Using kube-burner to measure VM deployment at scale

Introduction

Those of us that work in the area of performance analysis know there are many ways to measure a workload and standardizing on a tool can help provide more comparable results across different configurations and environments. The Red Hat Performance & Scale team maintains a workload tool called kube-burner (which has been accepted as a CNCF sandbox project) that can be used to test deployments at scale. While you can learn more about all the ways kube-burner can be used for scalability testing, including how the tool is extended for egress coverage, this guide will focus on customizing a kube-burner workload for VirtualMachine (VM) deployment at scale on OpenShift, with an additional focus on storage attachments and cloning.

The goal of this workload example is to exercise a few different areas of the OpenShift cluster and Virtualization control plane, as well as stress storage functionality provided by the Container Storage Interface (CSI) and underlying StorageClass through the use of DataVolume cloning and multiple disk attachments in each VM. This guide will walk through all the details of running a custom workload so that the examples provided can be extended in many different ways by changing the underlying templates and job actions.

Workload tool overview

The kube-burner tool can be built from source or users can download the latest release binary from the repository.

The tool can be executed in a number of ways, for instance by using the OpenShift wrapper functionality that can run predefined workload types and index the results into an ElasticSearch instance, here is one example:

```
./kube-burner-ocp cluster-density-v2 --qps=20 --burst=20 --gc=true
--iterations=50 --churn=false --es-server=https://my-es.com
--es-index=my-kube-burner
```

However, in order to simplify the infrastructure needs for the custom examples provided in this guide, we will instead use the “local” indexer type which will save the run summary and metric data to a local directory.

The examples below will focus on running a custom workload which can be launched using the “init” action and the -c configuration flag pointing to a custom YAML defining the workload, for example:

```
./kube-burner init  -c my-workload.yml
```

The tool provides many configuration options for the workload as described in the reference section, including what measurements and metrics to collect, submission options that control how the objects are created, and even some options around “churning” jobs. This guide will walk through some of the workload configuration options in more detail with a full custom workload example.

Kube-burner jobTypes can be used to measure the deletion of certain objects as part of the workload, and users can also clean up all objects created by the benchmark using the “destroy” action, by passing the uuid of the run to be deleted, for example:

```
./kube-burner destroy  --uuid=<run_uuid>
```

Finally, users can control the measurements and metrics collected during the run, and the thresholds to be considered a “passing” run. In these examples our main focus is vmiLatency, reported in milliseconds, which enables the collection of latency measurements for various stages of the pod, VM, and VMI objects throughout their lifecycle. This measurement includes a breakdown of latency for the virt-launcher pod creation, scheduling, initialization, and ready stages, as well as the VMI (running instance of the VM object) created, scheduling, pending, and ready stages until the full “VMReady” state is reached.

The rest of this guide will go through the workload customization details and finally bring it all together with an example full run and result analysis.

Custom workload

In this case, the goal is to customize a workload to stress both VM and storage deployments.

Note: for clusters where VM deployment at scale is the goal, for best performance it is recommended to enable the “HighBurst” setting in OpenShift Virtualization:

```
oc patch -n openshift-cnv hco kubevirt-hyperconverged --type=json
-p='[{"op": "add", "path": "/spec/tuningPolicy", "value": "highBurst"}]'
```

See the Tuning and Scaling guide to learn more.

Templates

In order for the custom workload configuration to deploy the desired objects, the YAML definitions should be created with any appropriate overrides. In this case all the files are stored in a local `templates` directory (this path can be customized in the workload configuration file).

This example is defining 3 types of objects, variables are used to provide convenient user overrides however aren't necessary in the definitions other than the Replica variable for objects that should be scaled up (the VM definition in this case).

DataVolume Source

In this workload configuration, the goal is to create a VM boot source that can easily be cloned to start many VMs in parallel. In this case a qcow2 is being referenced by an http url to provide this source image, an example url is provided later in the workload configuration.

Here is the full `templates/dv-source.yml` example:

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: dv-source
spec:
  source:
    http:
      ## source image url:
      url: "{{.url}}"
  pvc:
      accessModes:
      - ReadWriteMany
      resources:
      requests:
      storage: {{.storage}}
      volumeMode: {{.volumemode}}
      storageClassName: {{.storageclass}}
```

DataVolume VolumeSnapshot

To more efficiently clone the source PVC at scale, it is recommended to create a VolumeSnapshot (assuming the StorageClass in use supports this), then each VM disk clone will reference this snapshot as the source.

Here is the full `templates/dv-volsnap.yml` example:

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: dv-volsnap
spec:
  volumeSnapshotClassName: {{.volsnapclass}}
  source:
      persistentVolumeClaimName: dv-source
```

VirtualMachine

There is plenty of flexibility in the exact VM definition, but the key for the scale testing goal covered here is to exercise source image cloning, which is achieved by using a `snapshot` source in the `dataVolumeTemplates` section, and to stress additional PVC attachments, which is achieved by creating an additional “blank” disk -- more blank disks could be added as needed to test more attachments per VM. Note that in this configuration, the `dataVolumeTemplates` are defined inside the VM definition, meaning that the clone and blank PVC follows the lifecycle of the VM object which is the goal of this particular scale test, however the DataVolume definitions could be separated from the VM if desired.

Here is the full `templates/vm-dv.yml` example:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  labels:
      kubevirt-vm: vm-{{.name}}-{{.Replica}}
  name: {{.name}}-{{.Replica}}
spec:
  ## controls if the VM boots up on creation:
  running: {{.VMIrunning}}
  template:
      metadata:
      labels:
      kubevirt-vm: vm-{{.name}}-{{.Replica}}
      kubevirt.io/os: {{.OS}}
      spec:
      domain:
      ## vcpus:
      cpu:
            cores: {{.cpuCores}}
      ## create 3 VM disks (rootdisk clone, cloudinit, blank)
      devices:
            disks:
            - disk:
            bus: virtio
            name: rootdisk
            - disk:
            bus: virtio
            name: cloudinitdisk
            - disk:
            bus: virtio
            name: blank-1
            ## default network interface:
            interfaces:
            - masquerade: {}
            model: virtio
            name: default
```

```
            networkInterfaceMultiqueue: true
            rng: {}
      ## VM memory size:
      resources:
            requests:
            memory: {{.memory}}
      ## default pod network:
      networks:
      - name: default
      pod: {}
      ## volume definitions for the VM disks:
      volumes:
      - dataVolume:
            name: dvclone-{{.Replica}}
      name: rootdisk
      - cloudInitNoCloud:
            userData: |-
            #cloud-config
            password: fedora
            chpasswd: { expire: False }
      name: cloudinitdisk
      - dataVolume:
            name: blank-1-{{.Replica}}
      name: blank-1
## Data Volume population of the volumes:
dataVolumeTemplates:
## rootdisk:
- metadata:
      name: dvclone-{{.Replica}}
      spec:
      source:
      snapshot:
      name: dv-volsnap
      storage:
      accessModes:
      - ReadWriteMany
      resources:
      requests:
            storage: {{.storage}}
      storageClassName: {{.storageclass}}
      volumeMode: {{.volumemode}}
## blank disk:
- metadata:
      name: blank-1-{{.Replica}}
      spec:
      source:
      blank: {}
      pvc:
      accessModes:
```

```
      - ReadWriteMany
      resources:
      requests:
            storage: 500Mi
      storageClassName: {{.storageclass}}
      volumeMode: {{.volumemode}}
```

Workload configuration

Configuring the workload definition and job order is important to deploy the objects in a functional manner and to measure the intended operations.

Here is the full `vm-dvclone-density.yml` workload configuration example, each section is explained in more detail below:

```
metricsEndpoints:
  - indexer:
      type: local
global:
  measurements:
  - name: vmiLatency

jobs:

  - name: dv-source
      jobType: create
      jobIterations: {{.ITERATIONS}}
      namespacedIterations: true
      namespace: vm-density
      qps: {{.QPS}}
      burst: {{.BURST}}
      maxWaitTimeout: 1h
      jobPause: 5m
      objects:

      - objectTemplate: templates/dv-source.yml
      replicas: 1
      inputVars:
      url:
https://dl.fedoraproject.org/pub/fedora/linux/releases/40/Cloud/x86_64/imag
es/Fedora-Cloud-Base-Generic.x86_64-40-1.14.qcow2
      storage: 10Gi
      storageclass: ocs-storagecluster-ceph-rbd
      volumemode: Block

      - objectTemplate: templates/dv-volsnap.yml
      replicas: 1
      inputVars:
```

```
    volsnapclass: ocs-storagecluster-rbdplugin-snapclass

- name: vm-density
    jobType: create
    jobIterations: {{.ITERATIONS}}
    qps: {{.QPS}}
    burst: {{.BURST}}
    namespacedIterations: true
    namespace: vm-density
    maxWaitTimeout: 1h
    jobPause: 1m
    objects:

    - objectTemplate: templates/vm-dv.yml
    replicas: {{ .OBJ_REPLICAS }}
    inputVars:
    name: vm-dv-density
    OS: fedora
    cpuCores: 1
    memory: 1G
    storage: 10Gi
    storageclass: ocs-storagecluster-ceph-rbd
    volumemode: Block
    VMIrunning: true

- name: delete-job
    jobType: delete
    waitForDeletion: true
    qps: {{.QPS}}
    burst: {{.BURST}}
    jobPause: 1m
    objects:

    - kind: VirtualMachine
    labelSelector: {kube-burner-job: vm-density}
    apiVersion: kubevirt.io/v1

    - kind: VirtualMachineInstance
    labelSelector: {kube-burner-job: vm-density}
    apiVersion: kubevirt.io/v1

    - kind: Pod
    labelSelector: {kubevirt.io: virt-launcher}
    apiVersion: v1

    - kind: Namespace
    labelSelector: {kube-burner-job: vm-density}
```

First, the configuration starts by defining a “local” indexer type (i.e. save to local directory) and the desired vmiLatency measurement. Then each jobType is defined in order. Note that each job could be further configured by changing any of the default parameters.

Some general parameters to understand for these jobs:

- `namespacedIterations: true` means that each set of object replicas (i.e. an “iteration”) will be created in a new namespace, the total number of namespaces is controlled by `jobIterations`
- `qps` and `burst` settings control how quickly kube-burner will submit the object creation queries, the example run below will set these parameters to 1,000 which allows a high creation rate so that the benchmark is not introducing throttling itself
- the `objects` section describes what objects are to be associated with that job (based on the template definitions)
- `maxWaitTimeout` controls how long the benchmark waits before it may determine a failure
- `jobPause` is a pause introduced after the job is complete, pausing after each job type can be useful to help separate actions for observability and measurement
- optionally, a `jobIterationDelay` could also be added to control the run in smaller “batch” increments, this delay runs between each job iteration and each delete operation

The first job creates the DataVolume source and VolumeSnapshot, in this case it is configured to create a single source and snapshot per benchmark namespace (i.e. replica 1). The storageclass and volsnapclass are set to use OpenShift Data Foundation defaults in this configuration and an example Fedora qcow2 url is provided. Note that if there is a very long delay in data source import actions, the `jobPause` may need to be increased before moving on to the next stage.

The second job creates the VMs, which in turn (based on the example object definition) create the snapshot clones and empty PVCs for the VM disks. This job is also namespaced, and in this case $OBJ_REPLICAS number of VMs are created per $ITERATIONS number of namespaces. This job specifically starts the VMs as `running: true` and waits for the VMI (VirtualMachineInstance) Running state for the latency measurements.

Finally, this configuration measures the deletion of objects with the delete-job. Note that currently this doesn’t delete the DataVolume sources and VolumeSnapshots, although the kube-burner `destroy` action can be used after a run, and the default behavior is `cleanup: true` which will delete previously created namespaces on the next run.

VM and storage deployment at scale

Below is a full walk through of running the custom workload:

Install the tool:

```
wget https://github.com/kube-burner/kube-burner/releases/download/v1.10.1/kube-burner-V1.10.1-linux-x86_64.tar.gz

tar zxvf kube-burner-V1.10.1-linux-x86_64.tar.gz

## create the “vm-dvclone-density.yml” workload config file and the “templates/dv-source.yml” and “templates/dv-volsnap.yml” definition files in the local directory
```

Run the workload using the example configurations detailed above and define any variables, in the example below the benchmark will create 10 namespaces, each with 50 VMs, in parallel (i.e. 500 total VMs, each VM has 1 boot drive PVC and 1 blank PVC).

> *Keep in mind that max object density is dependent on each cluster environment including total schedulable resources, storage capacity, and total pod capacity as determined by the maxPods setting on each worker.*
> *Note that the benchmark is executed against the cluster defined by the $KUBECONFIG environment variable, $HOME/.kube/config, or in-cluster config, in that order.*

```
# ITERATIONS = number of namespaces to create
# OBJ_REPLICAS = number of VMs to create per namespace

QPS=1000 BURST=1000 ITERATIONS=10 OBJ_REPLICAS=50 ./kube-burner init  -c vm-dvclone-density.yml
```

The tool will log status as the run progresses, and assuming all VMIs successfully reach Running state the workload will end with a message similar to the following:

```
level=info msg="👋 Exiting kube-burner <run_uuid>" file="kube-burner.go:85"
```

Since the configuration is using the “local” indexing type, the results will be stored in a local directory called `collected-metrics`. This will include some general information about the run in `jobSummary.json`, detailed per object measurements in `podLatencyMeasurement-vm-density.json` and finally a summary of the latency performance in `podLatencyQuantilesMeasurement-vm-density.json` which includes max, avg, and P99/P95/P50 measurements (in ms) for various “quantileName”s for the different stages of the pod and VM lifecycle. In this case the “VMReady” measurement is the most interesting in terms of total VM deployment performance, however the other measurements may be of interest to understanding the overall behavior as well.

Adding Metrics

Optionally, metric data can also be captured during a workload by adding a metric endpoint definition.

The following section walks through how to add metrics collection to the example run shown above, in this case we will use the `kubevirt-metrics.yaml` metric definitions provided by the kube-burner repository, but any set of metric definitions can be configured as needed:

```
## save the metric definition file to the local directory:
wget
https://raw.githubusercontent.com/kube-burner/kube-burner/main/examples/met
rics-profiles/kubevirt-metrics.yaml

## Create the metric endpoint definition, referencing the metric file name:
cat << EOF >>  metrics-endpoint.yml
- endpoint: {{ .PROM }}
  token: {{.PROM_TOKEN}}
  metrics:
      - kubevirt-metrics.yaml
  indexer:
      type: local
EOF

## Run the workload, providing the PROM url and PROM_TOKEN for the cluster as shown
below, adding the -e flag for the metrics endpoint configuration:

PROM=https://$(oc get route -n openshift-monitoring prometheus-k8s -o
jsonpath="{.spec.host}") PROM_TOKEN=$(oc create token -n
openshift-monitoring prometheus-k8s) QPS=1000 BURST=1000 ITERATIONS=10
OBJ_REPLICAS=50 ./kube-burner init  -c vm-dvclone-density.yml -e
metrics-endpoint.yml
```

Since the "local" indexer type is used in these examples, after the run completes it will produce multiple json files in the collected-metrics directory along with the run results detailing all the data captured from the metric queries.

Conclusion

Hopefully this guide has provided some useful end-to-end examples walking through creation of a custom kube-burner workload to test VM deployments at scale while also exercising storage scalability. The examples can be modified as needed to adjust to other specific testing needs, but the general workflow is to determine which objects need to be created, the order of the job actions, and the metrics and measurements of interest. The same sort of workload configuration could be created for a pod and PVC scale case.

Generally for scale testing a certain max density target is the end goal, however it is often also useful to scale up from smaller batch sizes to observe how performance may change as the

overall cluster becomes more loaded. In many VM and CSI scale cases it is also important to exercise scaling in a multi-worker environment and, ideally, to further stress the underlying compute and storage resources by running a workload in the VMs so that they perform some amount of I/O to the attached PVCs. Also keep in mind that scaling performance evaluations during OpenShift cluster, Virtualization operator, and Storage operator upgrades remains an important use case to harden for your environment.

Happy benchmarking!