SelfTune: Tuning Cluster Managers

Microsoft Research + Microsoft [Office 365, Azure]

NSDI
April 2023
Cluster management frameworks

- Kubernetes
- Azure Kubernetes Service
- Borg
- Twine

container allocation, scheduling, VM placement
EXO Workload Manager

Schedules background jobs on Substrate machines

Cluster Manager

Adjust concurrency limit of disk
(#background tasks that can simultaneously access disk)

change state

measure health

CPU utilization, disk latency, network loss, ....

Thousands of lines of code implementing the scheduler heuristics

Hundreds of configuration parameters tweaked by experts.

(for different forests, machine types, resource types, job types, etc.)
Kubernetes

Cluster Manager

- deploy new container, resize container memory
- change state
- measure health

Thousands of lines of code implementing the control plane heuristics

CPU utilization, memory utilization, pod evictions...

Hundreds of configuration parameters
var (  
metricsFetcherInterval = flag.Duration("recommender-interval", 1*time.Minute, "How often metrics should be fetched")  
checkpointsGCInterval = flag.Duration("checkpoints-gc-interval", 10*time.Minute, "How often orphaned checkpoints should be garbage collected")  
prometheusAddress = flag.String("prometheus-address", "", "Where to reach for Prometheus metrics")  
prometheusJobName = flag.String("prometheus-cadvisor-job-name", "kubernetes-cadvisor", "Name of the prometheus job name which scrapes the cAdvisor metrics")  
address = flag.String("address", "", "The address to expose Prometheus metrics.")  
kubeconfig = flag.String("kubeconfig", "", "Path to a kubeconfig. Only required if out-of-cluster.")  
kubeApiQps = flag.Float64("kube-api-qps", 5.0, "QPS limit when making requests to Kubernetes apiserver")  
kubeApiBurst = flag.Float64("kube-api-burst", 10.0, "QPS burst limit when making requests to Kubernetes apiserver")
)

storage = flag.String("storage", "", "Specifies storage mode. Supported values: prometheus, checkpoint (default)")

// prometheus history provider configs
historyLength = flag.String("history-length", "8d", "How much time back prometheus have to be queried to get historical metrics")
historyResolution = flag.String("history-resolution", "1h", "Resolution at which Prometheus is queried for historical metrics")
queryTimeout = flag.String("prometheus-query-timeout", "5s", "How long to wait before killing long queries")
podLabelPrefix = flag.String("pod-label-prefix", "pod_label_", "Which prefix to look for pod labels in metrics")
podLabelsMetricName = flag.String("metric-for-pod-labels", "up\(job\("kubernetes-pods\")\)", "Which metric to look for pod labels in metrics")
podNamespacelabel = flag.String("pod-namespace-label", "kubernetes_namespace", "Label name to look for pod namespaces")
podNameLabel = flag.String("pod-name-label", "kubernetes_pod_name", "Label name to look for pod names")
ctrNamespacelabel = flag.String("container-namespace-label", "namespace", "Label name to look for container namespaces")
ctrPodNameLabel = flag.String("container-pod-name-label", "pod_name", "Label name to look for container pod names")
ctrNameLabel = flag.String("container-name-label", "name", "Label name to look for container names")

vpaObjectNameSpace = flag.String("vpa-object-namespace", apiV1.NamespaceAll, "Namespace to search for VPA objects and pod stats. Empty means all namespaces will be used.")
)

// Aggregation configuration flags
var (  
memoryAggregationInterval = flag.Duration("memory-aggregation-interval", model.DefaultMemoryAggregationInterval, "The length of a single interval, for which the peak memory aggregate")
memoryAggregationIntervalCount = flag.Int64("memory-aggregation-interval-count", model.DefaultMemoryAggregationIntervalCount, "The number of consecutive memory-aggregation-intervals")
memoryHistogramDecayHalfLife = flag.Duration("memory-histogram-decay-half-life", model.DefaultMemoryHistogramDecayHalfLife, "The amount of time it takes a historical memory histogram to decay to half the value")
cpuHistogramDecayHalfLife = flag.Duration("cpu-histogram-decay-half-life", model.DefaultCPUMemoryHistogramDecayHalfLife, "The amount of time it takes a historical CPU usage sample to decay to half the value")
)
Configuring cluster managers

• Domain expertise, (limited) empirical evaluations; can be sub-optimal
• Global, static configuration of cluster managers; can be sub-optimal
  • What works for one cluster need not work for another
  • What works for one cluster today may not work next week, as workload patterns drift or hardware is replaced
UseCase:
I have an app with low CPU usages over the weekends and the VPA CPU recommendation dips respectively. However, is it possible to configure the recommender in a way to not be influenced by the two day dip? I’d like to keep the recommendation the same as during regular usage days.

Here is a chart of my CPU usage / VPA CPU recommendation:
CPU usage - solid line
VPA CPU recommendation - dotted line

The reason I don't want the recommended CPU limit to dip is a risk factor because of how much business value the app brings and I'm willing to "waste" the CPUs over the weekend to prepare for anything.

Expected:
VPA CPU limit recommendation to stay consistent to the majority of the usages and not dip due to the two days of low usage on weekends.

https://github.com/kubernetes/autoscaler/issues/3684
Configuring cluster managers

• Domain expertise, (limited) empirical evaluations; can be sub-optimal
• Global, static configuration of cluster managers; can be sub-optimal
  • What works for one cluster need not work for another
  • What works for one cluster today may not work next week, as workload patterns drift or hardware is replaced

• Options for developers/infrastructure team:
  • Create and manage multiple configuration files for different environments
  • Use hyperparameter search/sampling/simulation mechanisms [CherryPick ‘17, BestConfig ‘17, Metis ‘18, MLOS ‘20, …]

None of these options is satisfactory
SelfTune: A framework to seamlessly tune

Kubernetes

- deploy new container, resize container memory
- change state
- measure health
  - CPU utilization, memory utilization, pod evictions

Developer specifies just what is needed to drive the system towards desired states.

1. params = selfTune.Predict()

// re-configure Kubernetes with params

2. selfTune.SetReward(numPodEvictionsDaily)
Cluster management is classic RL setting: observe health, deploy action, and repeat. We can piggyback on this setup to tune the underlying parameters.
Key Features

• (Almost) zero engineering overhead
  • integration, computational

• “Implicit context” – developers choose the granularity at which parameters need to be tuned
  • Machine-level? Cluster-level?
  • Application-level? Workload-level?

• Minimal assumptions

State-of-the-art RL systems for parameter tuning
[Decima ‘19, CDBTune ‘19, MS Decision Service ‘16]
feature engineering, complexity, categorical parameters
Environments, rewards, optimal configs vary with time

• Online tuning of parameters
• Sample complexity (# rounds it takes for the algorithm to catch up with an oracle) needs to be small

Bayesian Optimization common but not ideal in our setting
We use a simple and intuitive state-of-the-art algorithm for “continuous bandits”

Optimal regret algorithm for Pseudo-1d Bandit Convex Optimization.
Evaluation: Three cluster management settings

• Kubernetes
  • Focus: optimal latency for containerized applications
  • Results on DeathStar benchmark with a social networking application

• EXO Workload Manager
  • Focus: optimal cluster resource utilization & throughput for workloads
  • Results from months of deployment in LAM, NAM, APAC forests

• Azure Functions ("FaaS")
  • Focus: optimal latency for the cloud users and save costs for Azure
  • Results on full set of Azure traces (2M applications, >10M daily invocations, 4 months)
Revisiting the Kubernetes example

**Workload:** HTTP requests to service endpoint

**Implicit context:** Workload-level tuning

1. `params = SelfTune.Predict()`
2. `SelfTune.SetReward(P95latency)`

<table>
<thead>
<tr>
<th>Metric</th>
<th>SelfTune</th>
<th>BO</th>
<th>eps-greedy</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>P95 latency (ms)</td>
<td>19.5</td>
<td>19.9</td>
<td>20.0</td>
<td>31.1</td>
</tr>
<tr>
<td># Samples</td>
<td>8</td>
<td>41</td>
<td>30</td>
<td>-</td>
</tr>
<tr>
<td>P50 iter. cost (ms)</td>
<td>20.5</td>
<td>23.3</td>
<td>29.2</td>
<td>-</td>
</tr>
<tr>
<td>P75 iter. cost (ms)</td>
<td>21.1</td>
<td>33.0</td>
<td>33.2</td>
<td>-</td>
</tr>
<tr>
<td>P95 iter. cost (ms)</td>
<td>28.3</td>
<td>76541.9</td>
<td>67640.3</td>
<td>-</td>
</tr>
</tbody>
</table>
Evaluation: Three cluster management settings

• Kubernetes
  • Focus: optimal latency for containerized applications
  • Results on DeathStar benchmark with a social networking application

• EXO Workload Manager
  • Focus: optimal cluster resource utilization & throughput for workloads
  • Results from months of deployment in LAM, NAM, APAC forests

• Azure Functions (“FaaS”)
  • Focus: optimal latency for the cloud users and save costs for Azure
  • Results on full set of Azure traces (2M applications, >10M daily invocations, 4 months)
EXO Workload Manager

**Workload:** Live production scheduling workloads

**Implicit context:** Machine-level tuning

1. `params = SelfTune.Predict()`
2. `SelfTune.SetReward(GrantRatio)`

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Reward</th>
<th>Metric Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Res Utilization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P25</td>
</tr>
<tr>
<td>1</td>
<td>Throughput</td>
<td>SI</td>
</tr>
<tr>
<td>2</td>
<td>Throughput</td>
<td>SI</td>
</tr>
<tr>
<td>3</td>
<td>Res Utilization</td>
<td>2%</td>
</tr>
</tbody>
</table>
Evaluation: Three cluster management settings

• Kubernetes
  • Focus: optimal latency for containerized applications
  • Results on DeathStar benchmark with a social networking application

• EXO Workload Manager
  • Focus: optimal cluster resource utilization & throughput for workloads
  • Results from months of deployment in LAM, NAM, APAC forests

• Azure Functions (“FaaS“)
  • Focus: optimal latency for the cloud users and save costs for Azure
  • Results on full set of Azure traces (2M applications, >10M daily invocations, 4 months)
## Azure Function-as-a-Service

### Comparison Table

<table>
<thead>
<tr>
<th></th>
<th>Bare Metal</th>
<th>VMs (IaaS)</th>
<th>Containers</th>
<th>Functions (FaaS)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unit of Scale</strong></td>
<td>Server</td>
<td>VM</td>
<td>Application/Pod</td>
<td>Function</td>
</tr>
<tr>
<td><strong>Provisioning</strong></td>
<td>Ops</td>
<td>DevOps</td>
<td>DevOps</td>
<td>Cloud Provider</td>
</tr>
<tr>
<td><strong>Init Time</strong></td>
<td>Days</td>
<td>~1 min</td>
<td>Few seconds</td>
<td>Few seconds</td>
</tr>
<tr>
<td><strong>Scaling</strong></td>
<td>Buy new hardware</td>
<td>Allocate new VMs</td>
<td>1 to many, auto</td>
<td>0 to many, auto</td>
</tr>
<tr>
<td><strong>Typical Lifetime</strong></td>
<td>Years</td>
<td>Hours</td>
<td>Minutes</td>
<td>O(100ms)</td>
</tr>
<tr>
<td><strong>Payment</strong></td>
<td>Per allocation</td>
<td>Per allocation</td>
<td>Per allocation</td>
<td>Per use</td>
</tr>
<tr>
<td><strong>State</strong></td>
<td>Anywhere</td>
<td>Anywhere</td>
<td>Anywhere</td>
<td>Elsewhere</td>
</tr>
</tbody>
</table>
Azure Function-as-a-Service

Keeping functions in memory indefinitely.

Removing function instance from memory after invocation.
Hybrid Histogram Policy

App-specific tuning of policy parameters with **SelfTune**

1. pw, ka = SelfTune.Predict()

   // re-configure hybrid histogram policy with params

2. SelfTune.SetReward (memoryWastage)

**Parameters:** Prewarm and Keepalive

**Workload:** Azure Function traces

**Implicit context:** Application-level tuning

---

**Histogram policy**

- Evict function, Load function
- change state
- measure health
- Memory utilization, Cold starts

---
About 10%-12% reduction in mem wastage over 3 months of Azure traces, compared to ka=99, pw=5
Summary

- Minimal engineering overhead
- Online tuning with small sample complexity
- Versatile – implicit context
- Success on production workloads