MomentAllocator - Cache moments based allocator

Amir Dachbash

October 2021

1 Introduction

An increasing number of enterprise applications have migrated to hosted platforms in private enterprise and public cloud data centers. Such platforms are typically virtualized, i.e., tenants deploy applications in virtual machines (VMs) whose access to the underlying resources (memory, storage,network) is shared with other tenants, and mediated by hypervisors such as Hyper-V, VMware ESX, or Xen.
Uninhibited sharing of such resources in a multi-tenant environment leads to poor and variable application performance.
While recent efforts give providers control over how resources like network and storage are shared, there is no coordinated end-to-end control of the distributed caching infrastructure, made up of storage caches at multiple places along the IO stack (inside VMs, hypervisors, storage servers).
Today, operators statically divide memory across applications. For example, Facebook, which manages its own data centers and cache clusters [1, 2], has an engineer that is tasked to manually partition machines into separate cache pools for isolation. Similarly, Memcachier [3, 4], a cache-as-a-service for hundreds of tenants, requires customers to purchase a fixed amount of memory.
Static partitioning is inefficient, especially under changing application loads; some applications habitually under utilize their memory while others are short of resources. Worse, it is difficult for cache operators to decide how much memory should be allocated to each application. This manual partitioning requires constant tuning over time. Ideally, a web cache should automatically learn and assign the optimal memory partitions for each application based on their changing working sets; if an application needs a short term boost in cache capacity, it should be able to borrow memory from one that needs it less, without any human intervention.
To this end, I designed MomentAllocator, a cache allocation manager in a multi-tenant environment.
MomentAllocator intended to run as a service per application (can be extended to manage multiple application) in a multi environment and monitor the requests it gets, its goal is to advise on a time-based updated cache size which we should allocate to the application, it monitor the requests continually and publish it's advisement each short period. MomentAllocator requires very low resources and uses streaming moment-based algorithms in order to revalue the distribution of the requests in a real time manner.

1.1 Frequency moments

Consider a stream $S = \{a_1, a_2, ..a_m\}$ with elements from a domain $D = \{v_1, v_2, .., v_n\}$, Let m_i denote the frequency of value $v_i \in D$ i.e., the number of times v_i appears in S. The k^{th} frequency moment of the stream is defined as:

$$F_k = \sum_{i=1}^{n} m_i^k \tag{1}$$

We will use two moments in order to estimating the distribution of the requests.

1.2 F_0 moment

The F_0 moment estimates the number of distinct elements which arrived so far in the stream. The following is the algorithm by Alon, Matias, Szegedy in 1996[5], to estimate F_0.

1. Choose a random hash function h from a pairwise independent hash family.
2. For each item x of the stream, update z as z = Max(zeroes(h(x)), z), where zeroes(y) denotes the number of trailing zeroes of y in the binary representation.
3. Output 2^{z+c} for c = 1/2.

As we can see above, the F_0 estimator uses a constant memory and is very fast and lightweight (mainly depending on the hash function performance which today should be very fast, notably when today's processors support hardware acceleration of Secure Hash Algorithm (SHA) family).

1.3 F_2 moment

F_2 is interpreted as measure of homogeneity, namely, if some symbols in P are more likely than others, F_2 will be able to trace that. There is an elegant and efficient way (in terms of memory) to estimate F_2, which is called the Tug-of-War algorithm.

1. Pick a random hash function from some hash family H mapping $U \to \{-1, +1\}$.
2. Maintain counter C, which starts off at zero.
3. On update $(add, i) \in U$, increment the counter C $\to$ C + h(i).
4. On query about the value of F_2, reply with C^2

In this research, we have implemented these moments from scratch in order to be able to tweak them according to our needs.

2 Related works

There has been much work recently on caches in data centers and the cloud, but not on dividing a fixed global cache to number of applications in a global optimal way. However, Cliffhanger [6] designed to predict how beneficial will be to extend the cache using a shadow queue. A shadow queue is a simulation of another memory of the cache but instead of holding the key and the value it contains only the keys which evicted from the cache by the replacement policy, but if the cache was larger those keys were in it. The algorithm collect statistical information about the simulated hits of the shadow queue and increases or decreases the cache size. Although its impressive results, Cliffhangers performance depends on the shadow queue's size which may consumes many resources in order to predict the demand for a larger cache or it will provide bad predictions. Moreover, an algorithm that knows how to divide the memory between several applications is not presented in the article.

3 Motivation

Resources allocation in a multi-tenant environment is challenging, using limited resources we need to satisfy the requirements of all the applications in the environment and give it the feeling it runs alone with all the resources its needed. Dividing the resources between the application should be done carefully and responsive to the applications need.

3.1 Requests distribution

In order to predict the need of an application for a cache storage we estimate the distribution of the requests from the application, to understand how the distribution affect the cache utility lets examine two applications with different request distribution:
The requests from application A distribute normally with p = 0.5 as shown in figure 1, the second application B receives requests which distribute normally with p = 0.2 as shown in figure 2.
Now, lets examine the following cost model.

Definitions
Let the workset of our applications (which means all the requests the applications may receive) defined as: $R = r_0, r1, .., r_n$.
Let the frequencies of the request defined as $F = f_0, f1, .., f_n$.
Let the value of fetching a block from the cache defined as v=10 for all blocks, we define the value as the time units it saves us by satisfy the request without reaching to the disk.
Case A: cache size = 100
Calculating the value of the cache for application A we get score of: 78353.86416881322
Calculating the value of the cache for application A we get score of: 59687.884459068504
Case B: cache size = 500
Calculating the value of the cache for application A we get score of: 271453.45122841065
Calculating the value of the cache for application A we get score of: 280835.77052624506
Note: The python script with all the calculation above and creation of the distributions Appears in the appendix A.
From these calculations we can conclude that adding more cache to an application which benefit the most with X size won't be the best choice necessarily. Moreover we can see the strong correlation of requests distribution and the need for a specific cache size. Clearly, over time the distribution may change so having a service that monitor it and assign dynamically cache to applications in a multi-tenant environment can result in better utilization of the cache resource while giving the applications the resource they need to have a good performance. MomentAllocator optimize the global performance of the applications which share the same environment and not necessarily the performance of a single application.

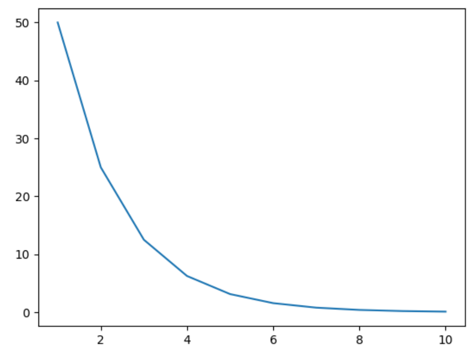

Figure 1: p=0.5

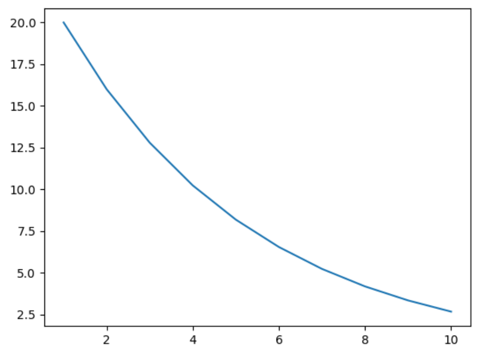

Figure 4: p=0.2

3.2 Blocks uniqueness

Another parameter that may affect the need for a cache is how many unique blocks were requested. Intuitively, the more unique blocks that were requested in a specific time slot the less biased the distribution was. So, tracking the number of unique blocks which were requested might help us estimating the distribution in a more accurate way.

In conclusion, estimating the requests distribution allow us to partition the cache in an optimal way at the machine view. Of course, this estimation must be very lightweight, responsive and consume very limited resource in cpu-time manner as well as memory manner. To accomplish that introduce an algorithm which based on frequency moments [5] as described above.

Briefly explain on modern mechanism of storage partition The optimized size of cache for an application is dynamically changing over time and must be tuned.

4 Algorithm

As mentioned above, in this research we focus in two moment F_0 and F_2. Our main goal is to process the raw values of F_0 and F_2 to a single result $R \in [0, 1]$ which indicates how skewed or uniform the distribution is, then multiplying it with the maximum cache size to get the suggested cache size of the next time slot. The skewer the distribution the smaller R value which results in a smaller suggested cache size, the more uniform the distribution is the larger R value is which results in a larger suggested cache size.
Our intention is to use these moments in a streaming fashion as described in algorithm1 and combine their results each time we want to output a suggested cache size, then restart the moments and start again.

Algorithm 1 Streaming algorithm

```
Input: Request key
1: $F_0$.add(key)
2: $F_2$.add(key)
```

Since the output of F_0 is an estimation of the number of distinct numbers in a time slot and the output of F_2 is an estimation of the likelihood to see some requests over others ($\in [0, number\_of\_requests]$), we need to process them and normalize them in order to get the output in the correct form which means a cache size between 0 and maximum cache size.
As described in algorithm2, we are getting the raw output from F_0 and F_2, then normalizing each output to $F_x \in [0, 1]$ as follows:
F_0: dividing F_0 raw value by the slot's length then pass it to a Sigmoid function.
F_2: dividing F_2 raw value by the squared slot's length, multiplying it by 1000 and then subtracting it from 1. The last subtraction is because of the contrasting correlation between F_2 and the need of cache storage, the bigger the value of F_2 the skewer the distribution and lesser the need of cache storage.
Note: The forms of the normalization were chosen empirically after testing a couple of forms. After normalizing the values and reducing them to [0,1] we are combining them using weighting parameters α and β to one final parameter in [0,1], this parameter indicates the how skewed the distribution is and according to the explanation from the previous sections how much cache storage the app needs. Note: The weighting values α and β of these parameters was assigned according to empirical tests, of course $\alpha + \beta = 1$ exists. The next step would be to multiply the maximum allowed cache size by the resulted parameter and get the current suggested cache size according to the last time slot. The last step would be to use exponential backoff in order to take into account the last time slots, such that the older the result the less it affects the current result.
In order to avoid basing our solution on a short term phenomenal in the requests distribution we use exponential back off to base our suggestion for cache size on the last time slot and previous suggestions, as follows:

1. $final\_cache\_size = \gamma * last\_cache\_suggestion + (1 - \gamma) * accumulated\_previous\_results$

2. $accumulated\_previous\_results = final\_cache\_size$

Algorithm 2 Suggest cache size algorithm

```
Output: Suggested cache size
1: $f_0\_raw = f_0.get\_distinct\_items\_count()$
2: $f_2\_raw = f_2.get\_homogeneity\_measurement()$
3: $f_0\_norm = 1/(1 + e^{-1*(f_0\_raw/slot\_length)})$
4: $f_2\_norm = 1 - 1000 * (f_2\_raw/slot\_length^2)$
5: $aggregated\_indicator = \alpha * f_0\_norm + \beta * f_2\_norm$
6: $current\_cache\_size = maximum\_cache\_size * aggregated\_indicator$
7: $final\_cache\_size = current\_weight * current\_cache\_size + (1 - current\_weight) * aggregated\_old\_results$
8: $aggregated\_old\_results = final\_cache\_size$
9: return $final\_cache\_size$
```

5 An improved algorithm

According to our tests, the weak spot of our algorithm was that the raw result of F_0 has not always close enough to the real distinct count of the requests, this is because it's strongly depends on the chosen hash function which should distribute the values uniformly.
To overcome this problem, we implemented an improved version of F_0. In this version we are executing in parallel a couple of F_0 estimators, when we need the estimation of F_0 output we aggregate all the results from all the estimators by averaging them as written in algorithm 3 and 4.

Algorithm 3 Aggregated streaming algorithm

```
Input: Request key
ForEach $est \in F_0\_estimators()$ do
    $est.add(key)$
endfor
```

Algorithm 4 Aggregated estimation of distinct number

```
Output: Estimated distinct requests
ForEach $est \in F_0\_estimators()$ do
    $sum\_of\_estimation = f_0.get\_distinct\_items\_count()$
endfor
return $sum\_of\_estimation/|\{f_0\_estimators\}|$
```

6 Implementation

Our system is built from 3 parts:

1. **Replacement policy** - this part includes two algorithms:
   (a) Eviction algorithm - choosing a candidate to be evicted from the cache.
   (b) Admission algorithm - Comparing the new candidate for insertion with the output of the eviction algorithm according to some heuristic and decide which one of the to admit to the cache.

This module also gets the suggested cache size from the Cache size tuner module (described below), and force this size for the next time slot as follows:

(a) In case the suggested cache size is smaller then the current size, this modules evicts blocks using the eviction policy until cache size equal to the suggested size.

(b) In case the suggested size if bigger then the current size, this module just extend the cache and allows more blocks to be admitted to the cache.

In our system we implemented two state of the art replacement algorithm which is very common on many systems: LRU and LFU.

2. **Cache size tuner** - this is an implementation of the algorithms we are discussing here. This is the most important part of the system in our context.
   This module includes a couple of algorithms that try to predict the need of the application's need for cache as described in the previous section. This module collects statistical data constantly and every time slot outputs a suggested size for the next time slot.

3. **Requests generator** - this module generates requests synthetically which distributes according to Zipf's distribution [7], we generate varied forms of the distribution using the skew parameter which determines how narrow the distribution is, from very narrow geometric distribution as demonstrated in figure 3 to almost uniform distribution as demonstrated in figure 4.

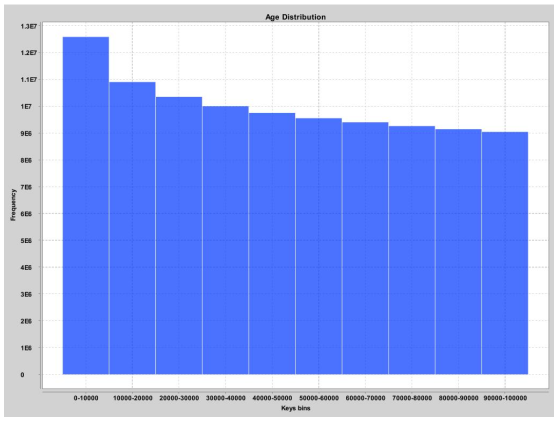

Figure 3: skew=0.1

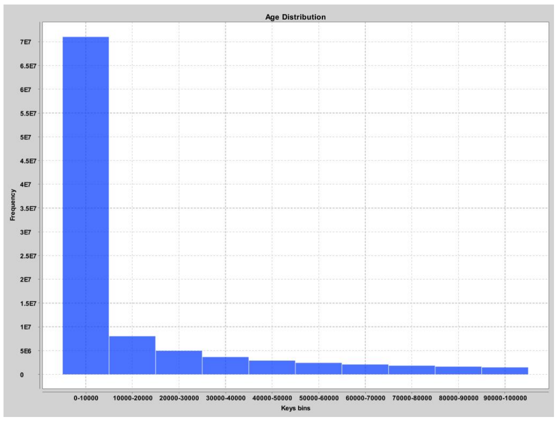

Figure 4: skew=0.9

7 Experiments

For our experiments we introduce the most important parameters:

1. t - constant, in our experiments we define it to be **10,000,000**.
2. w - size of work set (range of values for request id).
3. s - skew factor (of Zipf distribution).
4. cache_size - constant, in our experiments we define it to be **4096**.
5. phase_length - constant, in our experiments we define it to be **20,000**. This parameter defines the number of requests to process before executing the cache size estimator.

We use 2 replacement policies, LRU and LFU.

7.1 Experiment 1

Parameters:w=100,000, s=0.1.

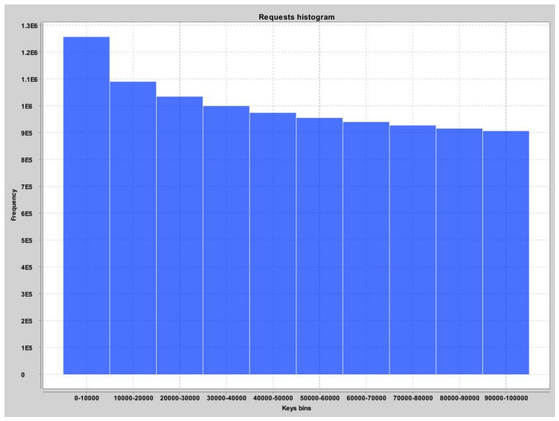

Figure 5: requests distribution plot.

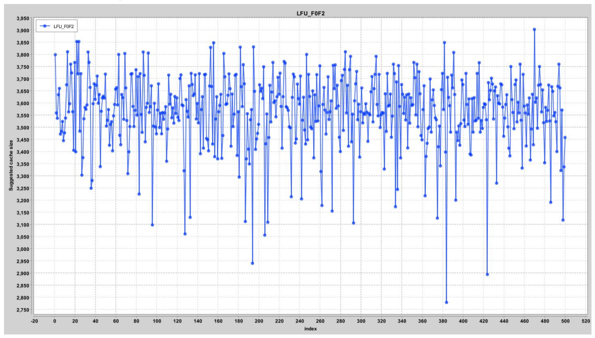

Figure 6: LFU with MomentAllocator cache allocation.

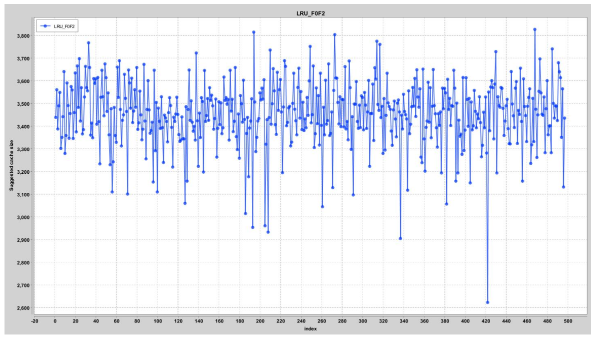

Figure 7: LRU with MomentAllocator cache allocation.

Cache manager	Hit ratio
LRU with constant size 4096	0.0492237
LRU with MomentAllocator	0.0378381
LRU with constant size 2048	0.0260962
LFU with constant size 4096	0.0513068
LFU with MomentAllocator	0.045197
LFU with constant size 2048	0.0273318

Table 1: Variable Descriptions

7.1.1 Results

We can see that since the distribution is almost uniform, MomentAllocator identifies it and suggests to use almost all the available cache. Moments allocator's result is slightly less good then the maximum size policy specially with the LFU policy but it using less space then it, on the other hand its result is far better then the policies with the half size cache.
To sum up, MomentsAllocator successfully identifies the requests distribution as almost uniform and therefore allocates almost all the cache storage for the application.

7.2 Experiment 2

Parameters:w=100,000, s=0.4.

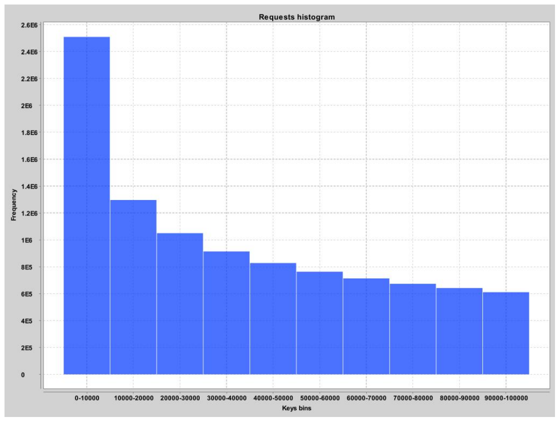

Figure 5: requests distribution plot.

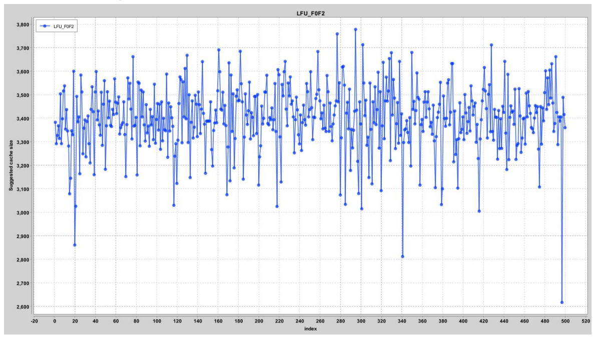

Figure 6: LFU with MomentAllocator cache allocation.

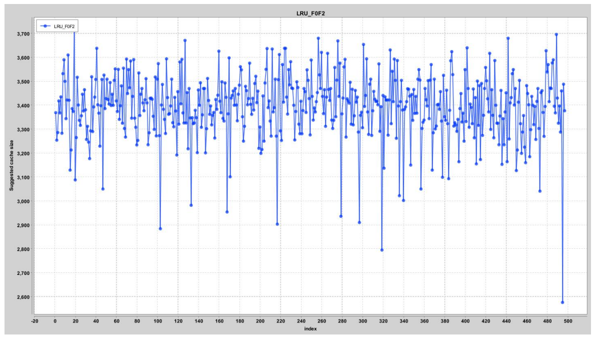

Figure 7: LRU with MomentAllocator cache allocation.

Cache manager	Hit ratio
LRU with constant size 4096	0.1414161
LRU with MomentAllocator	0.1071172
LRU with constant size 2048	0.0937533
LFU with constant size 4096	0.1437861
LFU with MomentAllocator	0.1285006
LFU with constant size 2048	0.0950485

Table 2: Variable Descriptions

7.2.1 Results

We can see that since the distribution is less uniform, still except the first interval which is very popular there are many requests which distribute equally in other intervals, MomentAllocator still suggests to use most of the available cache.
Like the first experiment Moments allocator's result is very close to the maximum size policy and again worked better with the LFU policy. To sum up, MomentsAllocator maybe suggests to allocate more storage then needed but still shows good results relative to the maximum and half size cache policies.

7.3 Experiment 3

Parameters:w=100,000, s=0.9.

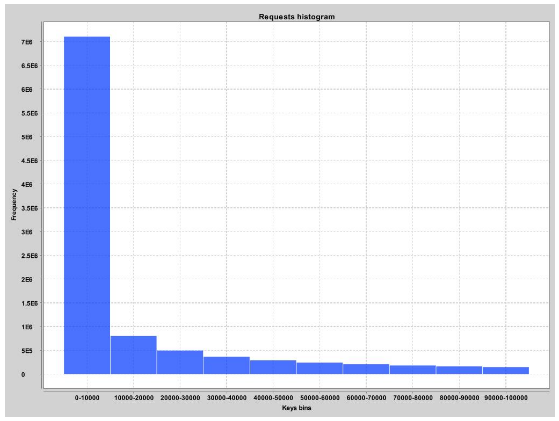

Figure 5: requests distribution plot.

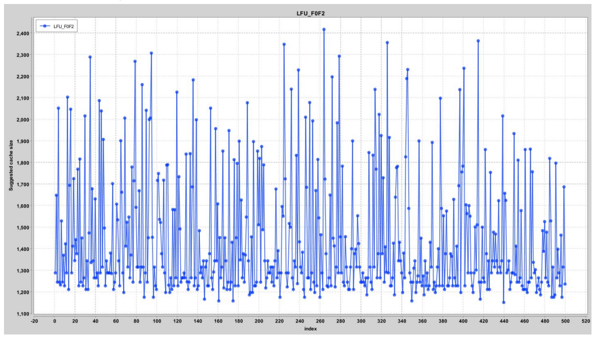

Figure 6: LFU with MomentAllocator cache allocation.

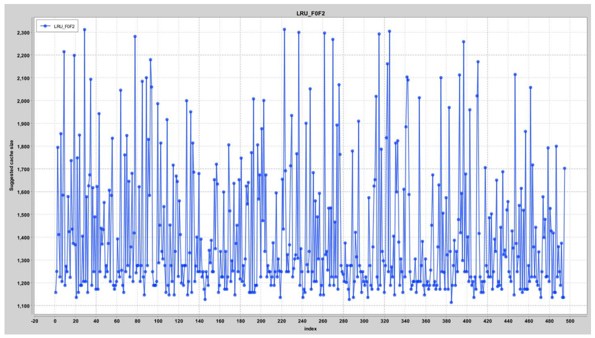

Figure 7: LRU with MomentAllocator cache allocation.

Cache manager	Hit ratio
LRU with constant size 4096	0.6118032
LRU with MomentAllocator	0.4820915
LRU with constant size 2048	0.5447707
LFU with constant size 4096	0.6130862
LFU with MomentAllocator	0.5093416
LFU with constant size 2048	0.5453452

Table 3: Variable Descriptions

7.3.1 Results

We can see that in this case the distribution is very skewed and not uniform at all, almost all the requests directed to the first interval and the other requests are pretty much neglected. Moments allocator estimated this distribution and respond by dramatically reduce the suggested cache size even below half of the maximum available storage.
We can see that most of the time Moments allocator suggested cache in the size of around 1300 blocks only, far less the 2048 of the half size policy.
In this experiment, Moments allocator got the least result but lets look at the numbers: Its hit ratio is about 10% less than the maximum size policy but most of the time uses about 68% less them memory then it. Its

hit ratio is about 4.5% then the half size policy but most of the time uses 36.5% less memory then it.
To sum up, MomentsAllocator took a dramatically step toward reducing the cache size but showed a good result in a very limited cache size.

8 Conclusions

Reviewing the experiments we can see that Moments allocator introduces a good trade off between space and hit ratio, its rank of action is still pretty rough and need more refinement but overall it shows good hit ratio with great space saving.
In most of the experiments Moments allocator's results were between the maximum space policy and the half one except the experiment with the very skewed distribution which Moments allocator resulted in the least hit ratio but saved a massive use of space.

9 Future work

A natural direction of future work would be to optimize the weight parameters of F_0 and F_2 to better express the accuracy of the estimators and to use more advanced algorithm to estimate the moments (i.e HyperLogLog for F_0).
Another direction would be to use more moments to estimate the requests distribution and to weight them in the estimation equation. Refinement of the steps taken by Moments allocator is another place for improvement, as we saw in the experiments Moments allocator tend to take extreme step when the distribution is skew or to take too gentle step when the distribution is just a little skewed.
Lastly, generalize Moments allocator to work with multiple applications and compare their distributions in order to take better decision in the machine (multi tenant environment) level.

References

[1] Berk Atikoglu Stanford, Berk Atikoglu, Stanford, Yuehai Xu Wayne State University, Yuehai Xu, Wayne State University, Eitan Frachtenberg Facebook, Eitan Frachtenberg, Facebook, Song Jiang Wayne State University, and et al. Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review*, Jun 2012.

[2] By: Rajesh Nishtala. Scaling memcache at facebook. *Facebook Research*, Dec 2016.

[3] Memcachier. www.memcachier.com.

[4] dynacache: dynamic cloud caching - mit csail.

[5] Mario Szegedy Noga Alon, Yossi Matias. The space complexity of approximating the frequency moments. 1999.

[6] Mohammad Alizadeh Asaf Cidon1, Assaf Eisenman and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. 2016.

[7] Zurab Silagadze. Citations and the zipf-mandelbrot's law. *Complex Systems*, 11, 02 1999.

10 Appendix A

Python script for plotting 2 different distributions to illustrate the effect of requests distribution over the need for cache storage:

```python
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats

value_of_cache_hit = 10
# First graph
mu = 0
std = 300
snd = stats.norm(mu, std)
x1 = np.linspace(0, 1000, 1000)
y1 = snd.pdf(x1) * 1000 * 60
plt.plot(x1, y1)

# Second graph
mu = 0
std = 800
snd = stats.norm(mu, std)
x2 = np.linspace(0, 1000, 1000)
y2 = snd.pdf(x2) * 1000 * 60 * 2
plt.plot(x2, y2, 'red')

plt.axis([0, 1000, 0, 100])
plt.show()

app1_value_with_size_100 = sum(y1[:100]) * value_of_cache_hit
app2_value_with_size_100 = sum(y2[:100]) * value_of_cache_hit
app1_value_with_size_500 = sum(y1[:500]) * value_of_cache_hit
app2_value_with_size_500 = sum(y2[:500]) * value_of_cache_hit

print("app1 with cache size 100 will result in " + str(app1_value_with_size_100))
print("app2 with cache size 100 will result in " + str(app2_value_with_size_100))
print("app1 with cache size 500 will result in " + str(app1_value_with_size_500))
print("app2 with cache size 500 will result in " + str(app2_value_with_size_500))
```