Efficient Container Allocation for Serverless Computing

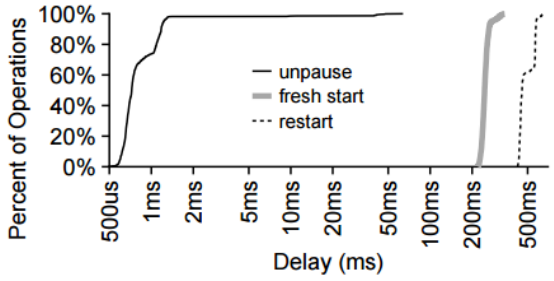

Figure 1: Latency of 3 container transitions: unpause (pause to running), fresh start (start a new one), and restart [5].

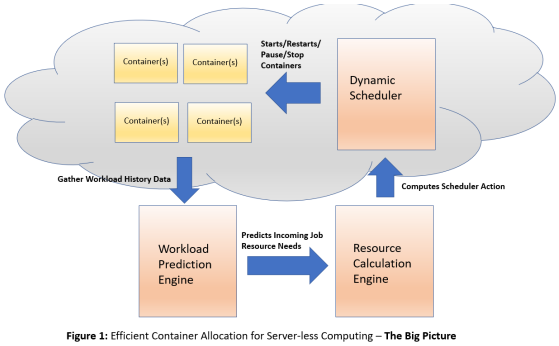

Figure 2: Overall Solution Approach

ACM Reference format:
. 2017. Efficient Container Allocation for Serverless Computing. In *Proceedings of Stony Brook University, Stony Brook, USA, Fall 2017 (AMS-559'97,CSE-591),* 5 pages.
DOI: 10.475/123_4

1 MOTIVATION

The rapid development and innovation in data centers and software platforms, people are allowed to access shared resources in cloud. The resource sharing paradigm has significantly changed from using hardware, virtual machines, and containers. However, managing these shared resources is a plethora of responsibilities and expenses including purchasing and setting up hardware, data center setup, operating system installation and patches, security updates and so on. The majority of consumers using these shared resources being developers or end-users really want to focus on their application or use-case rather than managing these resources. To manage these operational tasks seamlessly and enable more users to access the cloud infrastructure, server-less computing such as Serverless Framework used in AWS Lambda or Azure Functions, Kubelet atop Kubernetes, and Openlambda [2] have been developed recently.Despite the development in hardware and software, serverless frameworks are not mature for users yet.

A significant challenge is how to cut down the latency of initialization phase for job requests. To setup a container for a job, it takes almost 30 seconds in AWS Lambda [3]. To deal with this, AWS Lambda reuses the containers and the latency is cut down to 500 ms [2]. If the containers are maintained but not reused, resources like memory and energy go wasted.

AMS-559'97,CSE-591, Stony Brook, USA

DOI: 10.475/123_4

Problem statement: **In a serverless framework, how to minimize the latency of job initialization ?**

Given the fore-casted arrival workload in a serverless framework, the scheduler needs to decide the following:

- How many containers with specific memory sizes are required to maintain for reuse.

The final goal is to minimize the average setup latency of jobs. It will improve the existing auto-scaling strategy of container allocation ensuring better resource utilization of the infrastructure. Is it straightforward? No, as we can never stop the containers. Jobs have different resource demands and the running container(s) may not be large enough for them to fit in. If we just maintain the large containers, the resources are wasted and we can only fit a few jobs. Furthermore, active containers require higher energy consumption than in-active ones.

Overall Solution Approach: Three components working in tandem to divide and conquer this problem. (Refer to Figure 2)

- First, forecast the future workload using historical data. (Workload Prediction Engine)
- Next, formulate the problem and compute the resources necessary for incoming jobs using optimization techniques. It assuming the availability of workload prediction tom compute the container allocation strategy.(Resource Calculation Engine)
- Finally, develop a dynamic scheduling algorithm for container allocation (Dynamic Scheduler) based on inputs from the resource calculation engine.

2 LITERATURE REVIEW

2.1 Serverless computing platforms

Open Source:

Open Lambda – no recent updates.

Kubernetes Kubelet

- By default, the kubelet agent daemon has the following eviction rule: memory.available less than 100Mi.
- http://bit.ly/2fPUcrE for more details

IBM OpenWhisk

- https://github.com/apache/incubator-openwhisk/blob/master/docs/about.md
- Container is killed if it times-out (10 minutes). (refer: ContainerProxy.scala)

Open Stack

- https://www.youtube.com/watch?v=K8TFN0WU_rM: container is saved for 5 seconds and purged. They keep the container running for the same function handler from the same event type. Performance: 1.6 seconds.

Proprietary:

- *Azure Function* – not disclosed.
- *AWS Lambda Serverless framework* – not disclosed
- *Google Cloud Functions (Alpha)*
- useful resources [1]

2.2 Related work

Workload Prediction A lot of work has been done on the workload prediction as deciding the right amount of resources for a cloud computing environment is a double-edged sword, which may lead to either under-provisioning or over-provisioning.An appropriate workload prediction mechanism would resolve the shortage of hardware resources, waste of unused resources and reduce the energy consumed within the data centers. Some papers have gone with standard statistical models, some have applied only machine learning techniques while others have used a hybrid methodology of combining statistics and machine learning. This paper [2] elaborates on combining appropriate machine-learning methods and statistical methods, assuming that a historical record of workload is available for a specified period of time in the past. It details out about the analysis of time series using correlation, feature scoring and usage of confidence factors along with machine learning techniques to find a good predictive model.

Cache Resource Allocation There have been work that focuses on improving the cache performance, i.e., hit frequency. Cache may have different si Twitter [30] and Facebook [26] have tried to improve Memcached slab class allocation to better adjust for varying item sizes, by periodically shifting pages from slabs with a high hitrate to those with a low hitrate. Dynacache [3] decide the slabs sizes in a cache to maximizes the hit frequency given the hit rate curves of slab size and cache eviction policy. In practice, hit rate curves can be approximated and fed into the optimization problem [4, 9, 10]. [7] provided a general formulation of the container deployment problem, in which containers are mapped and allocated to virtual machines, as an Integer Linear Programming problem and optimize multiple QoS metrics.

Cache Policy. Cache has been wisely used in computers to speed up the read access to the low-speed data on the main memory or hard disks. There are a large body of traditional cache eviction policies, e.g., FIFO (First In First Out), LIFO (Last In First Out), Least Recently Used (LRU), Most Recently Used (MRU), etc. These algorithms are heuristic so that they only work well in some specific applications. For example, FIFO expects that the oldest cache items will be out of date and it is the least likely being used in the future. Meanwhile, LRU is the improvement on FIFO because there is no cache items having the same value. These cache policies have been modified to have better performance. The study in [6] shows that we can modify LRU to improve Facebook photo caching.

3 WORKLOAD PREDICTION

For serverless computing, one of the key factors is the auto-scaling strategy.Now workload in a cloud environment may change frequently. It could be stable workload, growing workload, cycle/bursting workload, on-and-off workload, continuously changing workload and so on. Hence predicting the future workload using historical data will definitely have a positive impact on the resource allocation problem for growing, periodic and unpredicted workloads. It will enable resources to be pre-allocated for jobs to be submitted. [8] [2]

3.1 Approach

The objective is to predict the memory in GB for the incoming job requests.Underlying strategy is to train the prediction model offline and perform the prediction online. Detailed steps for building the model are as follows: (Refer to Figure 3)

- Gathered 2 million observations with 28 features as historical workload records from AuverGrid Data Traces. http://gwa.ewi.tudelft.nl/datasets[2]
- Analyze the distribution of data to understand the relationships between variables
- Feature Scaling and normalization. For eg, memory consumed was represented in KB in the original dataset, but it was converted to GB during training and testing as containers would always be spawned in GB. On a similar note, string values were represented in their corresponding numeric form as and when applicable.
- Extraction of key features for prediction using Variance threshold. Features with higher values of variance threshold were preserved, rest dropped.
- Data cleansing, separation of training and test datasets
- Applied K-fold cross-validation on the training set. Kept test data separate from cross validation(held out)
- Applied both classification and regression machine learning techniques for memory size prediction of incoming jobs
- Measured the bias and variance and find an optimal path
- Calculate the root mean squared error(RMSE) generated by cross-validation dataset and choose the parameters with least error.(tuned the hyperparameters)
- Use test data to determine the accuracy of prediction and choose the best method with lowest RMSE findings.

The goal was to leverage classification and regression techniques to find a good model to predict this future workload. This predicted workload can then be used by the resource calculation engine to compute the container allocation strategy.

3.2 Evaluation

Setup.

- Load 2 million observations with 28 features
- Data cleaning and feature extraction
- Apply classification and regression techniques and tune hyper-parameters to find a good predictive model.

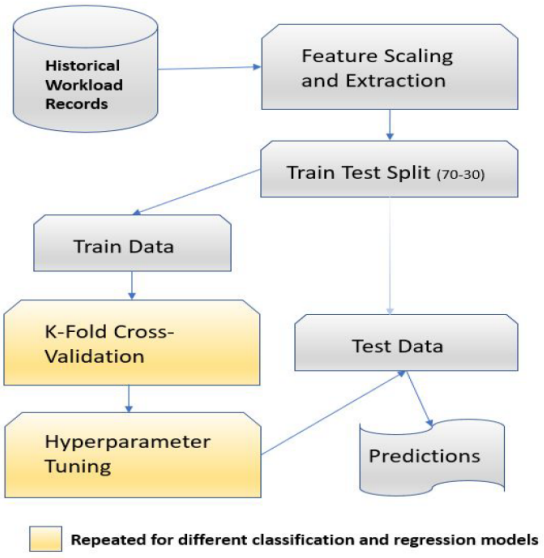

Figure 3: Overview of workload prediction approach

Raw Dataset : Observations - 2 million records, Features - 28				
Type	**Model**	**Best Hyperparameters after tuning**	**Training RMSE**	**Testing RMSE**
Classification	K-Nearest Neighbors	{'n_neighbors': 10, 'weights': 'distance'}	0.040019871	0.240112341
Regression	Linear Regression	LinearRegression(copy_X=True, fit_intercept=True , normalize=False)	0.40384744	0.402675052
Ensemble	Random Forest	{'min_samples_leaf': 1, 'min_samples_split': 2, 'max_depth': None, 'criterion': 'gini', 'max_features': 5, 'bootstrap': True}	0.174618631	0.21381556

Figure 4: Workload Prediction Results Summary

Baselines. The paper referred to [2] predicts the workload in terms of number of processors for the next 24 hours and not memory. This problem requires prediction of memory in GB for container allocation. Hence there is no direct baseline to compare against this prediction. However, the RMSE of 0.214 can be considered a good start with scope for optimization as future work.

Results.

- Three machine learning techniques were applied on 2 million records - Linear Regression(Regression), K-Nearest Neighbors(Classification) and Random Forest(Ensemble)
- Both training and testing RMSE were computed. (Refer to figure 4)
- Random Forest with tuned hyper parameters performed much better than the other techniques.

Future Work.

- Evaluate additional statistical, machine learning and deep learning models.
- Add derived features to improve prediction accuracy.
- Expand this solution to predict CPU usage and average running time of jobs.

4 RESOURCE CALCULATION ENGINE

In this section we came up with a resource calculation model that can make periodic decisions on container operations. The basic idea is that given the workload prediction and current system status, we calculate the best decision on pausing and stopping existing containers, and creating new ones, in order to minimize the total container preparation time.

In the first subsection, we describe the container scheduling model; the second subsection is mathematical formulation, in which we formulate the problem into an integer programming problem; and finally we show some evaluation results.

4.1 Container Scheduling Model

Specifically, our scheduler works like this:

Inputs

- **User demands**: number of jobs in the queue at time t; size and duration for each job. This information can be obtained from workload prediction, under the assumption that the prediction is perfect.
- **Current container status**: at time t, how many containers of what sizes are not occupied, and are available for reuse; how many free spaces are available.

Outputs

- **Container actions**: Which containers are reused; which containers are stopped; which containers are created.
- **Schedule**: mapping each job to a container.
- **Next container status**: what is the container status at time $t + t_0$.

4.2 Mathematical formulation

First we study the mapping between jobs and containers.

Definition 1. *A mapping between a set of jobs $J_i, i = 1 \cdots n$ and a set of containers $C_i, i = 1 \cdots n$ is said to be successful if it satisfies the following constraints: for $\forall J_i$, there $\exists C_{ji} \in \{C_i, \forall i = 1 \cdots n\}$, s.t. $J_i \leq C_{ji}$, and $\{C_{ji}, \forall i = 1 \cdots n\} = \{C_i, \forall i = 1 \cdots n\}$.*

WLOG, we suppose the two sets are both arranged in descending order, i.e. $J_m \geq J_n$ if $m \leq n$, $C_m \geq C_n$ if $m \leq n$.

Theorem 1. *A mapping between a set of jobs $\{J_i, i = 1 \cdots n\}$ and a set of containers $\{C_i, i = 1 \cdots n\}$ exists, if and only if $J_i \leq C_i$, for $\forall i = 1 \cdots n$.*

Proof. The backwards part is easy to prove. We now prove the forwards part.

For any pair of jobs J_m and J_n, $J_m \geq J_n$, and pair of containers of size C_p and C_q, $C_p \geq C_q$, a successful pairwise mapping means either (1) J_m maps to C_p, J_n maps to C_q ($J_m \leq C_p,\ J_n \leq C_q$), or (2) J_m maps to C_q, J_n maps to C_p ($J_m \leq C_q,\ J_n \leq C_p$).

Since $J_m \geq J_n$ and $C_p \geq C_q$, for situation (2) $J_m \leq C_q,\ J_n \leq C_p$, it is also true that $J_m \leq C_p,\ J_n \leq C_q$. In this way we can turn situation (2) to situation (1) by exchanging the mappings. Now for any given mapping between two sets, we do pairwise examination as above, until every pair with situation (2) is transformed to situation (1). In this way we can transform any successful mapping into J_i mapping C_i, for $\forall i = 1 \cdots n$. □

Now we can formulate the problem into an integer programming problem.

Inputs

- n: number of jobs in the queue at time t;
- $J_1 \geq J_2 \geq \cdots \geq J_n$: sizes of jobs;
- m: number of available containers at time t;
- $M_1 \cdots M_m$: sizes of available containers;
- M_0: available empty spaces at time t;

Variables

- $s_i, i = 1 \cdots m$: container i to stop if $s_i = 1$;
- $u_i, i = 1 \cdots m$: container i to unpause if $u_i = 1$;
- v: number of new containers created;
- $N_k, k = 1 \cdots v$: sizes of new containers;

Optimization

$$\min \quad \sum_{i=1}^{m} u_i \cdot t_{up} + \sum_{i=1}^{m} s_i \cdot t_{stop} + v \cdot t_{fs}$$

s.t. (1) $s_i \in \{0, 1\}, \forall i = 1 \cdots m$;

(2) $u_i \in \{0, 1\}, \forall i = 1 \cdots m$;

(3) $s_i + u_i = 1, \forall i = 1 \cdots m$;

(4) $v = n - \sum_{i=1}^{m} u_i$;

(5) $v \geq 0$;

(6) $J_1 \geq N_k \geq J_n, N_k$ integer, $\forall k = 1 \cdots v$;

(7) $max_p \{N_k, k = 1 \cdots v;\ u_i \cdot M_i, i = 1 \cdots m\} \geq J_p$, $\forall p = 1 \cdots n$;

(8) $\sum_{k=1}^{v} N_k \leq M_0 + \sum_{i=1}^{m} s_i \cdot M_i$.

Remarks

- t_{up}, t_{stop}, t_{fs} represent the time to pause(and unpause) a container, to stop a container, and to fresh start a container;
- (1) to (3): operations on existing containers;
- (4) to (6): creating new containers;
- (7): jobs fit in containers;
- (8): memory constraint.
- function max_p gets the p^{th} largest value in the set;

There are $2m + v + 1$ variables, where v is also a variable. We can modify it into a problem with m+n variables.

$$\min \quad \sum_{i=1}^{m} s_i \cdot (t_{stop} - t_{up}) + m \cdot t_{up} + \sum_{k=1}^{n} (N_k | 0) \cdot t_{fs}$$

s.t. (1) $s_i \in \{0, 1\}, \forall i = 1 \cdots m$;

(2) $0 \leq N_k \leq J_1, N_k$ integer, $\forall k = 1 \cdots n$;

(3) $max_p \{N_k, k = 1 \cdots n;\ u_i \cdot M_i, i = 1 \cdots m\} \geq J_p$, $\forall p = 1 \cdots n$;

(4) $\sum_{k=1}^{n} N_k \leq M_0 + \sum_{i=1}^{m} s_i \cdot M_i$.

We use the logical OR operation in objective function.

4.3 Evaluation

Since this problem is non-convex, non-linear, and can not be expressed in a standard form, we implement it in Matlab using exhaustive search. We show an example solution using this model.

Setup.

- $t_{stop} = 200ms; t_{fs} = 200ms; t_{up} = 1ms$.
- jobs in the queue: 7 jobs, sizes 9, 1, 7, 9, 8, 5, 3 GB.
- available containers: 4 containers, sizes 10, 2, 1, 4 GB.
- available spaces: 30 GB.

Results.

- Container 3 stop, other containers pause; create 4 new containers, size 5, 7, 8, 9 GB; minimized latency: 403 ms.
- Job set: $\{9, 9, 8, 7, 5, 3, 1\}$; Container set:$\{10, 9, 8, 7, 5, 4, 2\}$; the mapping is valid.

Future Work.

- Need to improve formulation to solve efficiently using a solver.
- Need to consider the influence of current decision at future time.
- Expand single resource (memory) in objective function to multiple resources (e.g., memory, CPU, energy consumption).

5 DYNAMIC CONTAINER SCHEDULER

In this section, we proposed a realistic approach that can be used in the dynamic scheduler. We assume that the distribution of demands can be estimated via the histogram of historical data. The approach can work in an online manner so that it can be well adapted to the system.

5.1 Approach

Our approach consists of 2 steps: container creation and contaner assignement.

Container Creation. Based on the histogram of demands, we prepare the containers in cluster. For each container size, the number of containers type i are created based on the number of bins h_i, the total historical jobs N, and the capacity of the cluster C. In particular, the number containers type n_i is computed as follows.

$$n_i = \frac{h_i \times C}{N}$$

Meaning, the number of containers are propotional the number of bins.

Container Assigment. When a new job arrives, it needs to be served. If there is an ready container that can accomudate the job, the schduler assigns the container to the job. If there is no container available, the system has to create the new container for the job by stopping other smaller containers.

5.2 Evaluation

Setup. It takes 0.2 seconds to create a new container while it takes only 0.001 secons to reuse a container from the pool. We replay 1831 jobs. The demand, job durations are found at http://gwa.ewi.tudelft.nl/datasets/. As the jobs are small, we set capacity at 1GB. The histogram of jobs are plotted on Figure 5.

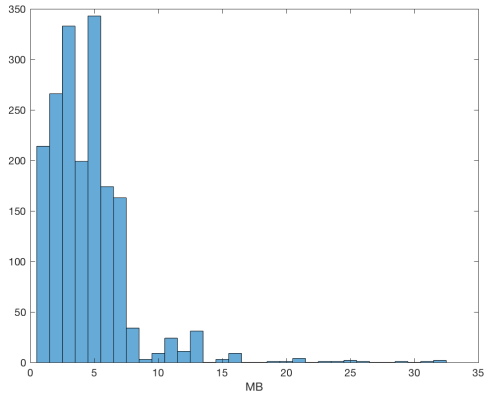

Figure 5: Histogram of 1831 jobs from http://gwa.ewi.tudelft.nl/datasets/.

Baselines. We compare our dynamic container scheduler with the following baselines:

- Worst: The system always creates the new container for the coming job.
- Naive: Each container is kept alive 100 secs after use. This approach is used in Amazon lambda.
- Ideal: We assume that there are available containers with any sizes for the new jobs.

Results. Our approach achieves 0.0013 secs which is very close to the ideal result (0.001 seconds). Meanwhile the Naive approach is far from the ideal performance. Naive only reachs 0.0897 seconds. The Worst approach takes 0.2 seconds to create a new container for any job.

Future Work. The simulation results show that the simple approach significantly improves the latency in creating new containers. We believe that this approach also help the real system. The future direction is to verify the proposed approach on Kubeless atop Kubernetes.

REFERENCES

[1] Nilton Bila, Paolo Dettori, Ali Kanso, Yuji Watanabe, and Alaa Youssef. 2017. Leveraging the Serverless Architecture for Securing Linux Containers. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*. IEEE, 401–404.

[2] Katja Cetinski and Juric Matjaz. 2014. Advanced model for efficient workload prediction in the cloud. AME, 1–11.

[3] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic Cloud Caching.. In *HotStorage*.

[4] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches.. In *NSDI*. 379–392.

[5] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with openlambda. *Elastic* 60 (2016), 80.

[6] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. 2013. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 167–181.

[7] Matteo Nardelli, Christoph Hochreiner, and Stefan Schulte. 2017. Elastic provisioning of virtual machines for container deployment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM, 5–10.

[8] Ali Nikravesh and Lung Chung-Horng Ajila. 2015. Towards an Autonomic Auto-Scaling Prediction System for Cloud Resource Provisioning. ACM, 1–11.

[9] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. 2014. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[10] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. 2015. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 174–181.