Scheduler Performance Test

Motivation

We already have a performance testing system -- Kubemark. However, Kubemark requires setting up and bootstrapping a whole cluster, which takes a lot of time.

We want to have a standard way to reproduce scheduling latency metrics result and benchmark scheduler as simple and fast as possible. We have the following goals:

- Save time on testing
  - The test and benchmark can be run in a single box. We only set up components necessary to scheduling without booting up a cluster.
- Profiling runtime metrics to find out bottleneck
  - Write scheduler integration test but focus on performance measurement. Take advantage of go profiling tools and collect fine-grained metrics, like cpu-profiling, memory-profiling and block-profiling.
- Reproduce test result easily
  - We want to have a known place to do the performance related test for scheduler. Developers should just run one script to collect all the information they need.

Currently the test suite has the following:

- density test (by adding a new Go test)
  - schedule 30k pods on 1000 (fake) nodes and 3k pods on 100 (fake) nodes
  - print out scheduling rate every second
  - let you learn the rate changes vs number of scheduled pods
- benchmark
  - make use of `go test -bench` and report nanosecond/op.
  - schedule b.N pods when the cluster has N nodes and P scheduled pods. Since it takes relatively long time to finish one round, b.N is small: 10 - 100.

How To Run

Density tests

```
# In Kubernetes root path
make test-integration WHAT=./test/integration/scheduler_perf ETCD_LOGLEVEL=warn
KUBE_TEST_VMODULE="''" KUBE_TEST_ARGS="-alsologtostderr=true -logtostderr=true -
run=." KUBE_TIMEOUT="--timeout=60m" SHORT="--short=false"
```

Benchmark tests

```
# In Kubernetes root path
make test-integration WHAT=./test/integration/scheduler_perf ETCD_LOGLEVEL=warn
KUBE_TEST_VMODULE="''" KUBE_TEST_ARGS="-alsologtostderr=false -logtostderr=false -
run=^$$ -benchtime=1ns -bench=BenchmarkPerfScheduling"
```

The benchmark suite runs all the tests specified under config/performance-config.yaml.

Once the benchmark is finished, JSON file with metrics is available in the current directory (test/integration/scheduler_perf). Look for `BenchmarkPerfScheduling_YYYY-MM-DDTHH:MM:SSZ.json`. You can use `-data-items-dir` to generate the metrics file elsewhere.

In case you want to run a specific test in the suite, you can specify the test through `-bench` flag:

Also, bench time is explicitly set to 1ns (`-benchtime=1ns` flag) so each test is run only once. Otherwise, the golang benchmark framework will try to run a test more than once in case it ran for less than 1s.

```
# In Kubernetes root path
make test-integration WHAT=./test/integration/scheduler_perf ETCD_LOGLEVEL=warn
KUBE_TEST_VMODULE="'''" KUBE_TEST_ARGS="-alsologtostderr=false -logtostderr=false -
run=^$$ -benchtime=1ns -
bench=BenchmarkPerfScheduling/SchedulingBasic/5000Nodes/5000InitPods/1000PodsToSchedule
```

To produce a cpu profile:

```
# In Kubernetes root path
make test-integration WHAT=./test/integration/scheduler_perf KUBE_TIMEOUT="-
timeout=3600s" ETCD_LOGLEVEL=warn KUBE_TEST_VMODULE="'''" KUBE_TEST_ARGS="-
alsologtostderr=false -logtostderr=false -run=^$$ -benchtime=1ns -
bench=BenchmarkPerfScheduling -cpuprofile ~/cpu-profile.out"
```

How to configure benchmark tests

Configuration file located under `config/performance-config.yaml` contains a list of templates. Each template allows to set:

- node manifest
- manifests for initial and testing pod
- number of nodes, number of initial and testing pods
- templates for PVs and PVCs
- feature gates

See `op` data type implementation in scheduler_perf_test.go for available operations to build `WorkloadTemplate`.

Initial pods create a state of a cluster before the scheduler performance measurement can begin. Testing pods are then subject to performance measurement.

The configuration file under `config/performance-config.yaml` contains a default list of templates to cover various scenarios. In case you want to add your own, you can extend the list with new templates. It's also possible to extend `op` data type, respectively its underlying data types to extend configuration of possible test cases.