Ozone EC Pipeline/Container Allocation

Ozone EC feature is a new IO path for Ozone apart from Ratis replicated IO. While Ratis Write uses Ratis pipelines which are based on underlying RaftGroups formed by DNs, EC Write doesn't use heavy things like RaftGroups to manage data flow, an EC pipeline is just a bunch of DNs recorded as a group in SCM DB. Moreover we normally have multiple open containers for a Ratis pipeline since the pipeline is heavy and the creation and closing operations are expensive. But we have only one open container per EC pipeline since the creation and closing of EC pipelines are just DB record operations on the SCM DB along with async close commands sent to DNs.
So we'd better have a new allocation strategy for EC pipelines/containers.

Basic Principles

After some discussions under the issue:
https://issues.apache.org/jira/browse/HDDS-5327
We've got several **principles** to follow:

1. There should always be multiple pipelines in a normal cluster so as to direct writes to different containers, then load could be spreaded and performance should benefit.
2. There should be a limit of the max number of open pipelines, so we don't overwhelm the cluster into an unusable state.
3. It is better to have a limit of the max number of pipelines a DN could participant in, so we don't overwhelm a single DN.
4. It is better to close pipelines/containers relatively quickly, as we don't replicate open containers. As closed containers are logically not writable then we don't have to worry much about inconsistency between closed container replicas. Then data consistency benefits.
5. It is better not to force pipelines/containers closed too often, or we can have small containers and the total number of containers grows fast and the scale of metadata managed by SCM grows fast. Then the SCM would be more and more stressed so the cluster stability suffers.

Possible concerns

After the principles aboves are raised, there are some questions to confirm based the principles:
Q1: How to calculate the max number of open pipelines in the cluster?
A1: It could be (NumberOfDNs * MaxNumberOfOpenPipelinesPerDN) / ReplicationConfig.getRequiredNodes(), just like the Ratis way. More DNs, more open pipelines; more open pipelines a DN can participate in, more open pipelines; More required nodes for a ReplicationConfig, less open pipelines**(Principle 2)**. Then how to decide MaxNumberOfOpenPipelinesPerDN? We'll answer this in Q3.

Q2: How to calculate the min number of open pipelines in the cluster?
A2: For the first version, we have a config item: ozone.scm.ec.pipeline.minimum(default: 5), it limits the open pipelines for each EC ReplicationConfig. But here we don't really pre-allocate 5 pipelines for each EC ReplicationConfig, it actually means that we always tend to create a new pipeline when a AllocateBlock request comes in and we currently have < 5 open pipelines. It actually sounds more like a high-water mark rather than a min. And it works to keep the number of pipelines/containers bounded and closed quickly, so data consistency benefits**(Principle 4)**. But a small value for a big cluster may slow down the performance as data is not spread widely**(Principle 1)** and clients will have to retry frequently when containers keep getting closed one by one. IMHO, a min number of open pipelines seems not so important, it could be naturally 0, the more critical thing is to control how the open pipelines grows.

Q3: How to calculate the max number of open pipelines for a single DN?
A3: This should be a function of the resources(disks, network bandwidth, cpu, mem) a DN holds, mainly the disks for a storage system. The more resources a DN has, the more pipelines it could serve. Naturally for each single kind of resource, the more the better, but we could be possibly bounded by the kind that runs out first. I don't know the exact calculation, but we could make it configurable like the Ratis way does(ozone.scm.datanode.pipeline.limit)**(Principle 3)**. Making it configurable also benefits the situation that we have a physical server shared by a DN service and some other services like Yarn, then we could limit the resources that a DN would use and keep all services running steadily.

Q4: How to calculate the min number of open pipelines for a single DN?
A4: IMHO, there's no way that we must force every DN to participate in at least a pipeline, because then we'll possibly get a lot of open pipelines in a large cluster which could be an overkill for the current load and containers may not be closed quickly**(Principle 4)**. So it is 0.

Q5: When do you prefer to allocate a new pipeline/container or reuse an existing one?
A5: Stephen has suggested a mechanism based on "open container requests" in which a higher request rate above some function of current pipeline count and rate hints to allocate more new pipelines and if the rate reduces again, we can allow the pipelines to decay back to a smaller value. (Actually this makes me think of the thread pooling mechanism, in a thread pool when we have a lot of tasks queued, the pool tends to create more threads to serve the tasks towards a max pool size.) This should basically work, since a higher rate indicates that there may potentially be a lot of client write requests, and the newly allocated pipelines will spread the load(**Principle 1**), and we are bounded by **Principle 2** and **Principle 3**. A lower rate indicates that we should reuse existing pipelines to serve the requests and containers will gradually go full and close(**Principle 4**).

For example we track the number of AllocateBlock requests arrived in the past 10s, if the rate is higher than the current number of open pipelines, we tend to allocate new pipelines, if lower we tend to find an existing pipeline first, if not found at last, allocate a new one. Of course we don't force pipelines to close before they are full(**Principle 5**), we just let them go full and close naturally.

At last the key point here is to decide the exact, performant, stable calculation method of the grow hint and reuse hint. We may have to run some actual tests to figure out some rationales with a prototype implementation.

Q6: When do you prefer to close an existing pipeline/container?
A6: In **Principle 4**, we stated that we should have pipelines/containers closed, **relatively**, quickly. But I should say that SCM is not in charge of this at most times. Normally, container replicas get closed when it is full(5GB) and DN sends notifications about this action to SCM. SCM could also do proactive checks over the total size of the container group, but it checks by referring to container reports which are sent at a default rate of 1 hour(hdds.container.report.interval), so these reports are quite delayed, usually we close the container group reactively by DN notifications.
Of course SCM could contribute to **Principle 4** by allocating new blocks in existing containers rather than allocating new pipelines. Or we could introduce a "container age" as Stephen suggested to force close an existing pipeline/container when it is old enough.
IHMO, on the SCM side, for now we may not want to introduce complex but inaccurate mechanisms trying to close an existing pipeline/container, we'd better focus more on Q5. Then I think we don't have to worry much about **Principle 5**.

Current Solution

The current implementation is in WritableECContainerProvider#getContainer, here is a screenshot with some unimportant things omitted or simplified(logs, comments):

```
/**
 * Get a proper container for an AllocateBlock Request
 * @param size the block size configured, default 256MB
 * @param repConfig ec replication, e.g. 6+3
 * @param owner the owner of the container
 * @param excludeList datanodes/container/pipelines should not be considered.
 * @return a container group for a block group to resides in
 */
public ContainerInfo getContainer(
      final long size,
      ECReplicationConfig repConfig,
      String owner
      ExcludeList excludeList) {

    // Space calculation for EC, we don't need a whole block of 256MB for each replica.
    long requiredSpace = Math.max(1, size / repConfig.getData());

    synchronized (this) {
      int openPipelineCount = pipelineManager.getPipelineCount(repConfig,
                                  Pipeline.PipelineState.OPEN);
      // If the current pipeline count is under the bound, feel free to create new.
      if (openPipelineCount < providerConfig.getMinimumPipelines()) {
        try {
          return allocateContainer(repConfig, requiredSpace, owner, excludeList);
        } catch (IOException e) {
          ...
        }
      }
    }

    // Here we hit the bound, we must first try to reuse existing ones.
    List<Pipeline> existingPipelines = pipelineManager.getPipelines(repConfig,
                                          Pipeline.PipelineState.OPEN,
                                          excludeList.getDatanodes(),
                                          excludeList.getPipelineIds());
    PipelineRequestInformation pri = PipelineRequestInformation.Builder.getBuilder()
                                        .setSize(requiredSpace)
                                        .build();
    while (existingPipelines.size() > 0) {
      // try to choose a pipeline that satisfies the request
      Pipeline pipeline = pipelineChoosePolicy.choosePipeline(existingPipelines, pri);
      if (pipeline == null) {
        // Not even one match, goto create new.
        break;
      }

      synchronized (pipeline.getId()) {
        try {
          // Get the only container for this pipeline.
          ContainerInfo container = getContainerFromPipeline(pipeline);
          if (container == null || !containerHasSpace(container, requiredSpace)) {
```

```
            // Maybe the container is already closed.
            existingPipelines.remove(pipeline);
            pipelineManager.closePipeline(pipeline, true);
          } else {
            // The Container is on the black list, not usable.
            if (containerIsExcluded(container, excludeList)) {
              existingPipelines.remove(pipeline);
            } else {
              // Found!
              return container;
            }
          }
        } catch (PipelineNotFoundException | ContainerNotFoundException e) {
          ...
        }
      }
    }

    // No existing one matches, create new one.
    try {
      synchronized (this) {
        return allocateContainer(repConfig, requiredSpace, owner, excludeList);
      }
    } catch (IOException e) {
      ...
    }
  }
```

The current implementation matches several principles mentioned above.

- **Principle 1: partially match**, we keep creating new ones without considering existing ones before we reach the bound. Even after the bound is reached, we have a chance to create new ones if the existing ones do not meet the requirement. But because of the often delayed reports, we have a big chance to try too hard to reuse on high concurrent loads.
- **Principle 2: no match**, we haven't defined a max limit for a cluster.
- **Principle 3: no match**, we haven't defined a max limit for a single DN.
- **Principle 4: match**, actually we may be trying too hard to reuse with a small bound configured.
- **Principle 5: match**, a frequent reuse will safely make the existing containers full, there are hardly any small containers.

To summarize, the current implementation is good enough after we define the max limits which would be simple.

But there are certain problems:

- The defined bound: *ozone.scm.ec.pipeline.minimum* which defaults to 5 will slow down the performance on high concurrent loads, and we have to tune it larger manually. But when we tune it larger, we match more for Principle 1 but match less for Principle 4, because new block groups are spreaded wider and each container has less chance to get full quickly. Of course I think Principle 1 and Principle 4 are

naturally conflicted. But we'd better think more about how to define this bound and change the confusing name.

Actually there are not many problems here, except for the max limits, we just have to find a better balance between Principle 1 and Principle 4 or in other words balance the performance and data consistency or simply we have to make the number of pipelines grow faster after we reach a more reasonable bound.

I think the "open container requests" rate as a hint for the current load is a good idea, but there's hardly any rationale to actually define a proper bound based on this idea.

Possible optimizations

So the main target is balance between **Principle 1** and **Principle 4**.
Also we have to introduce the 2 max limits for **Principle 2** and **Principle 3**.

Optimization 1

Here I have a point that **EC is used mostly for big files rather than small files**. The main point of EC is to save storage space and we have cost down while still keeping good data reliability. But for small files that are less than chunkSize(e.g. 4KB, 64KB, 1MB files), EC actually may waste more space than Replication. So we could consider optimizations towards the big file scenario and don't have to worry about the small file scenario.

So for big files(e.g. 500MB, 4GB, 10GB, 30GB files), we tend to allocate full blocks and the allocated blocks tend to be fully filled indeed except for some tailing blocks(e.g. a file of (256 * 3 + 1)MB, then the last block is only 1MB). So I suggest using the `requiredSpace` in as a hint(and just a hint, not used as the actual size, the actual size still comes from the DN reports) for preallocating a new pipeline/container, such as:

```
      synchronized (pipeline.getId()) {
        try {
          // Get the only container for this pipeline.
          ContainerInfo container = getContainerFromPipeline(pipeline);

          // Here we have chosen a container, let's check if it is about to be full
          // with the `requiredSpace`.
          preallocateContainerIfNeeded(containerInfo, requiredSpace,
                                       repConfig, owner, excludeList);

          if (container == null || !containerHasSpace(container, requiredSpace)) {
            // Maybe the container is already closed.
            existingPipelines.remove(pipeline);
            pipelineManager.closePipeline(pipeline, true);
          } else {
            // The Container is on the black list, not usable.
            if (containerIsExcluded(container, excludeList)) {
              existingPipelines.remove(pipeline);
            } else {
              // Found!
              return container;
            }
```

```
          }
        } catch (PipelineNotFoundException | ContainerNotFoundException e) {
          ...
        }
      }

...

  private void preallocateContainerIfNeeded(ContainerInfo containerInfo,
                                            long requiredSpace,
                                            ECReplicationConfig repConfig,
                                            String owner,
                                            ExcludeList excludeList) {
    ContainerID containerID = containerInfo.containerID();
    long allocatedSpace = containerAllocatedSpaceMap.getOrDefault(containerID, 0L);

    // The chosen container is possibly full, we could still use it this time,
    // but we'd better allocate a new one ahead for further loads to go smoothly
    // instead of keeping reusing the existing ones with very delayed reports.
    if (allocatedSpace + requiredSpace > containerSize) {
      try {
        allocateContainer(repConfig, requiredSpace, owner, excludeList);
      } catch (IOException e) {
        ...
      }
    }
  }
```

So in the above sample code, we introduce a Map to track the allocated space for each container, if the allocated space hints that the container is going to be full, then we could allocate a new container ahead before the DN sends an actual container close action.

Here we don't return the newly allocated container, we still return the chosen existing one, we just ensure that for the next AllocateBlock, there will be one more container to choose from, in this way we get closer to **Principle 1**. We may go a little further from **Principle 4** since existing containers have less chance to be chosen in the next round.

We try to reach a balance between **Principle 1** and **Principle 4**, because SCM has a random pipeline choosing policy by default, so we have a relatively fair chance for an existing container and a new container to be chosen. And we only preallocate a new container once we detect that there is potentially a container to close, so ideally we may get 1 in 1 out, in other words the number of open containers stay steadily at the balanced point.

For the optimization itself, the in-memory map size would be bounded by the max limit to be introduced later, and of course we should pay attention to cleanup the map introduced when containers actually get closed.
I shall have a POC test of this optimization to see if it helps with the performance of concurrent big files writes and how many open/total containers there will be compared to the current implementation.

Optimization 2

As we stated before, we should first introduce a max limit for each datanode about the open pipelines it can participate in, then a calculated max limit for the cluster or for each EC ReplicationConfig.

Optimization 3

Calculate a more reasonable value for the current *ozone.scm.ec.pipeline.minimum* as a balanced limit of the open pipelines for each EC ReplicationConfig. Maybe it could be a factor of the max limit calculated above, e.g. min = 0.5 * max, because I think this value is also related to the cluster scale and resources of a single DN, larger clusters tend to serve larger loads, thus more pipelines when we reach a balanced state. And at this balance point, the overall performance should be steady, more pipelines won't increase much performance.

Optimization 4

In extreme cases, if we reach the max limit and later the load returns to normal, how to decrease the number of open containers? If we do pre-allocation all the time, we'll tend to stay at the max limit for long, this violates **Principle 4**.

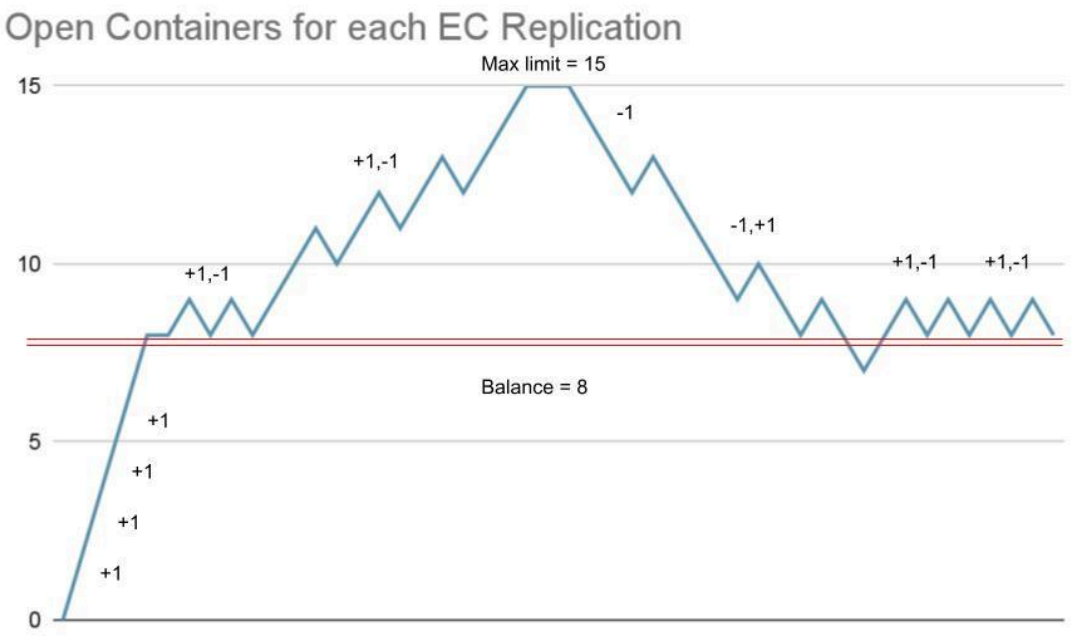
I think we do pre-allocation for the performance reason, but the performance won't benefit much when there are too many open containers, so once we reach the max limit, we could turn preallocation off until the number of open containers goes below the balance point.

```
private void preallocateContainerIfNeeded(ContainerInfo containerInfo,
                                          long requiredSpace,
                                          ECReplicationConfig repConfig,
                                          String owner,
                                          ExcludeList excludeList,
                                          List<Pipeline> existingPipelines) {

  // We reached the max once and we tend to go down towards the balance point.
  // Of course we still may allocate new containers if no existing containers match.
  if (afterMaxReached) {
     if (existingPipelines.size() >= providerConfig.getMinimumPipelines()) {
       return;
     }
     // If we go below the balanced point, we restart preallocation again.
     afterMaxReached = false;
   }

   ContainerID containerID = containerInfo.containerID();
   long allocatedSpace = containerAllocatedSpaceMap.getOrDefault(containerID, 0L);
   if (allocatedSpace + requiredSpace > containerSize) {
     try {
       allocateContainer(repConfig, requiredSpace, owner, excludeList);
     } catch (IOException e) {
       ...
     }
   }
```

```
}
```

Summary

At last the big picture I want to reach is like the following line(just ideally):



Here are the key points:

- There is a balanced point for a cluster, it balances performance and data consistency (**Principle 1 and Principle 4**). Well calculate this value in Optimization 3 and we could have a config item as well.
- There is a max limit for a cluster so as to prevent the cluster from overloaded and it is based on a limit for a single DN(**Principle 2 and Principle 3**). Also could be both calculated and configured.
- We don't force close containers at all, instead we drive the container to close by reuse it more often(**Principle 5**).
- Below the balance point, we always allocate containers.
- After we reach the balance point, we do pre-allocation upon detecting a about-close container and try to stay balanced by 1 in 1 out.
- We may still reach the max limit on heavy loads, after reaching, we turn off pre-allocation since it should not benefit performance any more.
- Then we tend to reuse heavily but will still allocate new ones if none of the existing matches the requirement.
- As load goes down, heavy reusing will guarantee to bring down the number of open

containers.

- After going below the balanced point from the max limit, we could turn pre-allocation on again to stay there.

POC Test

Test Case1

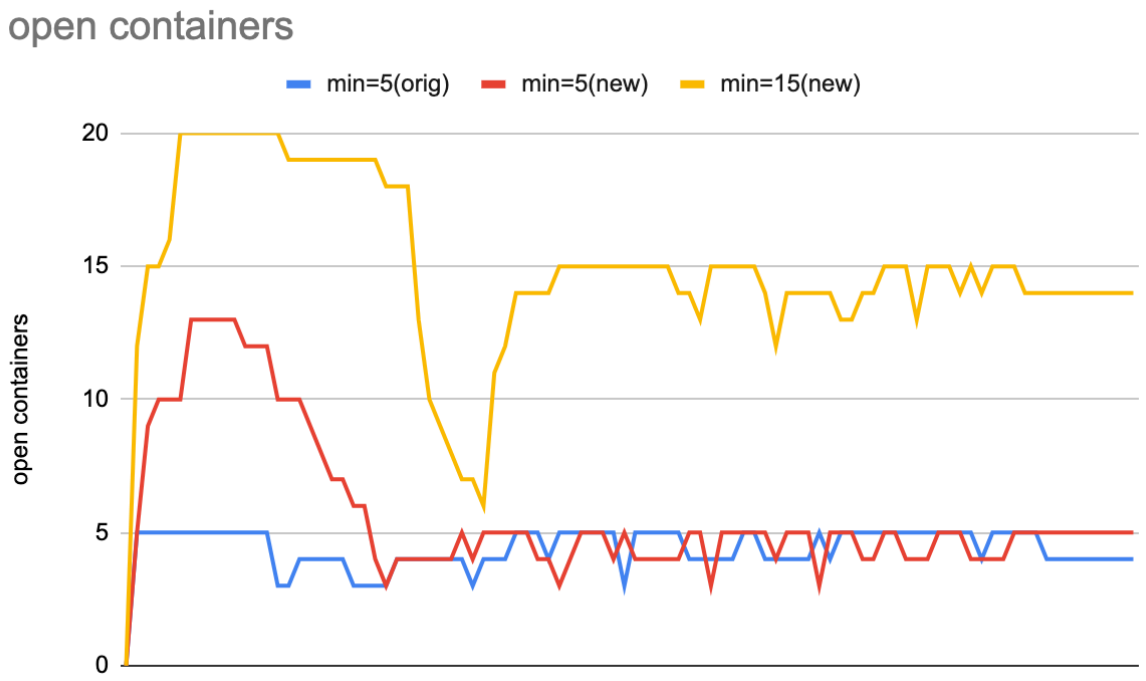
Large files write with increasing loads, watch the open/total containers and overall performance. New big file creations are big loads for this part since there are key level pre-allocations done during the file creation(see *ozone.key.preallocation.max.blocks*).

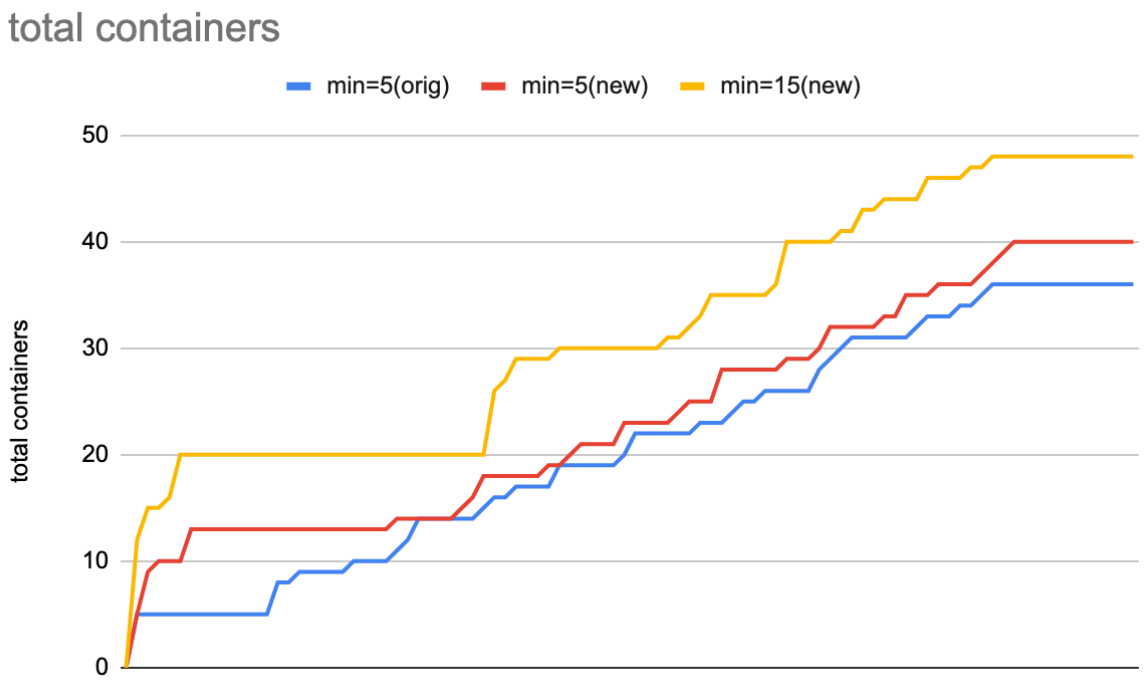
```
for ((i=0; i<5; i++)); do
    ozone freon ockg -s $((5*1024*1024*1024)) --type=EC
--replication=rs-3-2-1024k -n 10 -t 10 -p $i
    sleep 5
done
```

Ozone configurations to help make more containers:

```
  <property>
    <name>ozone.scm.container.size</name>
    <value>2GB</value>
  </property>

  <property>
    <name>ozone.scm.block.size</name>
    <value>128MB</value>
  </property>
```

I’ve tracked the open/total containers during the test with the current code and modified code:





Here the blue line is the current code, the red and yellow line are the updated code with different min pipelines configured.
We could see that:

- The number of open containers will stay at 5 with the current code and the number of total containers grows slow.
- The number of open containers will increase first with increased load and go down back to the min and stay steadily with the new code.

- The number of total containers are more with the new code, so data should be more spreaded across the cluster.

Here we are testing with the following hardwares:

- 3 physical servers with 10 simulated DNs for each, thus a cluster of 30 DNs.
- 1 dedicated physical server with a client.
- Servers have ~80cores, 250G mem, 10 disks each, bandwidth 10Gib.

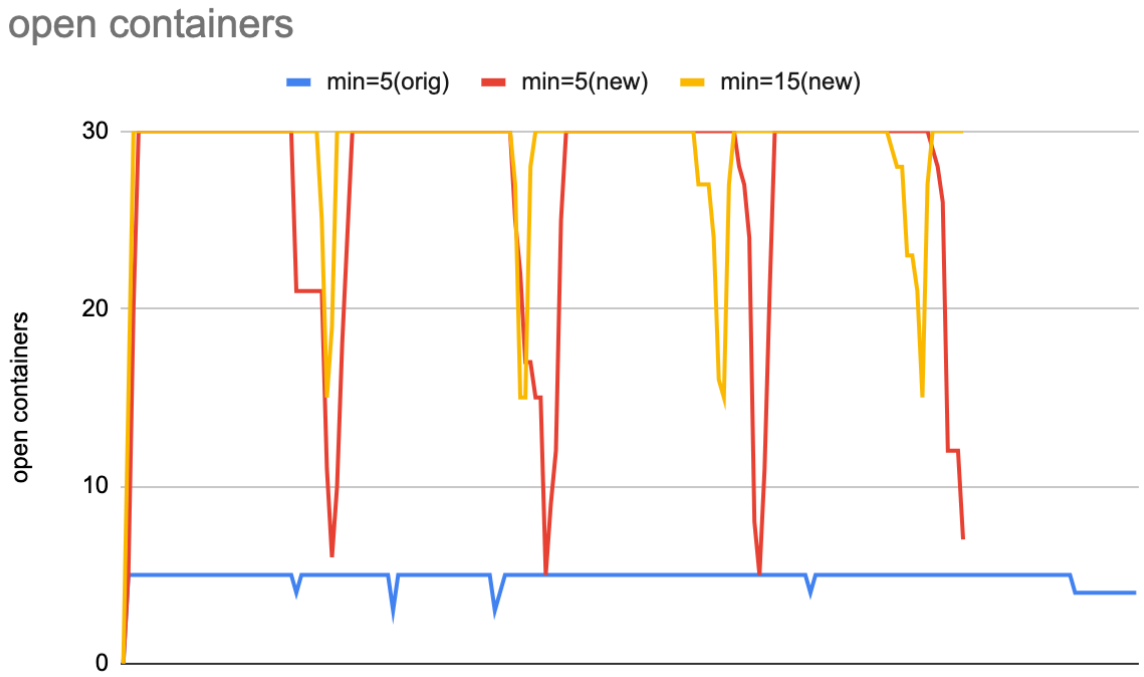
From the test, the overall performance actually does not benefit much with the new code, the total time cost of the 5 ockg rounds are all around 360s~390s. But I think it is due to the fact that the bandwidth of the client is already overloaded with this load.
The total data size are 5G * 10 * 5 = 250G, with EC3+2, actual data to transfer is 250G / 3 * 5 = 417G, so the average data transfer rate is 1.06~1.15GB/s which is very close to the max bandwidth(10/8 = 1.25GB/s). But if we have a larger bandwidth on the client side, we should gain more performance improvements(but I don't have such servers at hand :D).

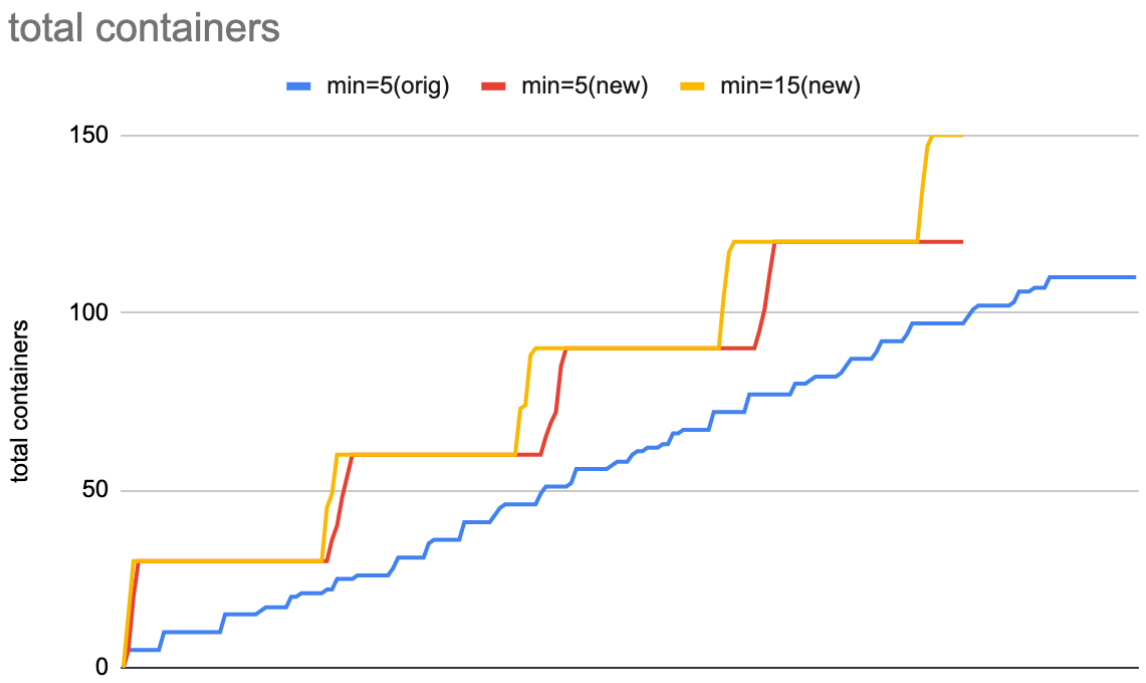
Test Case2

Small files continuously create in large numbers, watch the open/total containers and overall performance.

```
ozone freon ockg -s $((1*1024*1024)) --type=EC
--replication=rs-3-2-1024k -n 250000 -t 50
```

We have the same amount of data and concurrent threads with test case 1 to stay fair, file size is 1MB. The ozone-site.xml is the same.





Here we see that:

- We'll have more open containers most of the time with the new method, but we are at least bound by the max limit, and after max limit is hit, open containers tend to go down to the min by reusing. At last we have 10% more containers in total with the new method(min=5) and 40%(min=15).
- On the other hand, the overall time cost is ~870ms for orig, and ~770ms for new(both min=5 and min=15), which has about 13% raise in performance as load is spread across better. And this time we don't hit the upper bound of the client output bandwidth, so the performance gain is valid.

So generally we still have a reason to open more pipelines for small file writes for performance at the cost of more open containers.

At last, this new method is trying to fix the problem with the current method that it is not able to adjust the number of open containers with dynamic "loads". And I do it by balance between **Principle 1** and **Principle 4** instead of just focusing on **Principle 4** as the current method does.

The key point of the new method is preallocating pipelines by tracking allocated space which tends to be accurate on large file writes but a bit over-estimated on small file writes. And we can not be accurate all the time because of the current block allocation method. We never know ahead of time how much data we are going to write and we are going to allocate at least one block. But we have **Principle 2** and **Principle 3**, then optimization 4 could give a chance to do a best effort to stay closer to **Principle 4** as the graph above shows.

Suggestions:

- More configurable options to affect pipeline allocation/reuse.