Tiny Autoscalers for Tiny Workloads: Dynamic CPU Allocation for Serverless Functions

Yuxuan Zhao, Alexandru Uta

Presentation by Steven Golob

November 26, 2024

Zhao, Yuxuan, and Alexandru Uta. "Tiny autoscalers for tiny workloads: Dynamic CPU allocation for serverless functions." *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022.

The Problem

- **Client code might be throttled**
  - **Users select memory, and given static, limited CPU**

    "Although they are short- running, serverless functions exhibit dynamic and non-trivial resource usage, which makes it difficult for their authors to estimate correctly the amount of resources to be requested from the cloud provider"
- **Provider might have overprovisioned, under-utilized resources**
  - **many containers are alive, and idle to stay warm**

Why it’s important

- Most research has been on memory utilization and CPU utilization for long-running jobs
- Plus, these dynamic allocation algorithms are:
    - computationally expensive, and
    - rely on historical data that is not available for short-running jobs

Key Idea:

Instead of providing proportional-to-memory amount of vCPUs, dynamically allocate vCPUs through “tiny” autoscaling

Demonstrate the efficiency and feasibility of this approach

Implemented on top of Kubernetes, overriding the default autoscaling algorithm

Lightweight algorithms studied (i.e. “tiny autoscalers”)

- Simple moving average (SMA)
- Exponential moving average (EMA)

Compare with:

- Holt-Winters exponential smoothing (HW)
- long short-term memory (LSTM)

Kubernetes Default:

- Vertical Pod Autoscaler (VPA) Recommender

Architecture Overview

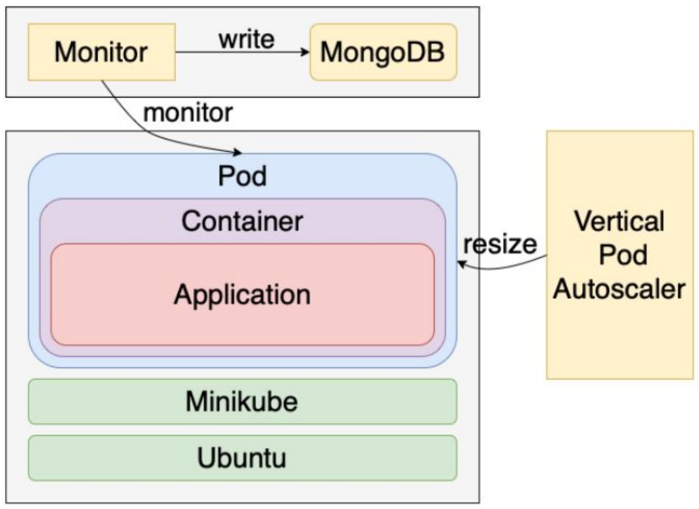

Fig. 1. The architecture overview of our system. We deploy a pod containing the application on minikube. The pod is monitored and the monitoring data is stored in a MongoDB database. A VPA is attached to the pod for resizing.

Metric Collection:

reports via “linux cgroups”

collected by a “cAdvisor” (open-source Google project)

cAdvisor integrated into kubelet (running on each node)

Also monitored with *kubernetes.client.CustomObjectsApi*, and *kubernetes.client.CoreV1Api*, *kubernetes.client.ApiClient*.

VPA

1. **recommender**
2. **updater**
3. **admission controller**

This is what they replace with tiny autoscalers

VPA

Simple algorithm that provisions resources based on decaying, “moving window” histogram of CPU usage

Main drawback: Does not respond to short, sudden workload changes

HW and LSTM

Utilize machine learning based on observations of past utilization

Authors do not integrate into VPA, and *emulate* them

Require lots of historical “training” data

Their solution

Inspired by web-based CPU-usage prediction systems [24]

- **SMA load tracker** $$SMA(S_n(t_i)) = \frac{\sum_{i-n \le j \le i} s_j}{n+1}$$
- **EMA load tracker** $$EMA(S_n(t_i)) = \alpha * s_i + (1-\alpha) * \mathrm{EMA}(S_n(t_{i-1})) \qquad \text{if } i > n$$

Drawback: parameters need to be fine-tuned to each task

Customizations

- **Tuned to prefer slight over-provisioning**
- **a novel “bottoming” mechanism**
  - **prevents rapid drops in CPU allocation**
  - **ensures quick rises in response to peak CPU usage**
- **All parameters tuned for much more responsive adjustments from original SMA / EMA algorithm**

Experiments

One criticism:

VPA does not update resources in-place.

so experiments here do not incorporate update mechanism, and only consider the recommendations. So empirical evaluation is severely hampered.

Experiments

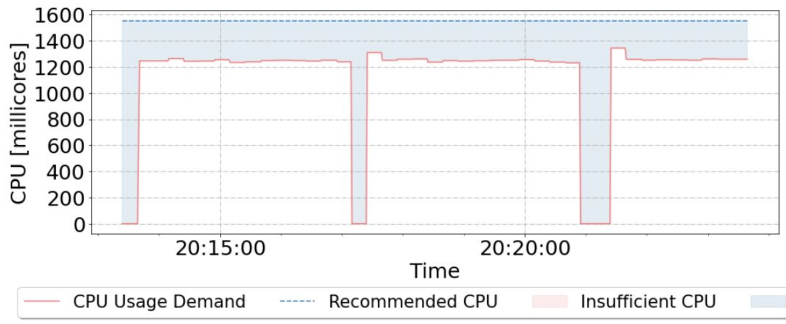

Fig. 3. Result of video_processing_127m with default VPA (metrics interval 1 minute). On this workload, the default VPA recommendation stays unchanged and cannot react with changes of the actual CPU usage.

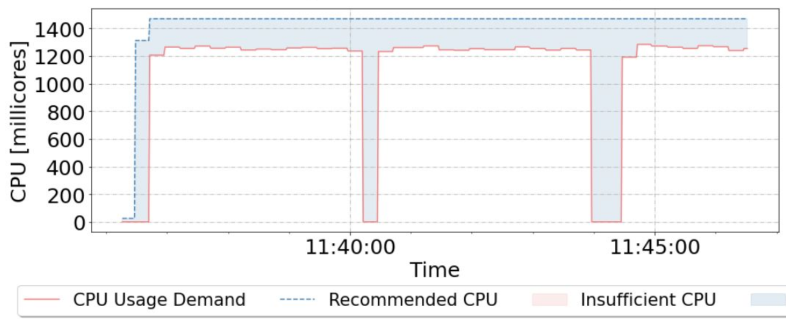

Fig. 4. Result of video_processing_127m with default VPA (metrics interval 1 second). The default VPA with metrics interval as 1 second can follow the trend of the actual CPU usage at the beginning but stays unchanged in the following warm starts.

Experiments

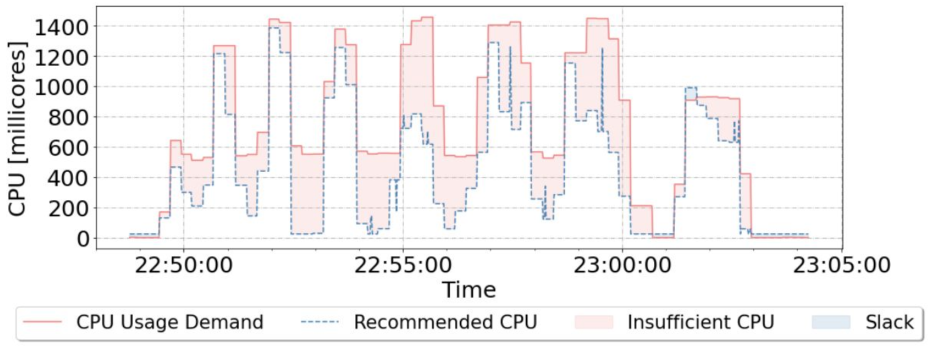

Fig. 5. Default ema5-3 dynamic CPU allocation (before tuning). Notice how the default EMA method consistently allocates insufficient CPU.

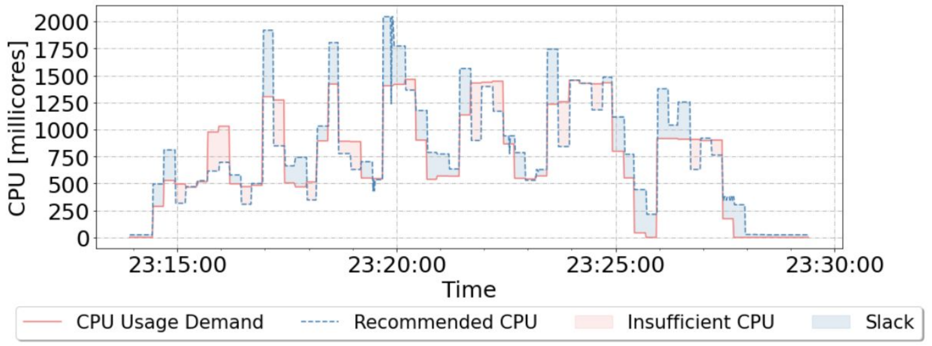

Fig. 6. Dynamic CPU allocation using ema5-3 including our tuning defined in Section III. Notice how the tuned EMA version has a much more balanced CPU allocation.

Experiments

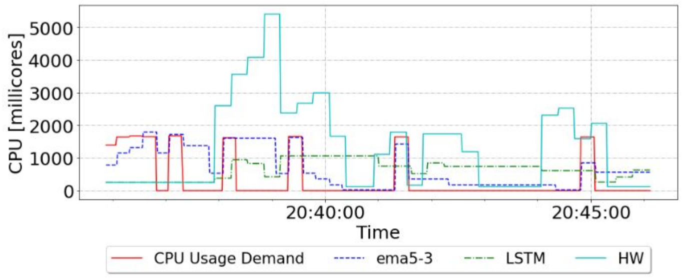

Fig. 11. Comparing HW and LSTM with our ema5-3 running on video_processing_17m workload. Notice our method follows the CPU usage trend more closely than HW and LSTM, offering better performance.

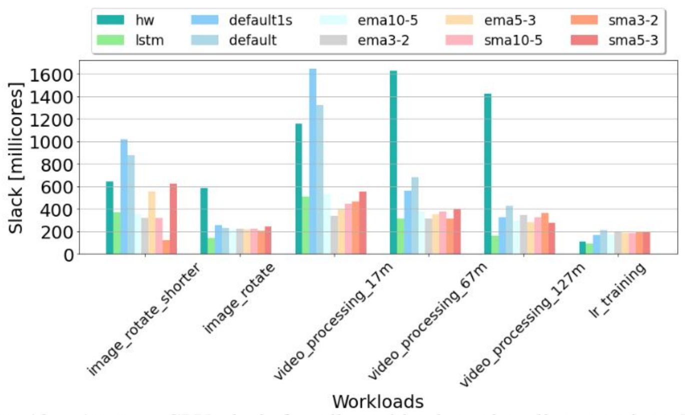

Fig. 12. Average CPU slack for all workloads under all autoscalers. The applications were run several times to emulate serverless warm starts.

Experiments

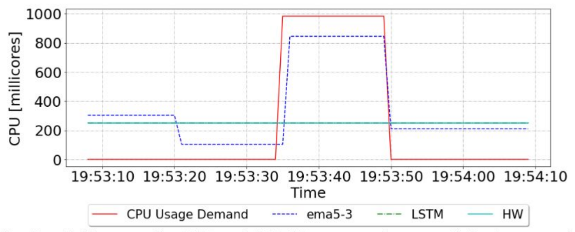

Fig. 7. Cold starts for HW and LSTM compared to ema5-3 when running image_rotate_shorter workload. Due to lack of historical information and training, LSTM and HW cannot allocate sufficient CPU. LSTM and HW curves overlap.

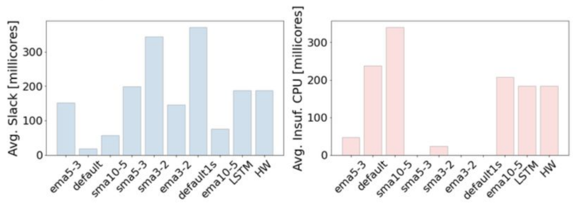

Fig. 8. Comparing the average slack and insufficient CPU among all the methods in this paper when running image_rotate_shorter workload in the cold start. Due to the training process of LSTM and HW, they do not perform well in both slack and insufficient CPU as a result of presenting a preset value.

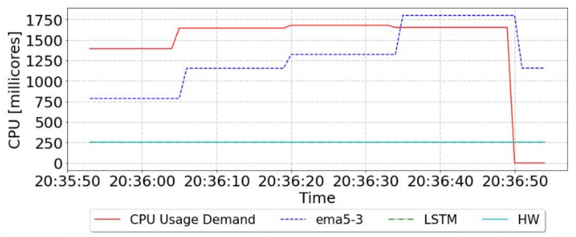

Fig. 9. Cold starts for HW and LSTM compared to ema5-3 when running video_processing_17m workload in the first run. Due to lack of historical information and training, LSTM and HW cannot allocate sufficient CPU. LSTM and HW curves overlap

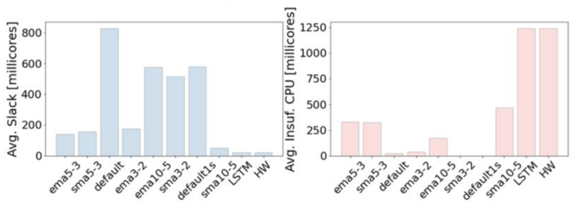

Fig. 10. Comparing the average slack and insufficient CPU among all the methods in this paper when running video_processing_17m workload in the cold start. Due to the training process of LSTM and HW, they do not perform well in both slack and insufficient CPU as a result of presenting a preset value.

Conclusion

•