Utilization-based Scheduling in OpenStack* Compute (Nova)

Tech Tip

September 2015

Authors: Lianhao Lu, Yingxin Cheng

Reviewers: Malini Bhandaru, Will Auld

Configurations:

Intel® Family 15 Model 6 Stepping 8 Genuine Intel® CPU (3.2GHz, 12T/2C, 8M cache)
Intel® Family 15 Model 4 Stepping 8 Genuine Intel® CPU (up to 2.66GHz, 16T/2C, 1M cache)

Contents

Figures

Revision History

Revision	Date	Description
001	June 3, 2015	Initial public release.
002	Sept 9, 2015	Update because of Liberty release change.

§

1 Introduction

OpenStack* is an open-source cloud computing software platform that has gained an exponential increase in developer interest and adoption. More than 200 companies, including Red Hat, HP, Intel, IBM, and VMware, are contributing to the project. With a rapid six-month release cycle from A (Austin) to J (Juno), OpenStack has proven to be deployable on more than 500 nodes running about 168,000 virtual machines a day on a single cloud instance.[1]

OpenStack aims to benefit businesses by providing infrastructure as a service (IaaS) consumed either as a public, private or hybrid cloud. Although OpenStack is open-source and free to the public, to minimize total cost of ownership, it is necessary to contain hardware expense by managing resources efficiently even as we seek to meet user service level requirements.

Two approaches have been considered to achieve better resource utilization, namely resource over-commit and scheduling improvements.

OpenStack allows over-committing CPU and RAM on compute nodes by providing more virtual cores and memory than the physical host capacity. Physical cores and memory are loaded with more guests, up to the over-commit limits. Over-commitment provides a substantial benefit, allowing compute nodes to serve more users; it taps the potential of each host given that virtual machines in typical workloads do not simultaneously peg the resources.

To achieve better resource utilization through scheduling, OpenStack allows the administrator to configure scheduling behavior based on resource usage. Prior to the current implementation, only RAM resource utilization was considered towards this end resulting in less than optimal utilization and performance. We explore the state of the art and the improvements implemented to allow for richer control towards obtaining improved performance in conjunction with better utilization.

[1] *http://www.enterprisetech.com/2014/05/16/firing-75k-vms-openstack-afternoon/*

2 Current OpenStack scheduling

Virtual machines consume most of the resources on a host or compute node, and it is difficult and costly to migrate a VM from one host to another after it is booted up. To ensure good performance and prevent resource wastage either from excessive occupation or through idling, OpenStack must predict a VM's consumption and choose a suitable host machine before it is booted. The **Nova-scheduler module** is responsible for scheduling decisions and takes into consideration the full fleet of available hosts which may be characterized by different resource capacities and features.

The Nova-scheduler supports multiple scheduling strategies. For additional information see *OpenStack filter scheduler and weighers documents*[2]. The default and most useful one is the **FilterScheduler**. When a VM launch request is received in phase one, filters are applied to determine whether a host has adequate free capacity to meet the requested resource parameters and/or any other requirements such a host capability/property. Filtering is a binary operation, culling all hosts that meet the VM needs. In the second phase, the FilterScheduler applies ***weighers*** to score all the qualifying hosts to choose the best among them to launch the given VM request.

More than 20 filters are currently available. They can be combined to meet complex requirements. They are categorized as follows:

- **Resource-based filters**: The decision is made according to available resources, including memory, disk, CPU cores, and so on. For example: CoreFilter, DiskFilter and RamFilter.
- **Image-based filters**: Filter hosts according to image properties. The properties can be set to select hosts from CPU architecture, hypervisor, and VM mode.
- **Host-based filters**: Select hosts based on grouping criteria such as location, availability zone, or designated use (host aggregate). Plenty of choices are available.
- **Net-based filters**: The SimpleCIDRAffinityFilter chooses hosts based on their IP or subnet.
- **Custom filters**: Allow users to build up a custom filter. JsonFilter leverages JSON for customization.
- **Others**: Such as AllHostsFilter, ComputeFilter, and RetryFilter.

The scheduler weighs the available hosts after filtering, sorts by weight, and selects the best rated host. Multiple weighting functions may be used. The default is to use ***'all_weighers'***, which selects all the available weighting functions, with the **RamWeigher** playing a significant role. The RamWeigher scores a host higher based on available memory. The more available memory a host has, the higher is its score. Thus the default weigher always balances the memory usage of hosts.

Compared to the number of available filters, there are relatively few weighting functions.

[2] *http://docs.openstack.org/developer/nova/devref/filter_scheduler.html*

3 The problem and solution

The FilterScheduler is an overall solution to allocate resources, and the multiple available filters give a comprehensive set of choices. However, filters only generate a server list that is ready to use; they address static reservation requests such as memory and virtual CPUs (vCPUs). They do not provide any strategies to maximize performance. It's the weighers' responsibility to balance workloads and to make the best use of the available resources. The weighers need to be intelligent enough to predict a better output.

The current strategy of only weighing against memory usage, by way of the RamWeigher, is limited in effectiveness. This is because each VM has its own separate memory space, and random-access memory performance is not largely impacted simply by memory consumption. Further, the OpenStack OS base, namely Linux*, always harnesses unused memory for caching and performance improvement. In contrast, if a VM is launched on a host with heavy CPU utilization, the VM performs poorly. On a compute heavy host, contention for CPU time slices results in the VMs on the host enduring a performance penalty.

The key problem is that the current weighting strategy is weak and leads to inefficient usage of resources. Weighers should measure more than RAM usage. VM performance is largely affected by the host's computation ability and its usage. Those factors can be CPU utilization rate, vCPU usage, and processor characteristics including frequency and model. It is better to dispatch a VM to an idle host with powerful CPUs and less memory. In particular, if a VM requires more cores and is compute intensive, more attention should be paid to a host CPU utilization than its available memory to ensure better performance.

It is important to gather and combine both static properties and dynamic usage statistics to generate the weight. The goal is to execute all the VMs at full speed and to ensure a better level of service. To generate an even better solution, this must be expanded to consider network bandwidth and power usage.

The **UBS (utilization-based scheduling) framework** has been developed to solve this problem. It measures various factors called metrics, and uses these in the weighers invoked in the FilterScheduler to determine the best launch-host. As a pluggable framework, users can easily implement their own resource monitors to collect dynamic statistics from hosts. FilterScheduler then uses these metrics to schedule VMs. The UBS framework consists of the following three parts:

- **Resource monitor**: This module resides in Nova-compute. The monitors are called periodically by the Nova-compute service to generate statistics called metrics, updating the metrics to a Nova database. The metrics can then be used for scheduling. Resource monitor is also extendable. Users can write their own monitors to collect custom metrics.
- **Metric scheduler filter**: For those hosts that do not have specific metrics, the metric filter is available to exclude them. This filter prevents exceptions from being raised from the lack of availability of a metric in the weighing step, the unavailable-metric issue.
- **Metric scheduler weigher**: The Metric weigher weights the metrics based on user configurations. It generates a score for each of the qualifying hosts based on the provided formula and weight values to choose the best host to launch the VM.

4 Using the UBS framework

The UBS framework is designed to be configurable and easy to use. The configuration file 'nova.conf' usually resides in '/etc/nova/'. Configuration consists of three parts available in three modules in UBS:

- **Resource monitor**: Configure 'compute_monitors' in the default section of nova.conf.
- **Metric scheduler filter**: Add MetricFilter to 'scheduler_default_filters' to apply metric filtering by the Nova-scheduler service.
- **Metric scheduler weigher**: Configure 'scheduler_weight_classes' to activate metric weigher. Then configure the 'weight_setting' with a comma-separated list, including metric names, values and hosts.

The following shows a simple example of UBS framework configuration in 'nova.conf'. The resource monitor is configured to ComputeDriverCPUMonitor, collecting metrics including cpu.percent periodically. The weigher is set to MetricsWeigher, and it orders the candidate hosts by CPU usage under the configuration 'weight_setting = cpu.percent=-1.0' in the metrics section of 'nova.conf'. The metric 'cpu.percent' comes from ComputeDriverCPUMonitor, which records the latest host CPU payload in percentage.

```
[DEFAULT]
# configure with your monitors entry point name
compute_monitors=virt_driver
# configure weigher here
scheduler_weight_classes=nova.scheduler.weights.metrics.MetricsWeigher
# the host is chosen randomly from best '1' host
scheduler_host_subset_size = 1
[METRICS]
# determine how metrics are weighed: score = -1.0 * cpu.percent
# the list format is weight_setting = name1=1.0, name2=-1.0
weight_setting = cpu.percent=-1.0
```

Before nova version 12.0.0.0b2 in Liberty release, the configuration is a little different. They are shown in boldface.

```
[DEFAULT]
# configure with your monitors class name
compute_monitors=ComputeDriverCPUMonitor
scheduler_weight_classes=nova.scheduler.weights.metrics.MetricsWeigher
scheduler_host_subset_size = 1
[METRICS]
weight_setting = cpu.percent=-1.0
```

5 Experimental results

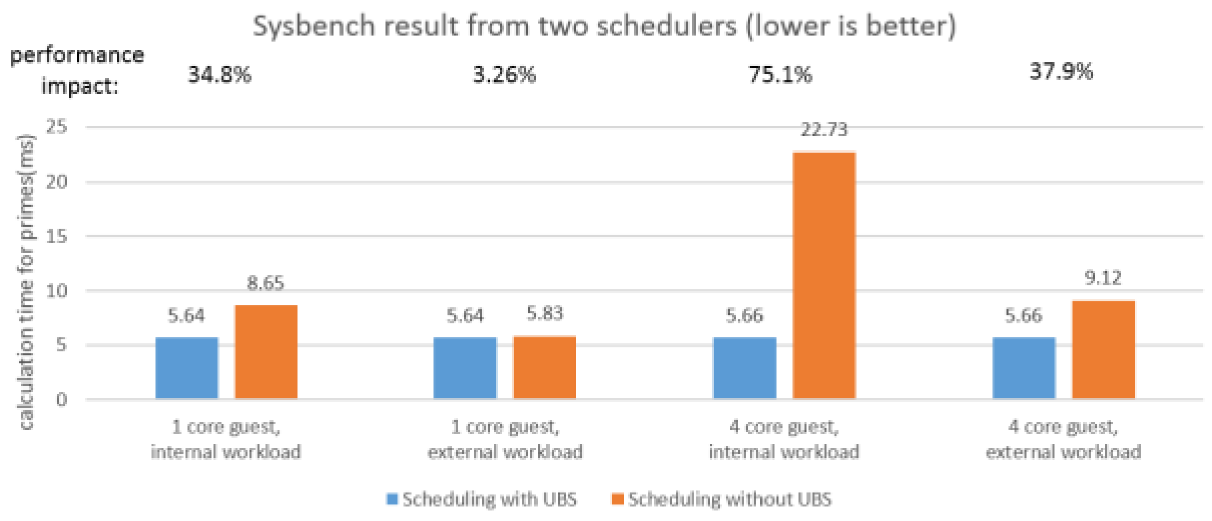
To demonstrate that factors other than RAM surplus can affect VM performance, we evaluated some benchmarks with and without UBS scheduling.

The underlying hardware and CPU infrastructure in the experiment are identical among the compute nodes; both simplify the discussion and allow us to ignore platform differences in the weighting function.

In the experiment, the default weigher without UBS (RamWeigher) dispatches a VM to a host with more free memory, while the UBS metric scheduler weigher dispatches the same VM to the least busy host.

We used two benchmarks, *SPEC2006*[3] and *Sysbench*[4] as the VM load. After scheduling, VM performance is measured by the benchmark score obtained. While the experiments are limited in scope, the results show that a VM dispatched by the UBS metric weigher can provide benchmark scores equal to and in some cases up to four times better than without UBS. Once we add more dynamic metrics, breaking out Sysbench into IO, compute and transaction type benchmark tests (see *https://www.howtoforge.com/how-to-benchmark-your-system-cpu-file-io-mysql-with-sysbench*) will help to illustrate better scheduling given different workload characteristics.

Figure 5-1 How scheduler strategy affects performance



[3] *https://www.spec.org/cpu2006/*
[4] *https://launchpad.net/sysbench*

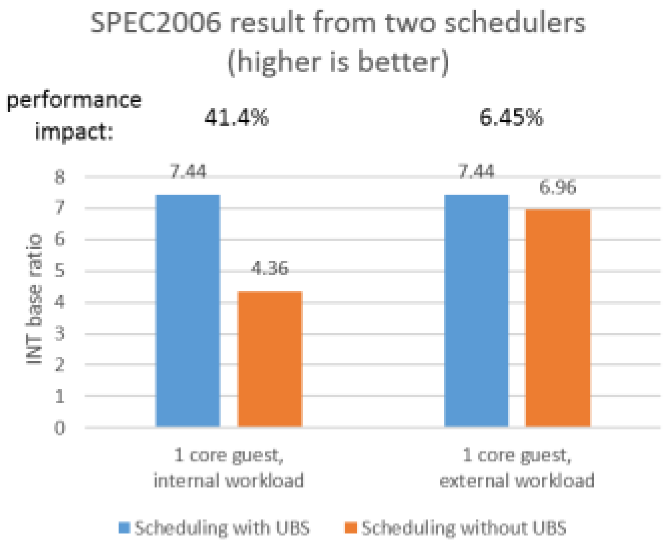
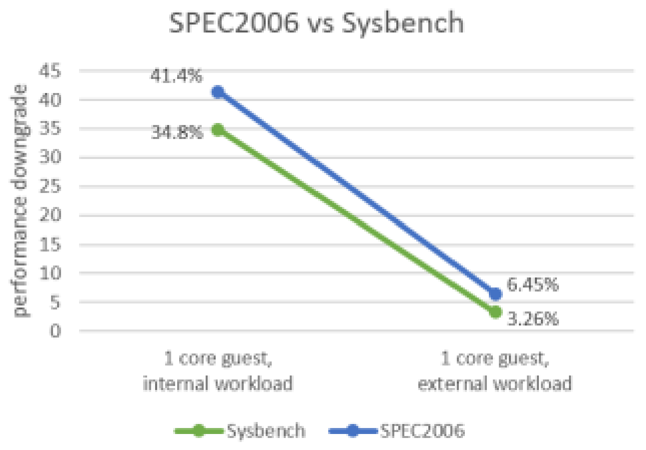


Figure 5-1 shows the experimental results from two well-known benchmarks. The benchmark software measures the performance inside the scheduled VM to compare the two scheduling strategies. There are two compute nodes. The first one has more free memory but with heavy workload, and the second node has only enough free memory for the VM but with low workload. The default scheduler without UBS using RamWeigher prefers the first node, while the scheduler with UBS configurations chooses the second node that is idle. There are two types of workload. In the internal workload configuration, CPU-intensive applications are running on ten other VMs on the same host, and in the external workload configuration, the same CPU intensive applications are running parallel in ten threads on the host OS. The result indicates that different types of workload have different effects on the scheduled VM. The experimental results show the scheduled VM has better performance if the UBS framework is applied.

The authors were aware of behavior differences between external and internal workloads on a host. See *http://people.seas.harvard.edu/~apw/stress* or even a tight C or Python 'while loop' code. This may stem from Linux differences in scheduling strategy for CPU time slices between applications and the hypervisor, versus how scheduling is handled within the hypervisor for VMs. Another possibility is page table loading related overhead. This needs further investigation. Typically an OpenStack compute host will carry only VM workloads with a few external workloads such as agents for telemetry like Ceilometer, remote attestation, etc.

Figure 5-2 goes further by comparing SPEC2006 and Sysbench results in the same experiment. The scheduled guest has one core, and the workload comes from either external or internal. The two slopes in the figure show that different types of workload cause similar performance downgrade variations as measured by the two benchmark tools. The results suggest that the statistics from chosen benchmark tools are reasonable.

Figure 5-2 Benchmark accuracy



This experiment indicates that the UBS scheduling strategy can manage resources much more effectively, especially on compute site. Considering that the CPU utilization rate is dynamic and difficult to predict, the current scheduler with UBS configurations assumes this rate is almost steady. Auto scaling technology is a solution to adjust the resource at runtime. Furthermore, OpenStack can be deployed on different platforms with various computation abilities. For instance, clouds could be deployed on heterogeneous hardware on new servers and legacy hardware. So in order to calculate the accurate weight value and perform better scheduling, benchmarking the target platform is useful to determine baseline weights. Those benchmark values can be pre-calculated according to CPU model of hardware. See *Comparing cpu utilization across different cpu capacity*[5].

[5] *http://www-01.ibm.com/support/knowledgecenter/SS8MU9_2.2.0/Admin/concepts/resourceoptimizationpolicy.dita*

6 Extending the UBS framework

To meet real-world diverse requirements, the UBS framework should be flexible enough to add user-defined metrics. Fortunately, the steps are simple: First, write a new monitor plugin in Nova-compute to collect custom metrics; and second, configure metric weigher to use the new metrics reported by the monitor plugin.

The new monitor class called 'SpamMonitor' acquiring the metric 'ham.eggs' can be created in the following five steps:

1. Inherit MonitorBase class, which provides a prototype of all the resource monitors:

```
from nova.compute import monitors
class SpamMonitor(monitors.MonitorBase):
    #implementations here
```

2. Implement the constructor to build the class instance properly.

3. Implement the '_update_data' method to acquire the metric value:

```
def _update_data(self):
    self._data["ham.eggs"] = ham_eggs_fetching_method()
```

4. Implement 'get_metric_names' method to return available metric names:

```
def get_metric_names(self):
    return set("ham.eggs")
```

5. Implement 'get_metric' method to return metric value according to its name:

```
def get_metric(self, name):
    return self._data[name]
```

6. Apply the modified code to all computer nodes and reboot the services.

   When multiple compute hosts are involved, as with any traditional upgrade task, all of them must be updated with the code extensions.

7 UBS framework roadmap

The UBS framework supports more than filtering and weighing computational resources. It allows the specification and collection of metrics the cloud operator deems useful in realizing a better scheduling decision. While we discussed a metric for CPU usage on a host, other metrics that would add value are network bandwidth and power usage. A next step would be to incorporate/upstream collection of these additional metrics and a weighting function that considers them. Cloud deployers can go ahead given the extensible UBS framework.

Cloud instances may have different characteristics based on their workload mix and the service levels they must support. The weighting function would need to be tweaked to realize this. A user interface to make this task easier would be beneficial.

Last but not least, each workload has different needs, such as a compute bound workload versus one that might process network packets or do a lot of file read/write operations. The metric weights would be workload-specific to meet service levels required and ensure optimal cloud resource utilization. As a first step, providing scheduling hints such as compute, networking and IO bound would help to select different weights. A more fluid solution would be monitoring the workload and perhaps moving it based on its behavior pattern, with some hysteresis to prevent thrashing. For long-running tasks, learning its behavior pattern would aid in making better scheduling.

To monitor the progression of UBS framework in the OpenStack community, see the *Utilization Aware Scheduling*[6] blueprint.

[6] https://blueprints.launchpad.net/nova/+spec/utilization-aware-scheduling