Preemption Hardening

Background

Current Preemption Implementation doesn't behave properly especially with different (or extra) resource types existing between ask, victims and guaranteed quota. When anyone of these entities has resource types not present on the others, results are not coming out as expected. In addition, preemption kills unnecessary victims selected from other different queue paths while the current preemptor (or ask) queue path may violate the queue max resource constraint after accommodating the ask due to not being very strict during fence selection.

Current Methods like IsWithIn, IsAtorAbove, GetRemaining() doesn't take extra resource types into consideration. All these methods work perfectly when resource types are the same on the ask and victim side, even on guaranteed quota configuration. These methods do not expose the actual values and return boolean values. Also, max resources have been taken into consideration which we think not required or should not be considered as all these three methods should work purely dependent on the guaranteed quota configuration. Because of all these problems, results are not coming as expected in certain cases.

We introduced two new methods like GetPreemptableResource(), GetRemainingGuaranteedResource() to replace the existing methods IsWithIn, IsAtorAbove, GetRemaining() to expose the actual values purely based on the guaranteed quota configuration.

How GetPreemptableResource works?

Any resource used more than defined in Guaranteed resource is preemptable. In case of resource types not defined in Guaranteed, it is completely preemptable.

For example,

1)
Guaranteed resource: vcores:5
Used: vcores:6.

Preemptable resource is vcores:1

2)
Guaranteed resource: vcores:5
Used: mem:600.

Preemptable resource is mem: 600

3)
Guaranteed resource: Not defined
Used: vcores: 6, mem:600

Preemptable resource is vcores: 6, mem:600

How GetRemainingGuaranteedResource works?

Remaining can be obtained only for resources defined in Guaranteed resources. In case of resource types not defined in Guaranteed, nothing would be returned.

For example,

1)
Guaranteed resource: vcores:5
Used: vcores:4.

Remaining Guaranteed resource is vcores:1

2)
Guaranteed resource: vcores:5
Used: mem:600.

Remaining Guaranteed resource is vcores: 5

3)
Guaranteed resource: Not defined
Used: vcores: 6, mem:600

Remaining Guaranteed resource is nil

How CheckPreemptionGuarantees works?

CheckPreemptionGuarantees used to make the decisions by removing the victim from Queue Preemption Snapshot of Victim Queue and see if that removal gives back the resources to ask (or preemptor queue). If the ask queue is still within guaranteed quota, even after the above said removal, then there is a possibility or minimum guarantee that preemption could end in freeing up resources for the ask waiting in queue for resources and it makes sense to proceed further and continue the actual preemption process.

Pseudo code:

1. Add ask to the ask queue
2. Remove victim from victim queue

3. Check ask queue is still with in guaranteed or not
4. Yes means there is a possibility or minimum guarantee that preemption could help.

This kind of resource shifting from one side to the other side of the queue hierarchy worked well so far because max resource was taken into account for the calculations.

In the new world, with new definitions and methods described above, max resource has no role and only guaranteed resources have been taken into account. Above explained two methods works purely based on guaranteed resources. So, If a guaranteed resource has been defined in parent or ancestor queues in the whole ask or preemptor queue hierarchy, checkQueuePreemptionGuarantees returns the same behavior using the above two methods. In case guaranteed resources not defined in parent or ancestor queues in the whole ask to preemptor queue hierarchy, above said removal of victims would not show up (or shift) any resources on the ask side.

Example 1:

Root.parent (Max => vcores:20) .parent1 (G => vcores:10, Used => vcores:10).child1 (G => vcores:5, Used => vcores:10)
Root.parent.parent1.child2 (G => vcores:5, No usage). Ask is waiting here.
Root.parent.parent2 (Used => vcores:10)

Guaranteed set on child1, child2 & parent1. Parent have max resources set. Above pseudo code works. Child1's remaining guaranteed resource is vcores: -5, parent1's remaining guaranteed resource is vcores: 0 and Child2's remaining guaranteed resource is also vcores: 0 because min of its remaining and parent1's remaining. So, removal of victim from victim queue would make Child1's remaining guaranteed resource from vcores: -5 to vcores: 0, parent1's remaining guaranteed resource from vcores: 0 to vcores: 5 which in turn, would make ask queue, Child2's remaining guaranteed resource from vcores: 0 to vcores: 5.

Example 2:

Root.parent (Max => vcores:20) .parent1 (Used => vcores:10).child1 (G => vcores:5, Used => vcores:10)
Root.parent.parent1.child2 (G => vcores:5, No usage). Ask is waiting here.
Root.parent.parent2 (Used => vcores:10)

Guaranteed set only on child1 & child2. Max resource set on Parent. Removal of victims in above pseudo code doesn't play any role. Child1's remaining guaranteed resource is vcores: -5 and Child2's remaining guaranteed resource is vcores: 5 as calculation depends only on its own queue settings and guaranteed not set anywhere in the queue hierarchy. Ask could be accommodated always easily in Child2 because there is remaining guaranteed.

Changes in fence selection

Currently, fence selection would be either root or any queue for which FencePreemptionPolicy property has been set. It does the recursive checks starting from the ask queue all the way up to the root to choose a fence. This kind of selection is a bit generous and includes ‘n’ number of different queue paths for choosing the potential victims. At times, killing victims coming from these different queue paths may not even be required at all. For example, If any parent or ancestor queue in the whole ask or preemptor queue hierarchy has max resource set and usage already reaches its max quota, then choosing and killing victims coming from completely different queue paths is of no use because filling the ask in the ask queue would make usage exceeds the max resources anyways and violate the quota constraint. So, a fence needs to be fixed here in this queue path and victims need to be chosen from only within this boundary. Choosing a victim within this fence would free up resources and filling the ask using those freed up resources wouldn't violate any quota constraints. Hence, fence selection also depends on the queue usage and max resources equals check. If equal for any queue, that queue would be marked as a fence.

How are victims collected?

After fixing the fence using findPreemptionFenceRoot, victims are chosen from different queue paths, starting from the fence and traversing down the different queue paths. For every different queue path, reach the end leaf queue. Except for the queue whose guaranteed defined but remaining is still available, applications running in all other remaining queues would be picked and allocations belonging to those apps are considered as victims based on the priority (lower priority than ask). These potential victims are stored mainly in three different maps, allocationsByQueue (Queue wise allocations), allocationsByNode (Node wise allocations), QueueByallocations (Allocation ID wise allocations). All these maps store the values as QueuePreemptionSnapshot. QueuePreemptionSnapshot is a snapshot of the Queue from preemption perspective at any point of time.

Once potential victims have been chosen from different queue paths, the next step is to try out different nodes using tryNodes(). Overall intention of tryNodes() is to choose the node and victims running on it. Inside tryNodes(), traverse node and node wise allocations (victims) list would be passed to calculateNodeVictims() to filter the victims based on the below mentioned different criterias:

As a first step, Node resources available would be checked and returned immediately if there is a space to accommodate the ask. Otherwise, node resource constraints are there and victims need to be preempted to free up resources for ask. Traverse the victims, for every victim, remove it from corresponding queue preemption snapshot and see preemptable resource is still available or become zero because of the current victim removal using GetPreemptableResource(). If available or reaches zero just now, then add the same victim to

ask queue's preemption snapshot and see it is not exceeding its own guaranteed quota using GetRemainingGuaranteedResource(). So, victims falling under these conditions on both preemptor and victim sides are taken to the next step. Like earlier, node shortfall also would be checked. Once victims have been chosen, the same would be passed to the second pass. In the second pass, retained earlier checks except checking preemptable resources are still available or become zero after removing it from the corresponding queue preemption snapshot.

Once Node victims have been chosen for all nodes, predicateChecks would be generated for all nodes and its victims combination. Then, predicateChecks would be passed to checkPreemptionPredicates() to run the predicates (regular) and preemptionPredicates to finalize the node. predicates (regular) would be called in case of space available on the node. preemptionPredicates would be called in case of resource constraints on the node. A node and its victims would be returned in case of success, otherwise nil. Once node and its victims have been chosen, the same victim list would be passed to calculateAdditionalVictims() to see if any additional victims can be found.

Other than the victims found earlier using calculateNodeVictims(), raw additional victims list would be generated using queue wise Allocations map (allocationsByQueue). Then these raw victims would be traversed to see preemptable resources are still available or become zero because of the current victim after removal of the same from corresponding Queue Preemption snapshot using GetPreemptableResource(). If available or reaches zero just now, then add the same victim to ask queue's preemption snapshot and see it is not exceeding its own guaranteed quota using GetRemainingGuaranteedResource(). So, victims falling under these conditions on both preemptor and victim side are taken to the next step. Like earlier, queue shortfall also would be checked. These additional victims would be returned only if the ask queue remaining guaranteed is either greater than or equals zero.

These additional victims would be merged to node victims collected earlier. This merged victims list would be checked to see collected victims fulfill the ask need. In case of any shortfall, preemption won't help even though victims have been collected. So, actual preemption won't even happen. In case of no shortfall, preemption would help. Hence allocation release starts. By any chance, if there are more victims than the need, then release would happen only for the required one. Other victims are left untouched.