

E-516 Assignment

FaaS KeepAlive Competition

Overview

In this assignment, we have used a FaaS backend simulator to experiment with different container KeepAlive policies. This is being done by trying to identify the “best” containers to evict such that the cold start percentage is minimized.

Policies Implemented and Tested

The below table is an overview of the various policies tested (in the order that they were invented & implemented) and a brief explanation of the approach each one uses.

Please Note:

- Each policy’s implementation can be found from the “self.eviction_policy” value and associated eviction function in “code/sim/LambdaScheduler.py”
- README.md contains the changes implemented.
- The final selected policy that I am submitting is below this table along with the algorithm, explanation, and results. This policy is set as default for all the run script files (run_a_100.sh, etc.)
- I have also implemented the Dual Greedy algorithm from the FaaS-Cache research literature to compare the results of my algorithm with this one.

Sr. No.	Policy Name	Approach
1	CLOSEST_SIZE_LARGEST_KICK	Sort the available containers by mem_size. Binary search on the this array for the container with “mem_size” closest to the “to_free” size. Closest here is “equal to or next greatest”. In case “to_free” is greater than largest “mem_size”, evict the largest container.
2	CLOSEST_SIZE_SMALLEST_KICK	(Only implemented this for experimenting. Heuristic doesn’t make much sense). Same as above but if “to_free” is greater than largest “mem_size”, we start evicting from the smallest container.

3	LRU	Maintain a Least Recently Used cache (ordered dictionary in Python).
4	LFU_CLASSIC	<p>Update the Container object to include an invocation frequency value. This is set to 1 when container is initialized and incremented on each invocation.</p> <p>During eviction, the available containers are sorted by invocation frequency and the LFU container is chosen as the victim.</p>
5	LFU_GROUP_CLOSEST	<p>Similar to above but here we form an LFU group by sorting by freq.</p> <p>The group size can be a hyperparameter. In this case, group size is 10% of available containers list. More specifically decided as:</p> <pre>lfu_group_size = int(0.1 * len(available)) if len(available) > 40 else 4</pre> <p>From this 10% LFU group, we Binary Search to find the container with size closest to “to_free”.</p>
6	LFU_GROUP_MAX_COLD_TIME	Similar to LFU_GROUP_CLOSEST, but we choose the container from the LFU group with the max cold time (run time for cold start)
7	LFU_GROUP_MAX_INIT_TIME	<p>Similar to above LFU group approaches, but we choose the container from the LFU group with the max initialization time. For a container, this is calculated as:</p> <p>Init_time = cold_time – warm_time.</p> <p>This is the time taken by the container to set up the execution environment, dependencies. Hence, it is better to keep containers with a large “init_time”.</p>
8	LFUGROUP_MAXINITGROUP_CLOSEST	<p>Here, we build upon the above approaches and go one level further by using a “double grouping” approach. Approach:</p> <ul style="list-style-type: none"> • First find "n" LFU containers. • Sort the lfu_group by initialization time. • From the sorted lfu_group, find "m" (where m < n) containers with the least initialization time • This is our maxinit_group. Sort the maxinit_group by mem_size. • In the sorted maxinit_group, perform Binary Search to find closest mem_size container. • This container is chosen as the victim.

		<p>Size strategy for the 2 groups is 10% (outer group) & 5% (group) of available containers. Percentages allow us to scale the group sizes dynamically depending on available containers.</p> <ul style="list-style-type: none"> • $\text{lfu_group_size} = \text{int}(0.1 * \text{len}(\text{available}))$ if $\text{len}(\text{available}) > 40$ else 4 • $\text{maxinit_group_size} = \text{int}(0.05 * \text{len}(\text{available}))$ if $\text{len}(\text{available}) > 40$ else 2
9	LFUGROUP_CLOSESTGROUP_MAXINIT	<p>Similar to above but after finding our LFU group, our sub group is formed by finding the closest containers by mem_size to “to_free” first. We then sort this 2nd group by initialization time and choose the container with the least initialization time.</p>
10	LFUGROUP_MAXINITGROUP_CLOSEST	<p>Similar to LFUGROUP_MAXINITGROUP_CLOSEST, however here we choose the largest sized container as our victim and don't binary search for the closest sized container.</p>
11	DUAL_GREEDY_PRIORITY (implemented for comparison)	<p>This approach uses the formula mentioned in the FaaS-Cache research paper [1]:</p> $\text{Priority} = \text{clock} + [(\text{freq} * \text{cost}) / \text{size}]$ <p>(Here, cost was calculated as difference between cold_run_time and warm_run_time)</p> <p>Reference: Fuerst, Alexander, and Prateek Sharma. "FaasCache: keeping serverless computing alive with greedy-dual caching." In <i>Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems</i>, pp. 386-400. 2021.</p>

Selected Policy

The policy selected for submission is “LFUGROUP_MAXINITGROUP_CLOSEST”. This exact string value is used by “self.eviction_policy” in LambdaScheduler.py

Heuristic / Idea

- The idea is that frequency, initialization time and memory size are all important factors to consider while choosing a victim. LRU could be useful but I am not considering it in this policy as LFU is a much stronger factor than LRU for improving hit ration.
- We keep on reducing our list of available containers based on our 3 chosen factors (invoke_freq, init_time, and mem_size).
- The order in which we carry out this reduction matters as it in a sense giving weightage/importance to these factors.
Hence our importance order is: frequency > initialization_time > closest_size
- LFUGROUP_MAXINITGROUP_CLOSEST performed slightly better than LFUGROUP_MAXINITGROUP_LARGEST where we evict largest container from the last group instead of closest. However, evicting closest containers is a more “greedy” approach to optimizing hits. I did consider evicting largest to free up more space but I have decided to stick with the “greedy” closest container. The greedy approach will too evict the largest container if required size is greater than max_container size. (See binary search implementation below)

Algorithm

The algorithm uses a “double-grouping” approach in choosing a victim container. This approach is inspired from the previous approaches I attempted with classic LFU and other LFU grouping.

LFU group size is the bottom 10% of all containers sorted by invocation frequency.

Initialization_time sub-group size is bottom 50% of LFU group size.

Below container pool size of 40, these values are 4 and 2 respectively.

Steps:

1. Find bottom “n” LFU containers. This is the LFU group
2. Sort the LFU group by initialization time. $\text{init_time} = \text{cold_time} - \text{warm_time}$
3. From sorted LFU group, find bottom “m” containers w.r.t. init_time. (here, $m < n$)
This is the maxinit_group.
4. Sort the maxinit_group by memory size of containers.
5. Perform binary search on this sub-group to find the container with size that is “equal or next greatest” and choose it as victim. If required size is more than max_container, then max_size_container is chosen as victim.

```
# ----- Helpers -----

def binary_search_closest(self, arr, key):
    start = 0
    end = len(arr) - 1
    while start < end:
        mid = start + (end - start) // 2
        if key > arr[mid].metadata.mem_size:
            start = mid + 1
        else:
            end = mid
    return end
```

```
def evict_lfu_group_maxinitgroup_closest(self, to_free):
    """
    Approach:
    Double-sorted-group approach:
    - First find "n" LFU containers.
    - Sort the lfu_group by initialization time.
    - From the sorted lfu_group, find "m" (where m < n) containers with the least initialization time
    - This is our maxinit_group. Sort the maxinit_group by mem_size.
    - In the sorted maxinit_group, perform Binary Search to find closest mem_size container.
    - This container is chosen as the victim.
    """

    eviction_list = []
    available = [c for c in self.ContainerPool if c not in self.RunningC]

    # Note: Sorting in ascending invoke_freq here, least invoked is at the start
    available.sort(key=lambda c: c.invoke_freq)

    lfu_group_size = int(0.1 * len(available)) if len(available) > 40 else 4
    maxinit_group_size = int(0.05 * len(available)) if len(available) > 40 else 2

    while to_free > 0 and len(available) > 0:
        lfu_group = available[:lfu_group_size]

        # ASCENDING sort by initialization time
        lfu_group.sort(key=lambda c: c.metadata.run_time - c.metadata.warm_time)

        maxinit_group = lfu_group[:maxinit_group_size]

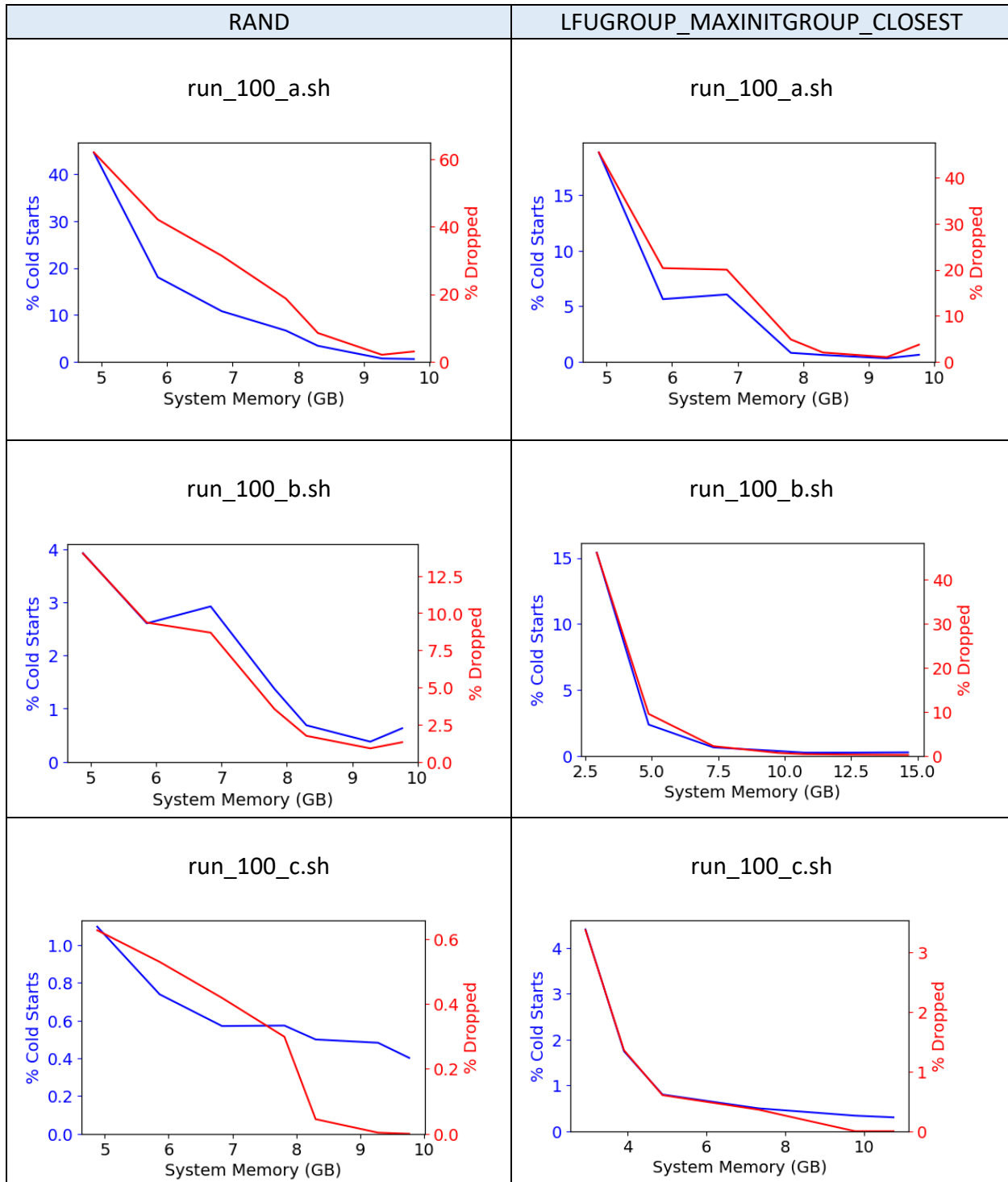
        # ASCENDING sort by mem_size
        maxinit_group.sort(key=lambda c: c.metadata.mem_size)

        # Binary search for the closest sized container from this sorted sub-group
        victim_index = self.binary_search_closest(maxinit_group, to_free)
        victim = maxinit_group[victim_index]
        # victim = maxinit_group[-1] # get largest container

        available.remove(victim)
        eviction_list.append(victim)
        to_free -= victim.metadata.mem_size

    return eviction_list
```

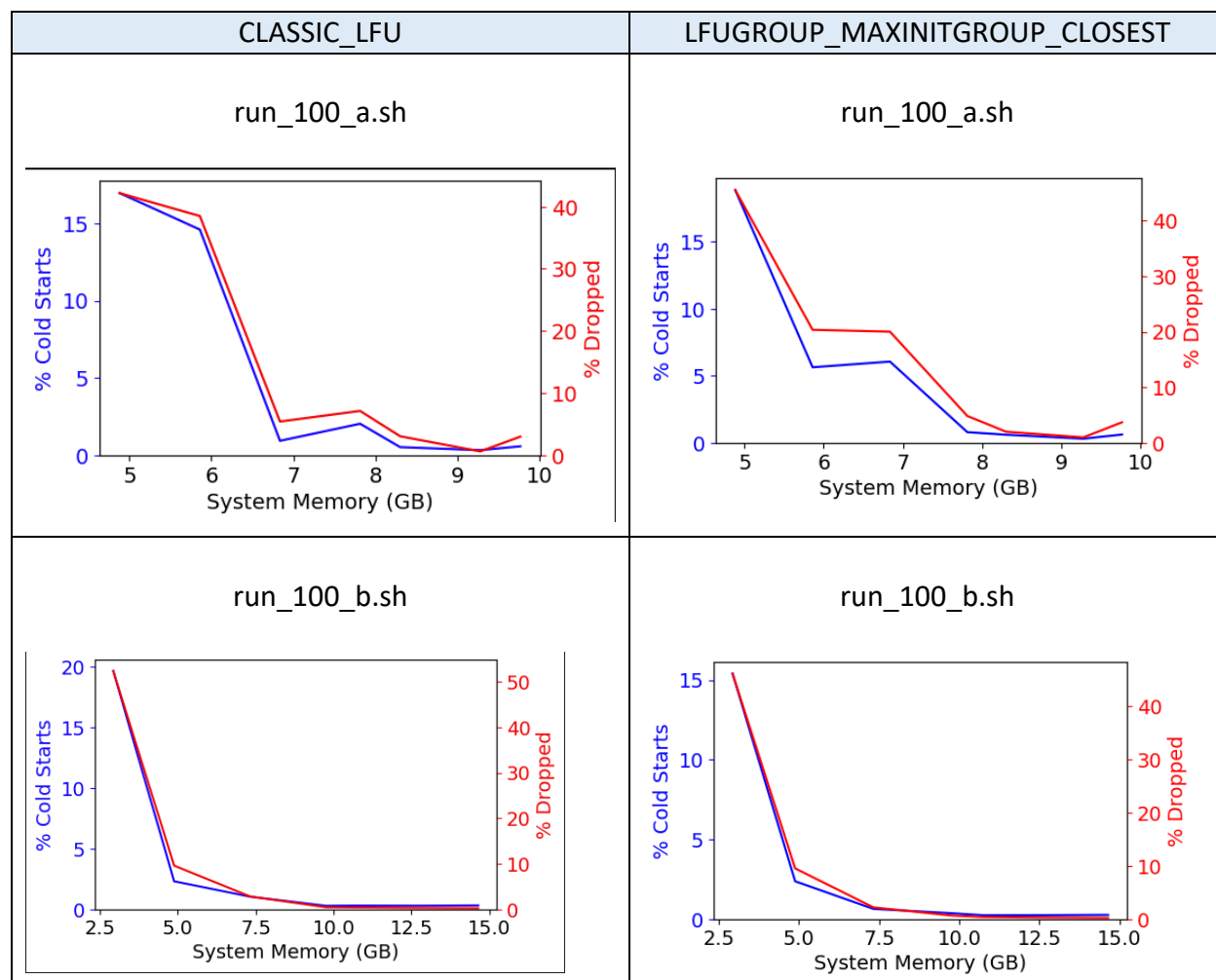
Performance Comparison with RAND

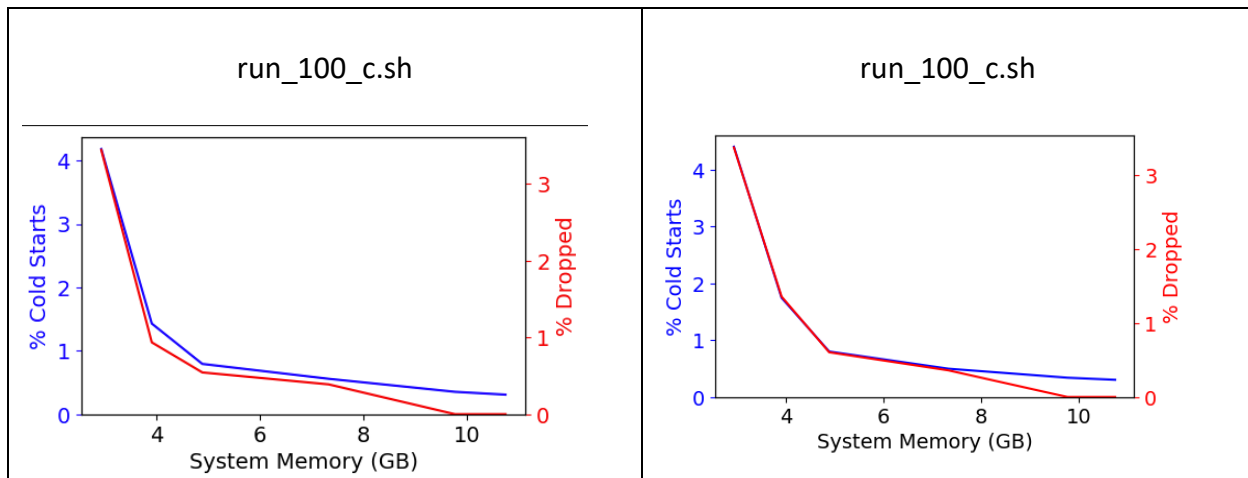


Other Interesting Results

CLASSIC_LFU vs LFUGROUP_MAXINITGROUP_CLOSEST

One interesting result was that CLASSIC_LFU (evict least frequent directly) performed slightly better than LFUGROUP_MAXINITGROUP_CLOSEST. However, this could be due for the current test sets only. Classic LFU doesn't consider initialization time or memory size at all, and hence I have not chosen it as a generic solution.





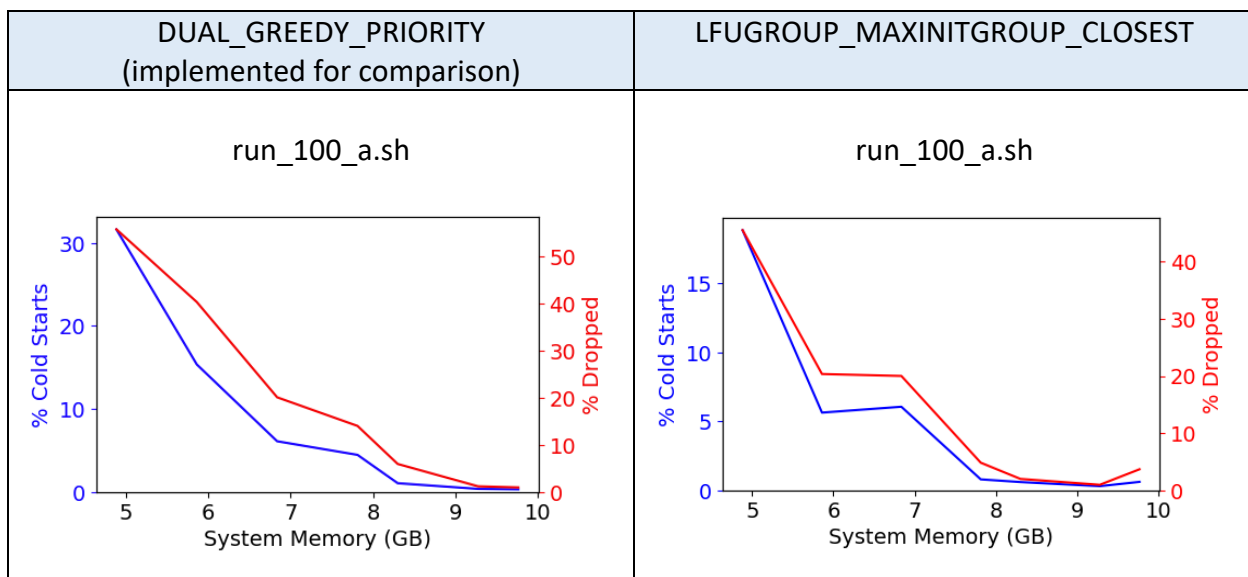
DUAL_GREEDY_PRIORITY vs LFUGROUP_MAXINITGROUP_CLOSEST

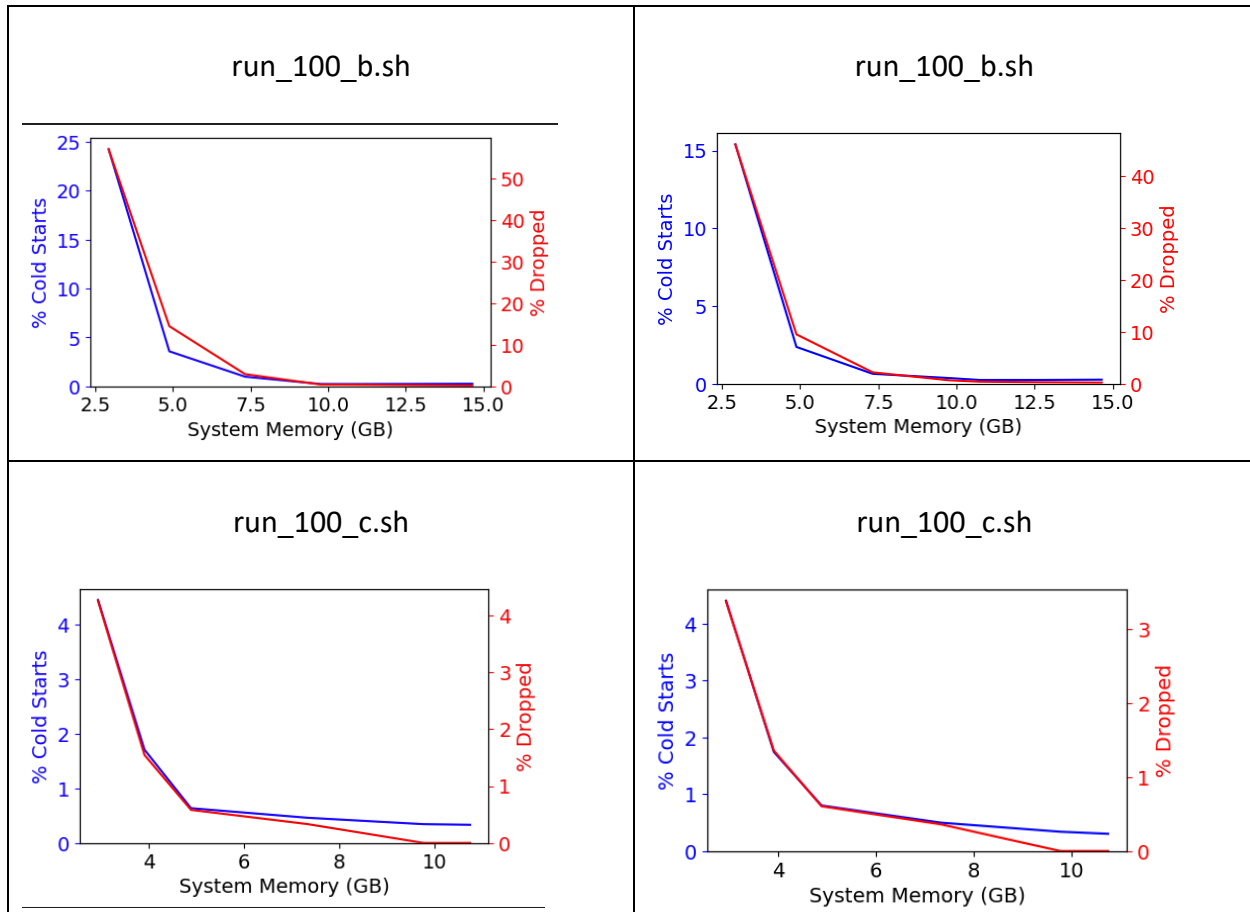
DUAL_GREEDY_PRIORITY is the algorithm implemented from the research paper [1] on FaaSCache. For the a, b, & c traces for 100 functions, the 2 algorithms resulted in similar graphs however this could be again not very generic. The DUAL_GREEDY_PRIORITY also takes into consideration “LRU” using “clock time” in its formula. It also makes priority inversely proportional to the container size. Thus, it is favoring to evict “large” containers. On the other hand, my policy tries to evict “closest” (or “equal or next_in_size”) containers first and if not, it will evict the largest one.

Choosing between largest sized container and closest sized container

Choosing the largest sized container from the sub-group frees up more space.

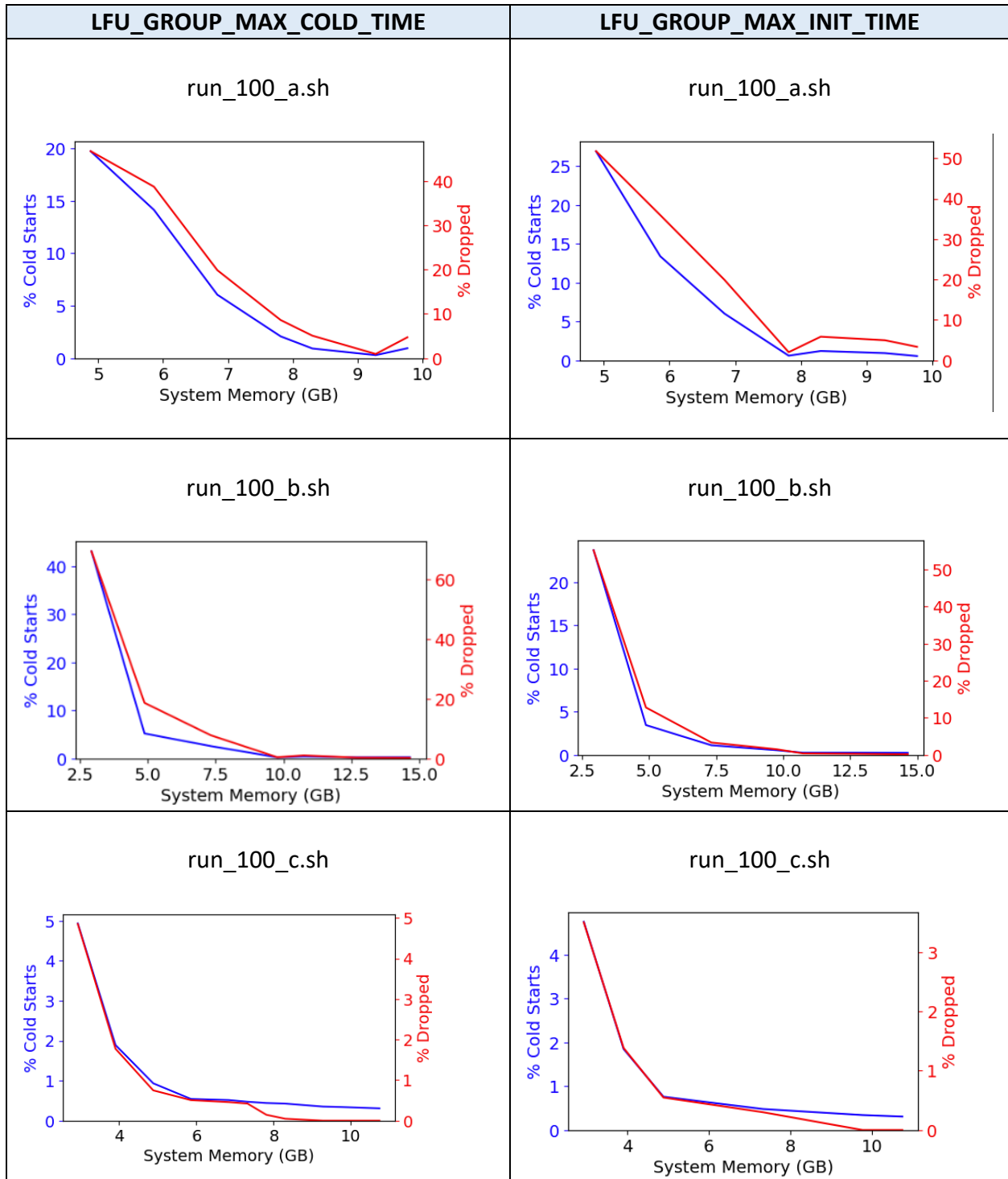
However, choosing the closest in size (equal or higher) is a more “greedy” approach in fitting our new container. To know for sure if largest is better, comparison with more offline data results might help.





LFU_GROUP_MAX_COLD_TIME vs LFU_GROUP_MAX_INIT_TIME

I have also compared eviction policies where we first form the LFU group and then evict by “cold run time” vs “initialization time”. The cold run time includes the warm run time of a function and the idea was to see if containers with long “warm run times” should also be considered for removal instead of just “initialization times”.



Conclusion & Future Experiments



For this assignment, the performance measurement criteria relied on the test set provided to us. Comparing eviction policies allowed us to understand FaaS-based container eviction more deeply. For future work, it would be interesting to generate an automation script to compare all policies together and see their performance on a single graph. As the assignment suggests, ML-based approaches could perform significantly better than heuristics mentioned in this report as essentially, this is a statistical minimization problem.

References

[1] Fuerst, Alexander, and Prateek Sharma. "FaasCache: keeping serverless computing alive with greedy-dual caching." In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 386-400. 2021.

Appendix

Changes made in the original codebase

 README.md 

Readme

Nirav Raje - Changes Made in the Original Codebase

- In LambdaScheduler.py, added eviction policies by setting `self.eviction_policy` in the constructor of LambdaScheduler
- In LambdaScheduler.py in the runInvocation() method, added the code to increment frequency on invocation for LFU-based policies, add to LRU cache for the LRU policy, priority calculation for DUAL_GREEDY approach.
- In Container.py, added `invoke_freq`, `init_time` and `priority` for the Container object:

```
class Container:

    state = "COLD"

    def __init__(self, lamdata: LambdaData):
        self.metadata = lamdata
        self.invoke_freq = 1
        self.priority = 0
        self.init_time = lamdata.run_time - lamdata.warm_time
```
- Below the RandomEvictionPicker function, there is a list of functions added as per the `self.eviction_policy` set containing all policy functions.
- The example scripts `run_all_20.sh`, `run_all_50.sh`, `run_all_100.sh` make it convenient to run a, b, and c traces for 20, 50 and 100 functions together.