EOS FlatScheduler & Freespace Engine

Abhishek Lekshmanan

On behalf of EOS team

EOS Workshop 2024

Freespace Balancer

Yet another Groupbalancer engine!

Filesystem Groups in EOS

- General Hierarchy of Filesystem concepts in EOS
  - **Filesystem:** mount points hosting a single disk - eos fs ls
  - **Group:** collection of Filesystems within which files are replicated/EC - `eos group ls`
  - **Space:** collection of Groups - eos space ls

- GeoScheduler picks groups in a round robin fashion
  - Filesystems for placement then chosen based on various internal parameters
    - Disks within the groups can be heterogenous in size
    - Files coming in can be in various sizes
    - Skew of placement builds over time
  - Groups end up filling at different rates over time

GroupBalancer

- Makes groups more or less equal in size based on some configurable parameters
- Different engines that choose what group parameters to balance on:
  - **Std**: Makes groups total capacity used(%) within the threshold (%), fine tuning via min_threshold & max_threshold
  - **Minmax**: Makes groups within the min_threshold (%) and max_threshold (%), useful when wanting to balance groups with very large deviation for short time periods
- All group balancer engines take configurable parameters as min/max thresholds to configure max acceptable deviations within which groups should fall into
- `groupbalancer.blocklist` for configuring groups that do not participate in balancing

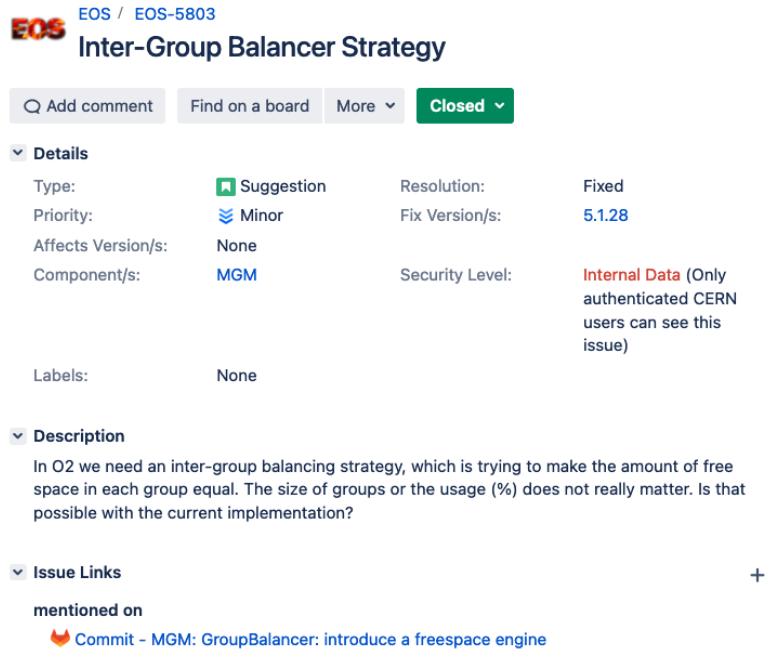
Configured via

```
eos space config <space>
groupbalancer.<key>= <val>
```

```
groupbalancer                    := on
groupbalancer.blocklist          := default.13
groupbalancer.engine             := std
groupbalancer.file_attempts      := 100
groupbalancer.max_file_size      := 10000000000
groupbalancer.min_file_size      := 10000000
groupbalancer.ntx                := 300
groupbalancer.threshold          := 2
```

Freespace balancer

- **EOSALICEO2** was expected to deliver peak data rates of over 250GB/s
  - Had groups with 14 – 16 disks, so group capacities were different
  - Since geoscheduler picks groups at round robin, different fill rates expected as they fill the instance
  - We needed to ensure groups had similar freespace (bytes) at start
- **Configuration:**
  - Min/max treshold – lower/upper limits of group's freespace

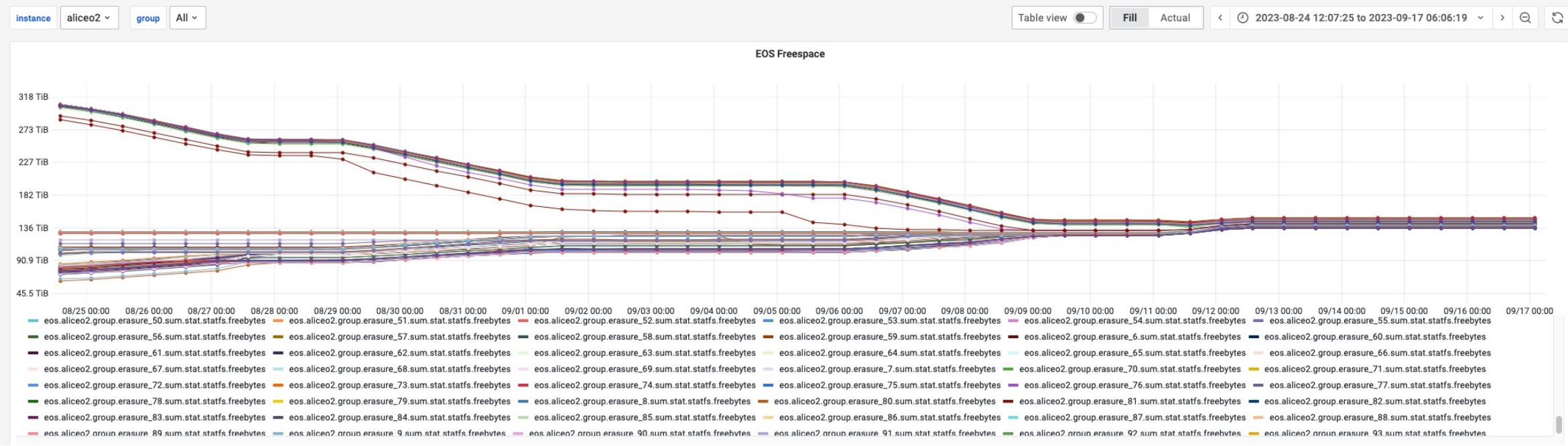



```
Engine configured: FreeSpace
Min Threshold   : 0.02
Max Threshold   : 0.02
Total Freespace : 247976935763968
Group Freespace : 6525708835893
Total Group Size: 38
Total Groups Over Threshold: 30
Total Groups Under Threshold: 8
Groups Over Threshold (Source Groups)
```

Group	UsedBytes	Capacity	Filled
default.10	3.40 T	9.00 T	0.38
default.11	3.41 T	9.00 T	0.38

Results

- In ALICEO2 cluster we calculated the freespace to be 139-147 TB and 2% threshold initially, which was lowered after everything converged

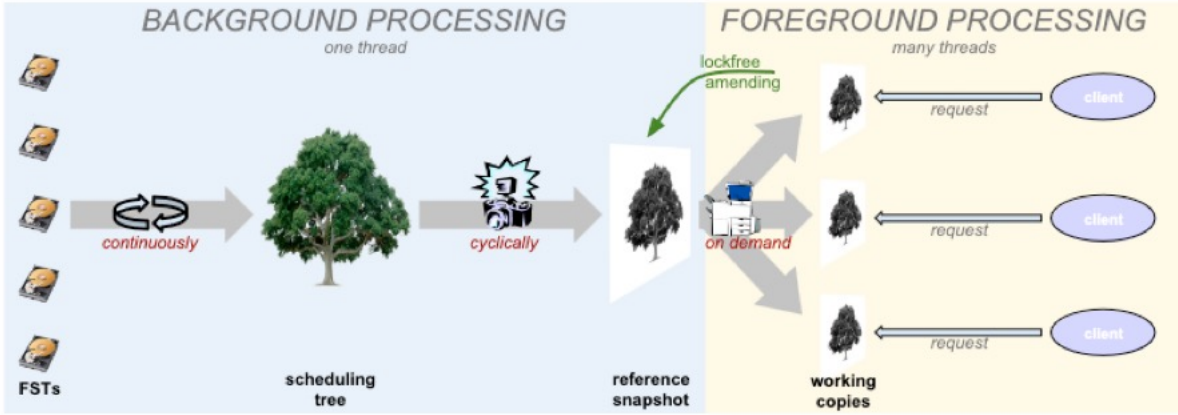


FlatScheduler

Scheduler: Purpose

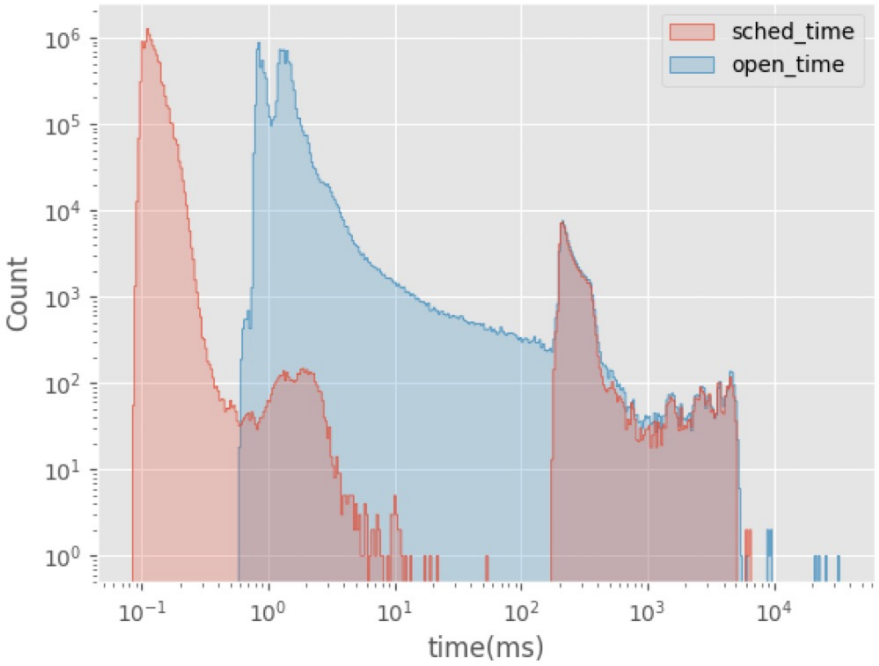
- Select Filesystems to place/access file replicas/stripes
  - Carried out at MGM
  - Always chosen within placement group - Ensures a bare minimum host level failure domain
  - Groups can span DCs
- Ensure files are evenly distributed considering
  - placement policies
  - Internal operations like draining/balancing

•GeoTreeEngine selects FS from within the groups
  - 2 tree structures - SlowTree having accurate representation and thread local FastTree snapshot
  - Double Buffer Mutex pattern to update TreeInfo



Geotree Engine – current scheduler

- Picks groups to pick in a round robin fashion
- Within the groups, filesystems are chosen based on:
  - Filesystem + client geotag
  - File layout
  - State of filesystem + machine
  - Admin penalties
  - User supplied placement options
- For large instances worst case times can be dominated by scheduling times



Flat Scheduler - Motivation

- Have simpler scheduling strategies
  - Round Robin scheduling
- Understand heterogenous storage
  - Weighted Round Robin/Random strategies
  - Better balancing within and among groups
- Ability to switch between strategies at a per space/directory/file level
- Can we perform better with these reduced constraints?
- Eventually make components like the various group/balancer progressively redundant

FlatScheduler: Design

Disks: ID corresponding to FSID, status, weight, usage

Buckets: Any other element in Storage Hierarchy

- Root, Site, Room, Rack, Group...
- Negative ID
- Contains a list of items which may be buckets or disks
- Total weight is the weight of elements underneath

ClusterData: A flat List of Buckets and Disks

RuleMap

- An array of rules of how many replicas to be chosen at each level, -1 denotes take as many items as replicas requested
- Default rule map just walks down from root -> group -> disk
- Easy to build frontends that can build this rule map

Concurrency Interlude: Publishing Pointer

- Pointer loads and stores are atomic (x86)
  - However nothing explicit about the instruction reordering
  - Compilers and hardware allowed to freely reorder instructions
- Introducing the concept of an Atomic Unique Ptr
  - Construction not thread safe, atomic loads
  - When resetting the pointer, we don't remove the old pointer, instead it is returned and the caller has to hold on to this and find a sufficiently safe point to GC
- Performance equivalent to a regular unique pointer in comparison to a Atomic SharedPointer
- Useful for data that is mostly read and rarely changing – which is our case for internal views

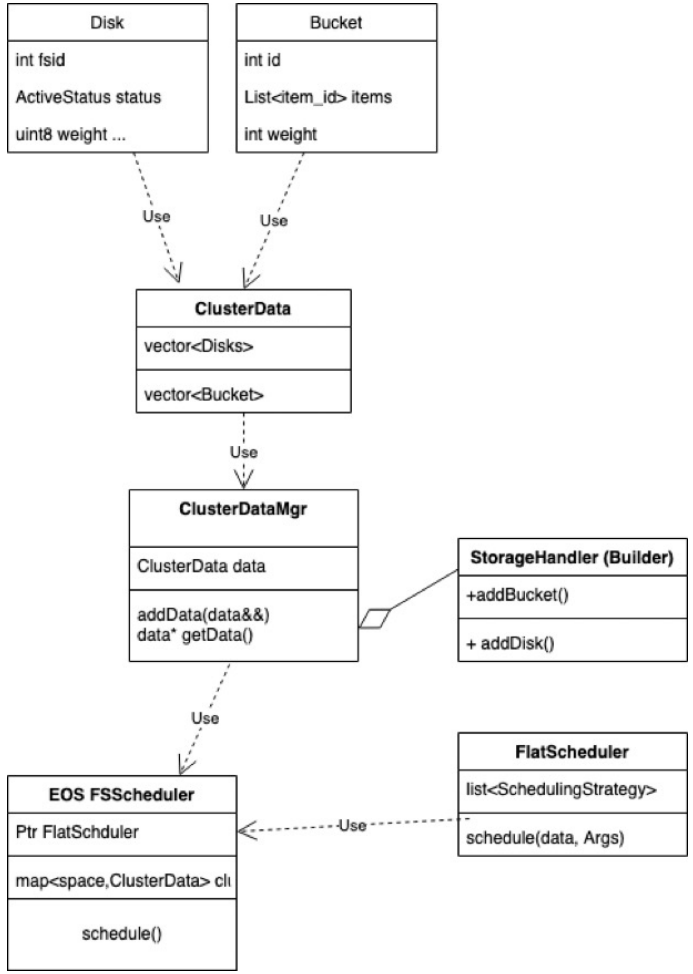
```
Running ./test/microbenchmarks/eos-atomic-ptr-microbenchmark
Run on (64 X 1798.87 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x32)
  L1 Instruction 32 KiB (x32)
  L2 Unified 512 KiB (x32)
  L3 Unified 16384 KiB (x16)
Load Average: 0.14, 0.08, 0.01
-----------------------------------------------------------------------------------------------------------
Benchmark                                                 Time             CPU   Iterations UserCounters...
-----------------------------------------------------------------------------------------------------------
BM_AtomicUniquePtrGet/real_time/threads:1          0.306 ns        0.306 ns   1000000000 frequency=3.26319G/s
BM_AtomicUniquePtrGet/real_time/threads:256        0.007 ns        0.599 ns   135805664256 frequency=145.081G/s
BM_UniquePtrGet/real_time/threads:1                0.308 ns        0.308 ns   1000000000 frequency=3.24369G/s
BM_UniquePtrGet/real_time/threads:256              0.007 ns        0.600 ns   156309076224 frequency=145.156G/s
BM_SharedPtrCopy/real_time/threads:1                10.7 ns         10.6 ns     64106208 frequency=93.8895M/s
BM_SharedPtrCopy/real_time/threads:256             0.111 ns         9.43 ns   6726456832 frequency=9.03931G/s
BM_AtomicSharedPtrGet/real_time/threads:1           25.8 ns         25.7 ns     26865129 frequency=38.8072M/s
BM_AtomicSharedPtrGet/real_time/threads:256         45.1 ns         3274 ns     22903808 frequency=22.149M/s
```

FlatScheduler: Design

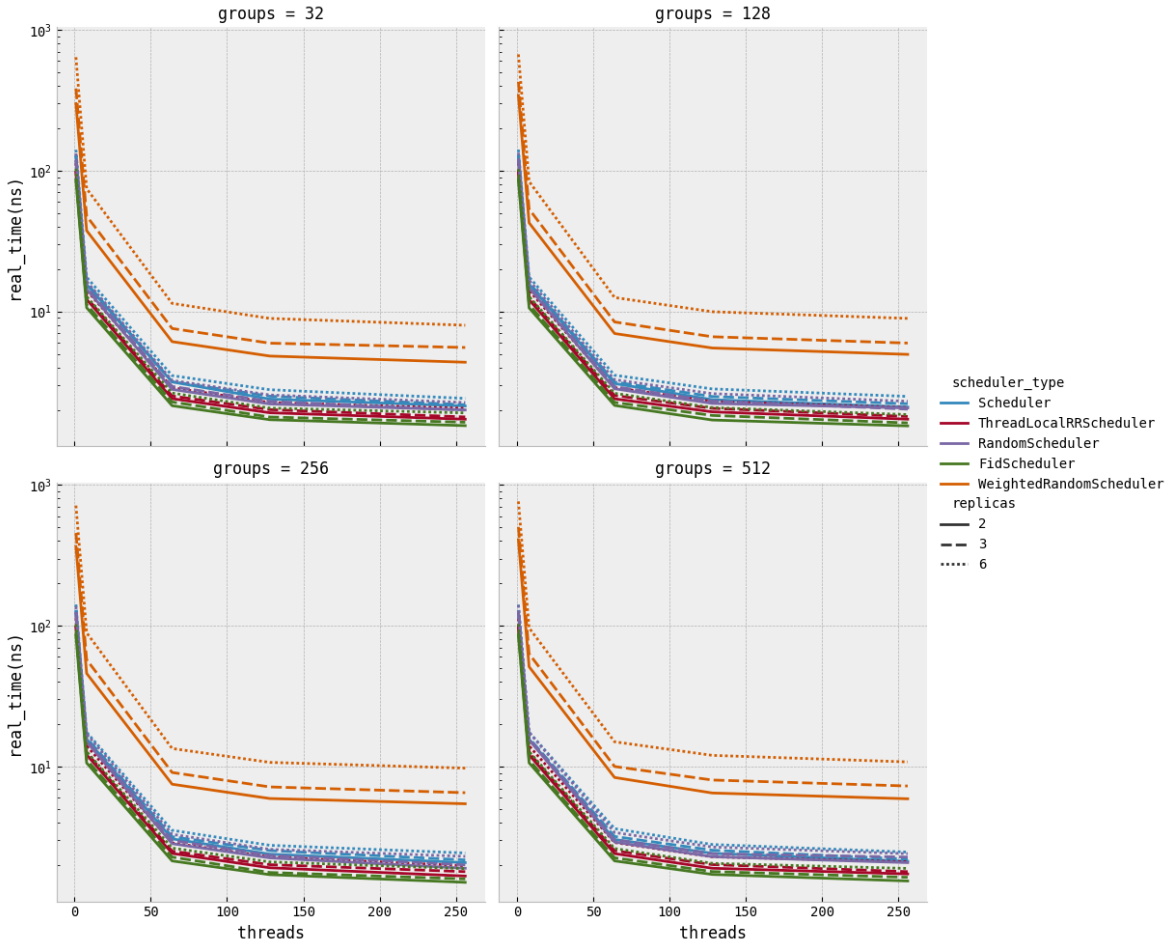
`Schedule` **delegates the task of scheduling to** `PlacementStrategy` **which decides how to choose disks or buckets**

```
const auto& bucket = cluster_data.buckets[bucket_index];
auto rr_seed = mSeed->get(bucket_index, args.n_replicas, args.fid);
int items_added = 0;
for (int i = 0;
     (items_added < args.n_replicas) && (i < MAX_PLACEMENT_ATTEMPTS); i++) {
  auto id = eos::common::pickIndexRR(bucket.items, rr_seed + i);
  // While it is highly unlikely that we'll get a duplicate with RR placement,
  // random seed gen can still generate the same seed twice.
  if (result.contains(id)) {
    continue;
  }

  item_id_t item_id = id;
  if (id > 0) {
    // we are dealing with a disk! check if it is usable
    ...validate...
    }
  }
  result.ids[items_added++] = item_id;
 }
...
```



Internal Benchmarks



- Google Benchmark library to measure time taken for a specific subroutine
- Varying group sizes and strategies against increasing thread count
  - Near linear scalability, due to atomic statuses, threads can usually make forward progress without affecting each other
  - Choice of strategy amortized at higher thread counts

Flatscheduler

- All strategies are activated, easy to switch between them, currently we have
  - RoundRobin {+ Weighted}
  - Random {+ Weighted}
- Configured via eos space config `scheduler.type` configurable
- Falls back to geoscheduler in case we don’t find a valid placement
- The internal clusterdata is built at boot time and then gets updates whenever FSTs/FsView decides to change state
- Disk statuses are all atomic, so updates are cheap

```
[root@abhi-dev-cc8 ~]# eos sched --help
 Usage:
 sched configure type <schedtype>
        <schedtype> is one of roundrobin,weightedrr,tlrr,random,weightedrandom,geo
        if configured via space; space takes precedence
 sched configure weight <space> <fsid> <weight>
        configure weight for a given fsid in the given space
 sched configure show type [spacename]
        show existing configured scheduler; optionally for space
 sched configure forcerefresh [spacename]
        Force refresh scheduler internal state
 ls <spacename> <bucket|disk|all>
```

Future work

- Testing more
  - Currently in Pilot, but future plans to make more real world test setups in Pilot & go to production in Physics instances
  - Briefly enabled in production for a different incident, bugs found have been fixed
- User Interface
  - Improving the scheduler list output to match rest of EOS, monitoring output
  - Easier interface to configure weights at a hierarchial level
- Have a slow reweighting policy when new storage is added
  - Better externally done or internally handled?

home.cern