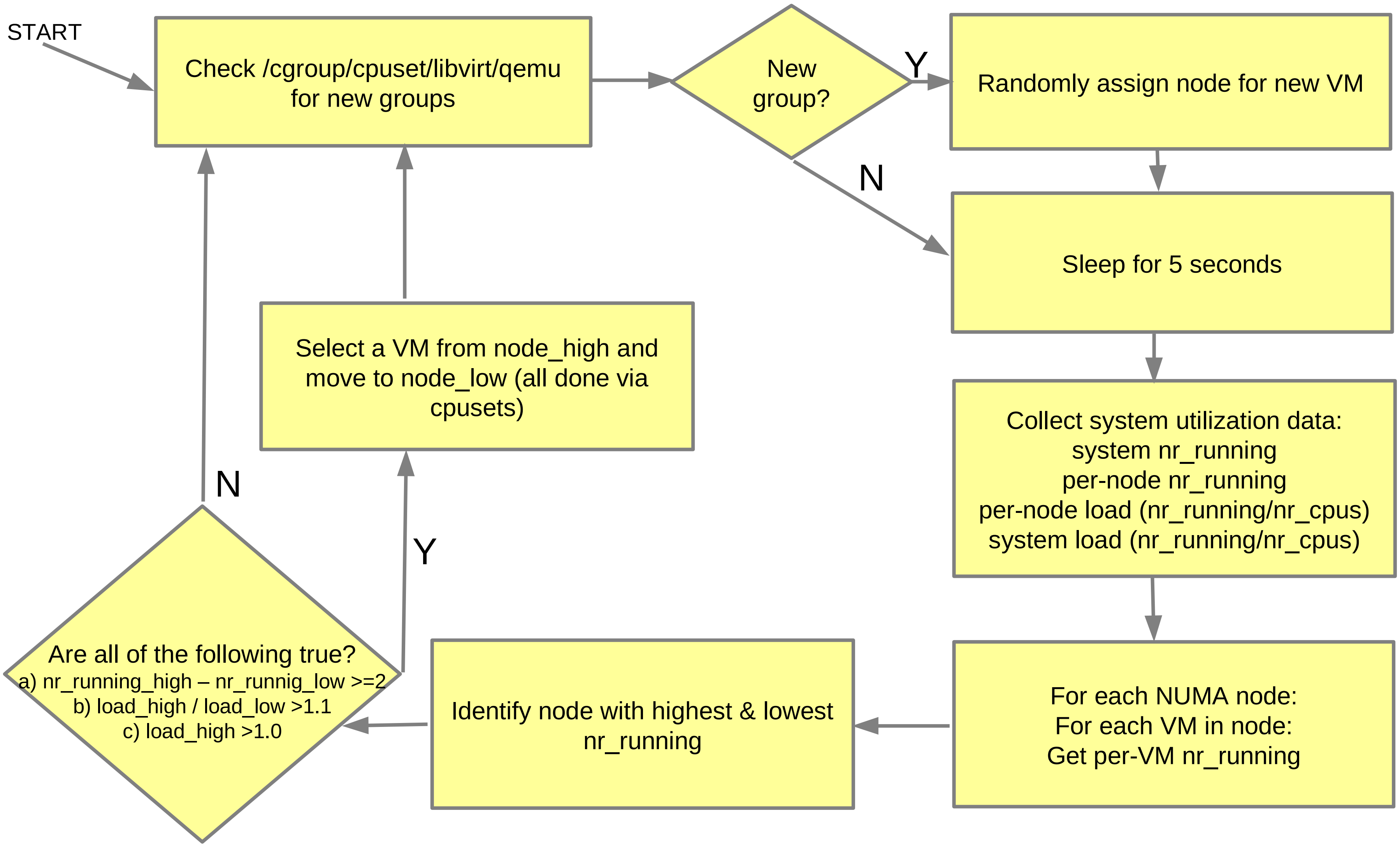
Analysis – What went wrong?

- **Why is memory scattered?**
  - Linux kernel CPU scheduler NUMA policies:
    - Current policies work well for short-lived tasks:
      - Initial placement in least loaded Node
      - Idle CPUs look for tasks to steal
      - Periodic, timer based load balances
      - CPUs can steal tasks from other CPUs, but scope is limited:
        - » Only sibling thread most often
        - » Sibling cores less often
        - » All logical CPUs in system even less often
    - Long lived tasks (like VMs!) do not work well under current policies
      - Load balances with large scopes of CPUs to steal from (whole system) eventually do happen, scattering tasks for a VM across system
      - VM Memory is faulted in the same node where the vCPU is running, so as vCPUs run across the system, memory is also faulted in across the system
      - No policy to keep tasks in a group “close” and no policy to “bulk-move” these tasks to balance the CPU load
      - No influence from current memory placement for tasks

User-space VM balancer

- **Proof-of-Concept: A first attempt at optimizing VM placement to promote node-local CPU-memory communication**
- **Requires cpuset cgroups (works well with libvirt)**
  - Cpuset can migrate cpus *and* memory
- **User-space perl program (vmbalanced) performs the following:**
  - Monitor cgroups, discover new VMs, do initial VM to NUMA node placement
  - Every 5 seconds analyzes CPU load and attempts to re-balance VMs
- **What this does not yet do:**
  - Does not handle really large VMs (ones that would not fit in a single node)
  - Does not currently overcome memory capacity issues
    - Current tests have enough host memory to not make this a problem
    - Trying to keep the first pass at this simple
    - Obviously needs to be addressed

User-space VM balancer



Results with VM Balancer

IBM

- **40 VMs running dbench:**
  - MXH 22919 MB/sec
  - KVM, no balancer: 14541 MB/sec
  - KVM, with balancer: 18771 MB/sec (29% improvement!)
  - KVM, manual binding (10 VMs per node) 18896 MB/sec
    - About the same throughput as balancer and the best we could expect for balancer
  - This test is actually not that challenging
    - Initial placement gets it mostly right
    - Only a few VM migrations necessary during dbench run
    - Regardless, a simple algorithm can make a dramatic difference
  - Perf stats:
    - Off-node memory accesses (lower is better):
      - No balancer: 217.6 M/sec
      - Balancer: 6.2 M/sec
      - Manual Binding: 0.9 M/sec
    - Instructions per cycle (higher is better)
      - No Balancer: 0.293
      - Balancer: 0.374
      - Manual Binding: 0.374

Results with VM Balancer (balancer output)

```
[Mon Aug  8 22:12:46 CDT 2011]
node     nr_running     nr_cpus   load       imbalance   VMs(nr_running)
node0    37             20        1.850000   -008.64 vg-db0030: 4 vg-db0038: 4 vg-db0016: 4 vg-db0002: 4 vg-db0026: 4 vg-db0037: 4 vg-db0023: 4 vg-db0028: 4 vg-db0010: 4
node1    44             20        2.200000   0008.64 vg-db0018: 5 vg-db0003: 4 vg-db0036: 4 vg-db0007: 4 vg-db0004: 5 vg-db0014: 4 vg-db0027: 3 vg-db0011: 5 vg-db0012: 4 vg-db0021: 4 vg-db0035: 4
node2    52             20        2.600000   0028.40 vg-db0020: 4 vg-db0006: 4 vg-db0032: 5 vg-db0017: 4 vg-db0001: 4 vg-db0034: 4 vg-db0024: 5 vg-db0019: 4 vg-db0031: 4 vg-db0040: 4 vg-db0015: 4 vg
5 vg-db0008: 5
node3    29             20        1.450000   -028.40 vg-db0022: 4 vg-db0039: 4 vg-db0025: 4 vg-db0013: 5 vg-db0009: 4 vg-db0033: 5 vg-db0029: 4
all      162            80        2.025000   0000.00

The nr_running_high[52], nr_running_low[29], nr_running_diff[23], load_high[2.600000], load_low[1.450000], load_ratio[1.792980]
moving [vg-db0020] from node [node2] to node [node3]
VM migration elapsed time: 6.905896

[Mon Aug  8 22:12:56 CDT 2011]
node     nr_running     nr_cpus   load       imbalance   VMs(nr_running)
node0    37             20        1.850000   -008.64 vg-db0030: 4 vg-db0038: 5 vg-db0016: 4 vg-db0002: 4 vg-db0026: 4 vg-db0037: 4 vg-db0023: 5 vg-db0028: 4 vg-db0010: 4
node1    43             20        2.150000   0006.17 vg-db0018: 4 vg-db0003: 5 vg-db0036: 4 vg-db0007: 4 vg-db0004: 4 vg-db0014: 4 vg-db0027: 4 vg-db0011: 4 vg-db0012: 4 vg-db0021: 5 vg-db0035: 4
node2    50             20        2.500000   0023.46 vg-db0006: 4 vg-db0032: 4 vg-db0017: 4 vg-db0001: 4 vg-db0034: 4 vg-db0024: 4 vg-db0019: 4 vg-db0031: 4 vg-db0040: 4 vg-db0015: 4 vg-db0005: 5 vg
4
node3    32             20        1.600000   -020.99 vg-db0022: 4 vg-db0039: 4 vg-db0025: 5 vg-db0020: 4 vg-db0013: 4 vg-db0033: 4 vg-db0029: 4 vg-db0009: 5
all      162            80        2.025000   0000.00

The nr_running_high[50], nr_running_low[32], nr_running_diff[18], load_high[2.500000], load_low[1.600000], load_ratio[1.562402]
moving [vg-db0006] from node [node2] to node [node3]
VM migration elapsed time: 4.228818

[Mon Aug  8 22:13:04 CDT 2011]
node     nr_running     nr_cpus   load       imbalance   VMs(nr_running)
node0    36             20        1.800000   -011.11 vg-db0030: 4 vg-db0038: 5 vg-db0016: 4 vg-db0002: 4 vg-db0026: 5 vg-db0037: 4 vg-db0023: 4 vg-db0028: 5 vg-db0010: 4
node1    44             20        2.200000   0008.64 vg-db0018: 4 vg-db0003: 4 vg-db0036: 4 vg-db0007: 5 vg-db0004: 4 vg-db0014: 4 vg-db0027: 4 vg-db0011: 4 vg-db0012: 4 vg-db0021: 4 vg-db0035: 5
node2    44             20        2.200000   0008.64 vg-db0032: 4 vg-db0017: 4 vg-db0001: 4 vg-db0034: 4 vg-db0024: 4 vg-db0019: 4 vg-db0031: 4 vg-db0040: 4 vg-db0015: 4 vg-db0005: 4 vg-db0008: 4
node3    38             20        1.900000   -006.17 vg-db0022: 5 vg-db0039: 5 vg-db0025: 5 vg-db0020: 4 vg-db0006: 4 vg-db0013: 4 vg-db0033: 4 vg-db0029: 4 vg-db0009: 4
all      162            80        2.025000   0000.00

The nr_running_high[44], nr_running_low[36], nr_running_diff[8], load_high[2.200000], load_low[1.800000], load_ratio[1.222154]
moving [vg-db0018] from node [node1] to node [node0]
VM migration elapsed time: 4.913064

[Mon Aug  8 22:13:11 CDT 2011]
node     nr_running     nr_cpus   load       imbalance   VMs(nr_running)
node0    41             20        2.050000   -004.65 vg-db0018: 4 vg-db0030: 4 vg-db0038: 4 vg-db0016: 5 vg-db0002: 4 vg-db0026: 4 vg-db0037: 4 vg-db0023: 4 vg-db0028: 5 vg-db0010: 4
node1    45             20        2.250000   0004.65 vg-db0003: 4 vg-db0036: 4 vg-db0007: 4 vg-db0004: 4 vg-db0014: 4 vg-db0027: 5 vg-db0011: 4 vg-db0012: 4 vg-db0021: 4 vg-db0035: 4
node2    46             20        2.300000   0006.98 vg-db0032: 4 vg-db0017: 4 vg-db0001: 4 vg-db0034: 4 vg-db0024: 5 vg-db0019: 4 vg-db0031: 4 vg-db0040: 4 vg-db0015: 4 vg-db0005: 4 vg-db0008: 4
node3    40             20        2.000000   -006.98 vg-db0022: 4 vg-db0039: 4 vg-db0025: 4 vg-db0020: 4 vg-db0006: 4 vg-db0013: 4 vg-db0033: 4 vg-db0029: 4 vg-db0009: 4
all      172            80        2.150000   0000.00

The nr_running_high[46], nr_running_low[40], nr_running_diff[6], load_high[2.300000], load_low[2.000000], load_ratio[1.149943]
moving [vg-db0032] from node [node2] to node [node3]
VM migration elapsed time: 5.302738

[Mon Aug  8 22:13:20 CDT 2011]
node     nr_running     nr_cpus   load       imbalance   VMs(nr_running)
node0    41             20        2.050000   -001.20 vg-db0018: 4 vg-db0030: 4 vg-db0038: 4 vg-db0016: 4 vg-db0002: 4 vg-db0026: 4 vg-db0037: 5 vg-db0023: 4 vg-db0028: 4 vg-db0010: 4
node1    41             20        2.050000   -001.20 vg-db0003: 4 vg-db0036: 4 vg-db0007: 4 vg-db0004: 4 vg-db0014: 4 vg-db0027: 4 vg-db0011: 4 vg-db0012: 4 vg-db0021: 4 vg-db0035: 4
node2    43             20        2.150000   0003.61 vg-db0017: 4 vg-db0001: 5 vg-db0024: 4 vg-db0034: 4 vg-db0019: 4 vg-db0031: 4 vg-db0040: 4 vg-db0015: 4 vg-db0005: 4 vg-db0008: 4
node3    41             20        2.050000   -001.20 vg-db0022: 4 vg-db0039: 4 vg-db0025: 4 vg-db0020: 4 vg-db0006: 4 vg-db0032: 4 vg-db0013: 5 vg-db0033: 4 vg-db0029: 4 vg-db0009: 4
all      166            80        2.075000   0000.00

The nr_running_high[43], nr_running_low[41], nr_running_diff[2], load_high[2.150000], load_low[2.050000], load_ratio[1.048729]
```

When 40 VMs start their workloads there is some load imbalance

After a few iterations the VMs are balanced

Results with VM Balancer

- **Let's try something more challenging**
- **Use 20 of the 40 VMs: select 20 VMs from just the first 2 NUMA nodes**
  - Immediately following the 40 VM test
- **At the beginning of the test, 20 VMs will saturate the CPU from first 2 nodes**
- **To get the best throughput, ½ of these VMs will need to be migrated**
  - MXH 19164 MB/sec
  - KVM, no balancer: 15298 MB/sec
  - KVM, with balancer: 19374 MB/sec
    - Slightly better than MXH!
  - KVM, manual binding (10 VMs per node) 9096 MB/sec
    - Good example of why manual binding has limited use (VMs are stuck on first two nodes)
  - Perf stats:
    - Off-node memory references (lower is better):
      - No balancer: 212.2 M/sec
      - Balancer: 5.8 M/sec
      - Manual Binding: 0.7 M/sec
    - Instructions per cycle (higher is better)
      - No Balancer: 0.307
      - Balancer: 0.395
      - Manual Binding: 0.346

Results with VM Balancer (Summary)

- **40 VM test**
  - Out of the box performance improved by 29%
  - NUMA optimization relatively easy, as initial placement does most of the work
  - Relatively few balance operations needed to get even balance
  - Can achieve same throughput as manual binding
  - Still need another 29% to get parity with MXH
    - CPU is over-committed
      - vCPU run time can affect cache warmth, probably worth investigating
      - Lock-holder preemption might be occurring
- **20 VM test**
  - Out-of-the box performance improved by 26%
  - Performance parity with MXH
  -

More work to do

- **Re-balance to correct memory imbalance**
  - Probably not too hard if there is not a CPU constraint
  - Much harder when you are trying to fix memory and CPU imbalance
  - Instead of simply moving a single VM one at a time, may require swapping (1 for 1, 1 for 2 or 3) VMs across nodes to get good balance
- **Re-balance to optimize KSM for NUMA locality**
  - If a set of VMs have a lot of shared pages, ideally they should be on the same node
- **VM migration probation period (to correct a CPU imbalance)**
  - If you are concerned the need for CPU is temporary, don't waste a lot of cycles moving VM memory around
  - Move CPUs first, confirm this was not a very short term need, then move VM memory. If the need for CPU goes away, then revert the CPU move.
  - Or, just always lazily move memory (but not easy to implement)
- **When moving VMs pick a VM which has lowest resident memory/CPU-usage**
  - Moving memory is costly, get the best bang/buck by picking VMs that are “easy” to move
- **Handle really big VMs**
  - Big VMs can require CPU and memory from more than one node
  - Create multi-Node VMs, with CPU *and memory* per VM-node
  - Treat each VM-node as a small VM in the host, move VM-nodes independently (not really compatible with CPU sets, need to migrate individual memory mappings)

Kernel or User?

- **Should this work move to kernel scheduler?**
  - Pros
    - More control – scheduler can generally react to changes much faster
    - Opportunity to do with other things like gang scheduling, entitlement guarantees, latency guarantees for virtualization
    -
  - Cons
    - You have to actually get it included in scheduler code
    - Much higher risk and probably requires a lot more testing
      - Could lower the speed at which changes could be made and delivered to users

Questions?

- **/proc/stat provide nr_running per CPU**
  - necessary for user space VM balancer
  - cpu_load also made available, but not used at this time

```
diff -Naurp linux-2.6.39/fs/proc/stat.c linux-2.6.39b/fs/proc/stat.c
--- linux-2.6.39/fs/proc/stat.c 2011-05-18 23:06:34.000000000 -0500
+++ linux-2.6.39b/fs/proc/stat.c        2011-07-20 13:51:45.376004463 -0500
@@ -91,7 +91,7 @@ static int show_stat(struct seq_file *p,
                guest_nice = kstat_cpu(i).cpustat.guest_nice;
                seq_printf(p,
                        "cpu%d %llu %llu %llu %llu %llu %llu %llu %llu %llu "
-                       "%llu\n",
+                        "%llu %lu %lu\n",
                        i,
                        (unsigned long long)cputime64_to_clock_t(user),
                        (unsigned long long)cputime64_to_clock_t(nice),
@@ -102,7 +102,9 @@ static int show_stat(struct seq_file *p,
                        (unsigned long long)cputime64_to_clock_t(softirq),
                        (unsigned long long)cputime64_to_clock_t(steal),
                        (unsigned long long)cputime64_to_clock_t(guest),
-                       (unsigned long long)cputime64_to_clock_t(guest_nice));
+                        (unsigned long long)cputime64_to_clock_t(guest_nice),
+                        nr_running_cpu(i),
+                        cpu_load(i));
        }
        seq_printf(p, "intr %llu", (unsigned long long)sum);

diff -Naurp linux-2.6.39/include/linux/sched.h linux-2.6.39b/include/linux/sched.h
--- linux-2.6.39/include/linux/sched.h  2011-05-18 23:06:34.000000000 -0500
+++ linux-2.6.39b/include/linux/sched.h 2011-07-20 13:50:27.096004478 -0500
@@ -137,9 +137,11 @@ extern int nr_threads;
 DECLARE_PER_CPU(unsigned long, process_counts);
 extern int nr_processes(void);
 extern unsigned long nr_running(void);
+extern unsigned long nr_running_cpu(unsigned long cpu);
 extern unsigned long nr_uninterruptible(void);
 extern unsigned long nr_iowait(void);
 extern unsigned long nr_iowait_cpu(int cpu);
+extern unsigned long cpu_load(int cpu);
 extern unsigned long this_cpu_load(void);
```

```
diff -Naurp linux-2.6.39/kernel/sched.c linux-2.6.39b/kernel/sched.c
--- linux-2.6.39/kernel/sched.c 2011-05-18 23:06:34.000000000 -0500
+++ linux-2.6.39b/kernel/sched.c        2011-07-20 13:50:14.746004482 -0500
@@ -3017,6 +3017,11 @@ unsigned long nr_running(void)
        return sum;
 }

+unsigned long nr_running_cpu(unsigned long cpu)
+{
+        return cpu_rq(cpu)->nr_running;
+}
+
 unsigned long nr_uninterruptible(void)
 {
        unsigned long i, sum = 0;
@@ -3061,6 +3066,12 @@ unsigned long nr_iowait_cpu(int cpu)
        return atomic_read(&this->nr_iowait);
 }

+unsigned long cpu_load(int cpu)
+{
+        struct rq *this = cpu_rq(cpu);
+        return this->cpu_load[0];
+}
+
 unsigned long this_cpu_load(void)
 {
        struct rq *this = this_rq();
```

- **CPU utilization of 20 VM test (20 VMs initially on just first 2 NUMA nodes)**
  - First minute indicates VMs moved to 2 unused NUMA nodes and eventually using CPU from all nodes
  - After first minute, a couple periods of lower CPU might indicate incorrect balances