Improving the Out-of-Box KVM Performance
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!?!)


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?

Memory scattered across nodes for all VMs
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

Check /cgroup/cpuset/libvirt/qemu for new groups

New group?

Randomly assign node for new VM

Sleep for 5 seconds

Collect system utilization data:
- System nr_running
- Per-node nr_running
- Per-node load (nr_running/nr_cpus)
- System load (nr_running/nr_cpus)

For each NUMA node:
For each VM in node:
Get per-VM nr_running

Identify node with highest & lowest nr_running

Are all of the following true?
- a) nr_running_high – nr_running_low >= 2
- b) load_high / load_low > 1.1
- c) load_high > 1.0

Select a VM from node_high and move to node_low (all done via cpuset)
Results with VM Balancer

- **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]

<table>
<thead>
<tr>
<th>node</th>
<th>nr_running</th>
<th>nr_cpus</th>
<th>load</th>
<th>imbalance</th>
<th>VMs(nr_running)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>37</td>
<td>20</td>
<td>1.85000</td>
<td>-0.0864</td>
<td>vg-db0030: 4</td>
</tr>
<tr>
<td>node1</td>
<td>44</td>
<td>20</td>
<td>2.20000</td>
<td>0.0084</td>
<td>vg-db0018: 5</td>
</tr>
<tr>
<td>node2</td>
<td>52</td>
<td>20</td>
<td>2.60000</td>
<td>0.0284</td>
<td>vg-db0002: 4</td>
</tr>
</tbody>
</table>

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]

<table>
<thead>
<tr>
<th>node</th>
<th>nr_running</th>
<th>nr_cpus</th>
<th>load</th>
<th>imbalance</th>
<th>VMs(nr_running)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>37</td>
<td>20</td>
<td>1.85000</td>
<td>-0.0864</td>
<td>vg-db0030: 5</td>
</tr>
<tr>
<td>node1</td>
<td>44</td>
<td>20</td>
<td>2.15000</td>
<td>0.0061</td>
<td>vg-db0003: 5</td>
</tr>
<tr>
<td>node2</td>
<td>50</td>
<td>20</td>
<td>2.50000</td>
<td>0.0234</td>
<td>vg-db0006: 4</td>
</tr>
</tbody>
</table>

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


<table>
<thead>
<tr>
<th>node</th>
<th>nr_running</th>
<th>nr_cpus</th>
<th>load</th>
<th>imbalance</th>
<th>VMs(nr_running)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>36</td>
<td>20</td>
<td>1.80000</td>
<td>-0.1111</td>
<td>vg-db0030: 5</td>
</tr>
<tr>
<td>node1</td>
<td>44</td>
<td>20</td>
<td>2.20000</td>
<td>0.0084</td>
<td>vg-db0003: 5</td>
</tr>
<tr>
<td>node2</td>
<td>44</td>
<td>20</td>
<td>2.20000</td>
<td>0.0084</td>
<td>vg-db0003: 5</td>
</tr>
</tbody>
</table>

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


<table>
<thead>
<tr>
<th>node</th>
<th>nr_running</th>
<th>nr_cpus</th>
<th>load</th>
<th>imbalance</th>
<th>VMs(nr_running)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>41</td>
<td>20</td>
<td>2.05000</td>
<td>-0.0065</td>
<td>vg-db0018: 4</td>
</tr>
<tr>
<td>node1</td>
<td>45</td>
<td>20</td>
<td>2.25000</td>
<td>0.0046</td>
<td>vg-db0003: 4</td>
</tr>
<tr>
<td>node2</td>
<td>46</td>
<td>20</td>
<td>2.30000</td>
<td>0.0009</td>
<td>vg-db0032: 4</td>
</tr>
</tbody>
</table>

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: 5.302378


<table>
<thead>
<tr>
<th>node</th>
<th>nr_running</th>
<th>nr_cpus</th>
<th>load</th>
<th>imbalance</th>
<th>VMs(nr_running)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node0</td>
<td>41</td>
<td>20</td>
<td>2.05000</td>
<td>-0.0120</td>
<td>vg-db0018: 4</td>
</tr>
<tr>
<td>node1</td>
<td>41</td>
<td>20</td>
<td>2.05000</td>
<td>-0.0120</td>
<td>vg-db0003: 4</td>
</tr>
<tr>
<td>node2</td>
<td>43</td>
<td>20</td>
<td>2.15000</td>
<td>0.0003</td>
<td>vg-db0001: 5</td>
</tr>
</tbody>
</table>

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]
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)
**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

```c
diff -Naurp linux-2.6.39/fs/proc/stat.c linux-2.6.39b/fs/proc/stat.c
@@ -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 %lu %lu
",
             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(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/kernel/sched.c linux-2.6.39b/kernel/sched.c
@@ -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();
```
Backup Slides

- **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