École Polytechnique Fédérale de Lausanne

Optimizing allocation of storage partitions using constraint programming

by Swarali Karkhanis

Master Thesis

Approved by the Examining Committee:

Prof. Dr. Viktor Kunčak
Thesis Advisor

Pierre-Yves Ritschard
Thesis Supervisor

March 13, 2020

Acknowledgments

I’m would like to thank my advisor Viktor Kunčak for his encouragement and advice during the thesis.
I would like to thank Exoscale for giving me an opportunity to work on this project. I also want to especially thank my mentors at Exoscale- Pierre-Yves Ritschard and Lennert Buytenhek for their constant support and guidance during the project.

Lausanne, 13 March, 2020 Swarali Karkhanis

Abstract

Allocation of cloud resources like VMs over hypervisors, storage partitions over disks in the cloud environment need to satisfy various interrelated conditions to ensure reliability and performance. These problems can be efficiently modeled and solved with constraint programming. In this project we model the storage partition allocation problem in datacenter and study strategies to improve it. We use constraint programming for allocation of storage partitions in the data center, expansion of datacenters and to optimize reliability and recovery (in case of disk/host failures) while minimizing the impact on network and existing allocation.

The existing allocator at Exoscale uses a combination of constraint programming and randomization to allocated storage partitions. By formulating a custom search strategy on the constraint solver and leveraging uniformity at low-levels of topology hierarchy, we designed a storage partition allocator integrated completely with the constraint solver. We compare the performance of the allocator with the existing allocator in terms of run-time, quality of solution on datacenter topologies at Exoscale. The storage partition allocator is significantly faster than the existing allocator for large topology changes in the zone. It generates a fairer distribution of partitions across all hierarchies of the datacenter topology.

Contents

Chapter 1

Introduction

In the recent years, individuals and businesses are leveraging in cloud-based infrastructure at unprecedented levels. Improvements in networking technology and increase in need for storage resources have prompted users to outsource their storage needs to cloud service providers. Storage-as-a-service solutions allows customers to upload, access and distribute their data online without upfront investment. To assure customers to trust them with their data, the providers often provide guarantees in terms of availability and reliability.

Exoscale offers Scalable Object Storage (SOS) solution on it's cloud platform. They provide HA guarantees and maintain 3 copies of data across different nodes. An optimal allocation strategy of storage partition replicas in the datacenter is essential for reliability and performance. At Exoscale, the data for object storage service is partitioned into fixed sized storage partitions.

This problem falls broadly under the domain of resource allocation problem Resource allocation is one of the most common Operation Research problems. By definition in [3], the resource allocation problem seeks to find an optimal allocation of a fixed amount of resources so as to minimize the cost incurred by the allocation. Efficient resource allocation is a critical for cloud service providers like Exoscale. It not only affects reliability and quality of service but also helps to plan and minimize cost for the future. Resource allocation problem is essentially as an optimization problem. Constraint logic programming allows solving these problems by specifying constraints, objectives, domain and search strategy. However in case of a large state space, it is impossible to check all solutions. Therefore we develop a strategy to find an optimal allocation by improving the search strategy and local search.

Currently at Exoscale we have a hybrid solution with a Constraint solver at the higher level and a randomized allocater at the lower level. This works well for small changes to the existing topology (like addition of hosts, failure of disks). However for large-scale expansion of zones (like addition of a new rack), one has to compromise the benefits of a new rack or wait for a long time to get to a solution. The current solution is not fair in allocation of partitions, meaning it does not optimize the spread of partitions to optimize the usage. In this project we

optimize time and allocation of storage partitions in the datacenter by designing efficient state search strategy. The search strategy is designed to be fail-fast so that it does not waste time in looking through unnecessary states. It also relies on local search techniques that have proven to be quite useful in on practical topologies.

Chapter 2

Background

2.1 Zone layout

Exoscale offers Simple Object Storage (SOS) [2] service for storing, distributing and managing objects(files) on its cloud platform through 6 datacenters in Europe. A datacenter(zone) consists of one or more server-racks for storage. Rack is an enclosure that holds storage, networking and power-backup hardware. A standard rack is 42 rack units(RU) (~6 feet) tall where 1 RU is minimum possible height of the enclosed equipment in the rack. Rack holds storage hardware in the form of blade server enclosures(chassis). Chassis is an enclosure designed to hold blade servers in a compact and energy efficient fashion. Chassis are connected to the network through their rack's top-of-rack switch. Chassis distributes power, cooling and networking to its servers. Blade server(Host) is lowest computing unit that controls a set of hard disk drives. Disks are the smallest replaceable units in the data center.

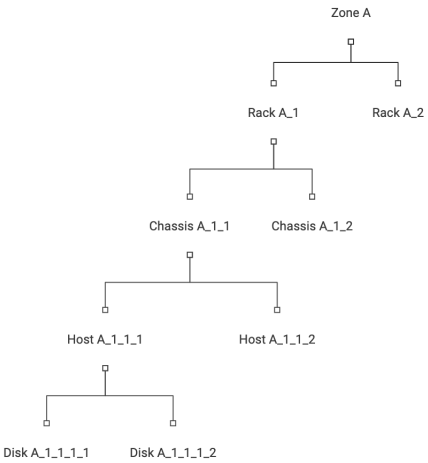

Figure 2.1 – Hierarchical structure of datacenter layout

At Exoscale, zone-expansion is done at chassis-level (ie. by addition of new chassis). New rack is added to the zone when a new chassis cannot be accommodated in the existing racks. Layout of a new chassis is usually selected from a small set of custom designed layouts. In Exoscale zones, a chassis contains mostly homogeneous hosts and a host contains mostly homogeneous disks. Another case of planned change to topology occurs in case of heavy traffic or usage on the zones. In such scenarios, reserved capacity is allotted to the zone.
Unplanned changes to the zone topology occur in case of failure in any of the components. Failures at the lowest levels of hierarchy- disk and hosts are the most prevalent. Therefore to keep the data highly available, data partitions must replicated across at-least 3 hosts, preferably at the furthest distance possible.
After replacement of a failed component, the newer component will sync up the existing replicas in the zone. Therefore it will be preferable to minimize the reconciliation load over the component containing the partition replicas by keeping the partition overlap between a pair of components to the minimum.

2.2 Formalization of partition allocation problem

In mathematical terms partition allocation problem can be described as follows:

Input:

$$
\begin{aligned}
Racks &= \{r_0, r_1, ..r_{R-1}\}, |Racks| = R \\
Chassis &= \{c_0, c_1, ..c_{C-1}\}, |Chassis| = C \\
Hosts &= \{h_0, h_1, ..h_{H-1}\}, |Hosts| = H \\
Disks &= \{d_0, d_1, ..d_{D-1}\}, |Disks| = D
\end{aligned}
$$

$$
\begin{aligned}
rack &: Chassis \rightarrow Rack \\
chassis &: Host \rightarrow Chassis \\
host &: Disk \rightarrow Host \\
capacity &: Disk \rightarrow \mathbb{N} \\
Partitions = \{0, 1, ..P-1\}, |Partitions| = P &= 3 * \sum_{d \in Disks} capacity(d)
\end{aligned}
$$

$$
existingAlloc : Disk \rightarrow PartitionSet \mid PartitionSet \subseteq Partitions
$$

Given the input following function can be defined for ease of use:

$$\begin{aligned}
&\text{For } h \in Hosts: \\
&\quad diskList(h) := \{d \mid d \in Disks, host(d) = h\} \\
&\quad existingAlloc(h) := \cup_{d \in diskList(h)} existingAlloc(d)
\end{aligned}$$

$$\begin{aligned}
&\text{For } c \in Chassis: \\
&\quad hostList(c) := \{h \mid h \in Hosts, chassis(d) = c\} \\
&\quad existingAlloc(c) := \cup_{h \in hostList(c)} existingAlloc(h)
\end{aligned}$$

$$\begin{aligned}
&\text{For } r \in Racks: \\
&\quad chassisList(r) := \{c \mid c \in Chassis, rack(c) = r\} \\
&\quad existingAlloc(r) := \cup_{c \in chassisList(r)} existingAlloc(c)
\end{aligned}$$

Output:

$$newAlloc : Disk \rightarrow PartitionSet \mid PartitionSet \subseteq Partitions$$

Given the output, following functions can be defined for ease of use:

$$\begin{aligned}
&\text{For } h \in Hosts: \\
&\quad newAlloc(h) := \cup_{d \in diskList(h)} newAlloc(d) \\
&\text{For } c \in Chassis: \\
&\quad newAlloc(c) := \cup_{h \in hostList(c)} newAlloc(h) \\
&\text{For } r \in Racks: \\
&\quad newAlloc(r) := \cup_{c \in chassisList(r)} newAlloc(c)
\end{aligned}$$

Satifying conditions:

$$\begin{aligned}
\forall_{p \in Partitions} |\{d | p \in newAlloc(d)\}| &= 3 \\
\forall_{d \in Disks} |newAlloc(d)| &= capacity(d) \\
\forall_{h \in Hosts} |newAlloc(h)| &= \sum_{d \in diskList(h) capacity(d)}
\end{aligned}$$

Optimizing conditions:

In the order of priority:

$$\text{Minimize } cOverlapLocal : \forall_{c \in Chassis, h_i, h_j \in hostList(c), i \neq j} cOverlapLocal \geq |newAlloc(h_i) \cap newAlloc(h_j)|$$
$$\text{Minimize } rOverlapLocal : \forall_{r \in Racks, c_i, c_j \in chassisList(r), i \neq j} rOverlapLocal \geq |newAlloc(c_i) \cap newAlloc(c_j)|$$

$$\text{Minimize } dOverlapOverall : \forall_{d_i, d_j \in Disks, i \neq j} dOverlap \geq |newAlloc(d_i) \cap newAlloc(d_j)|$$
$$\text{Minimize } hOverlapOverall : \forall_{h_i, h_j \in Hosts, i \neq j} hOverlap \geq |newAlloc(h_i) \cap newAlloc(h_j)|$$
$$\text{Minimize } cOverlapOverall : \forall_{c_i, c_j \in Chassis, i \neq j} Ocverlap \geq |newAlloc(c_i) \cap newAlloc(c_j)|$$
$$\text{Minimize } rOverlapOverall : \forall_{r_i, r_j \in Racks, i \neq j} rOverlap \geq |newAlloc(r_i) \cap newAlloc(r_j)|$$
$$\text{Minimize } moves := \sum_{d \in Disks} |existingAlloc(d) \setminus newAlloc(d)| \cup |newAlloc(d) \setminus existingAlloc(d)|$$

2.3 Constraint Programming

Constraint programming is a paradigm for solving combinatorial problems in Operations Research, Artificial Intelligence and Computer Science [8]. It enables the users to state their problem in a declarative fashion in terms of constraints. The constraint program run a search for the satisfying or optimizing state by efficiently searching the universe of states.

2.3.1 Constraint Satisfaction Problem

Formally a constraint satisfaction problem (CSP)[7] can be defined by a (X, D, C) where:

$X = \{X_1, \ldots, X_n\}$ is the set of variables
$D = \{D_1, \ldots, D_n\}$ is a set of domains, where D_i is the domain for X_i
$C = \{C_1, \ldots, C_m\}$ is a set of constraints, where $C_i = (\mathcal{X}_i, \mathcal{R}_i)$
defined by $\mathcal{X}_i = \{X_{i_1}, \ldots, X_{i_k}\}$ and a relation $\mathcal{R}_i \subset \mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_k}$

A state in the CSP is defined by an assignment of values to some or all variables, $S = \{X_i = v_i, X_j = v_j, \ldots\}$. A state that does not violate any constraints is called a legal assignment and a state in which every variable is assigned a value is called a complete assignment. A consistent complete assignment is a solution to the CSP. An assignment which is not complete is called a partial assignment.

CSP solvers conduct a search over the state space by moving through legal partial assignments towards a complete assignments. For discrete finite domain, a CSP solver can always find a solution (if it exists). For problems where small partial assignments can be

proven to be illegal quickly, CSP solvers can eliminate large swatches of search space quickly. In such cases CSP solvers can yield solutions much faster than regular-space searchers.

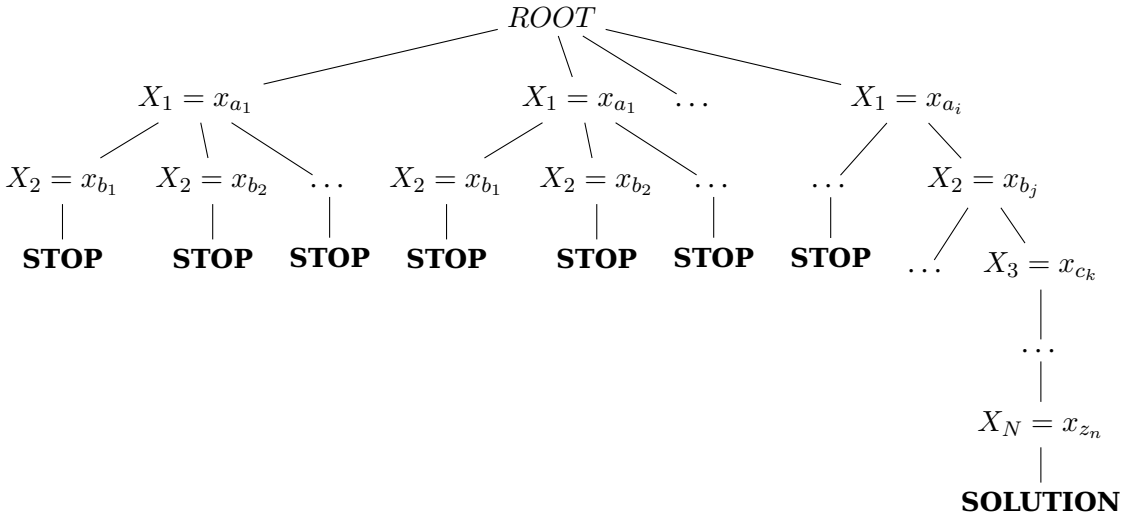

Figure 2.2 – CSP where CSP-solver is efficient

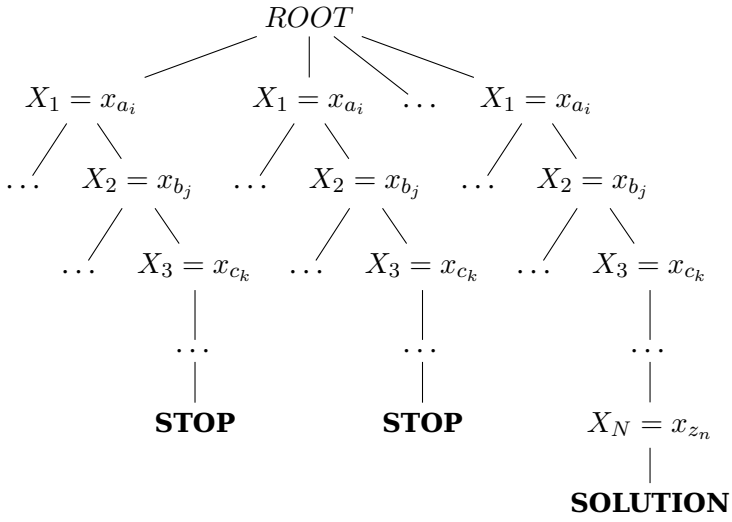

Figure 2.3 – CSP where CSP-solver is inefficient

The constraint solvers moves through the search space using the following techniques:
- Extension: Constraint solver sets a (previously unexplored) value on a variable and explores the rest of the search space.
- Propagation: Based on the already set variables, the solver propagates the values through constraints and tries to set values/ update bounds of other variables or detect contradiction
- Repair: Upon detecting a contradiction on the given state of values, solver repairs the search by back-tracking to the last feasible state.

Without an efficient search strategy, constraint solver may continue searching in wrong state space. Search strategy of a constraint solver determines:
- Variable selection: Strategy to choose a variable to extend
- Value Selection: Strategy to choose a value for the chosen variable
- Backtrack strategy: Strategy to backtrack after failure on the current path

2.3.2 Constraint Optimization Problem

In addition to the absolute constraints, many real-world CSPs include preference constraints indicating which solutions are preferred. This can typically be expressed in terms of a cost function, $f : X \to \mathbb{R}$ Such problems are called constraint optimization problems (COP). Linear programming problem fall under this kind of optimization.

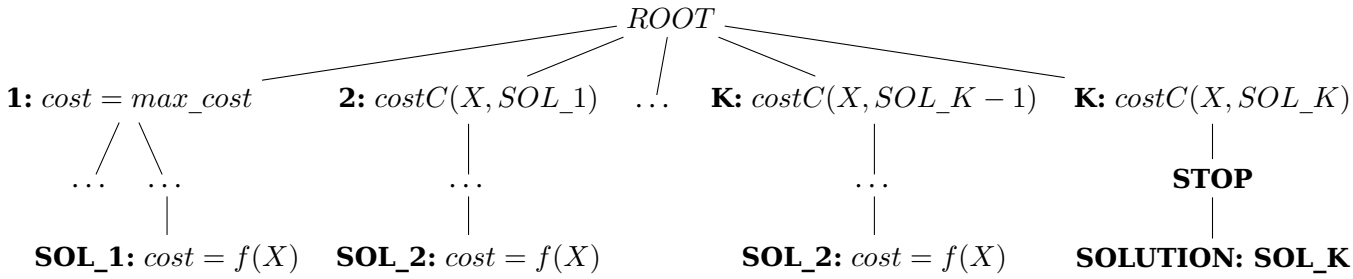

Figure 2.4 – COP with a $costC$ constraint on the variables based on the previous solution

COP can be solved using path-based or local search methods.
- Path-based search does an exhaustive search over all possible solutions by constraining the cost incrementally.
- Local search does a neighborhood search[5] over the partial assignment derived from the previous solution.

Chapter 3

Design

3.1 Partition allocation as a CSP

Partition allocation problem can be expressed as CSP: (X, D, C) as follows:

3.1.1 X- Variables

Variables is a set of locations to be used in the zone.

$$X = \{l_{d,i} | d \in Disks, i \in \{1, 2, ..capacity(d)\}\}$$

Every location will belong to disk, host, chassis and rack. For convenience and readability let us define following functions on location, $l_{d,i}$:

$$\begin{aligned} disk(l_{d,i}) &= d \\ host(l_{d,i}) &= parentHost(disk(l_{d,i})) \\ chassis(l_{d,i}) &= parentChassis(host(l_{d,i})) \\ rack(l_{d,i}) &= parentRack(chassis(l_{d,i})) \end{aligned}$$

Also conversely every disk, chassis and rack will contain a list of variable:

$$\begin{aligned} locationList(disk) &= \{l \mid l \in X, disk(l) = disk\} \\ locationList(chassis) &= \{l \mid l \in X, chassis(l) = chassis\} \\ locationList(rack) &= \{l \mid l \in X, rack(l) = rack\} \end{aligned}$$

3.1.2 D- Domains

Domain of location variable is the set of partitions.

$$D = \{Partitions | l \in X\}$$

3.1.3 C- Constraints

Partition replication constraint

Constrain every partition is allotted to exactly 3 locations. For a given partition, p:

$$\begin{aligned} ReplicaConstraint(p) = \{(\mathcal{X}, \mathcal{R}) \mid \mathcal{X} &= \{X_{i_1}, X_{i_2}, \ldots X_{i_k}\}, \\ \mathcal{D} &= D_{i_1} \times D_{i_2} \times \ldots D_{i_k}, \\ \mathcal{R} &= \{v \mid v \in \mathcal{D} \text{ if } |\{v_i \mid v_i \in v, v_i = p\}| = 3\}\} \end{aligned}$$

Disk disjoint constraint

Constrain every disk to contain only disjoint partitions. For a given disk, d, :

$$\begin{aligned} DiskDisjointConstraint(d) = \{(\mathcal{X}, \mathcal{R}) \mid \mathcal{X} &= \{X_{i_1}, X_{i_2}, \ldots X_{i_k}\}, \\ \mathcal{D} &= D_{i_1} \times D_{i_2} \times \ldots D_{i_k}, \\ \mathcal{R} &= \{v \mid \in D \text{ if } \forall_{X_i, X_j \in X \cap locationList(d), i \neq j} v_i \neq v_j\} \end{aligned}$$

Host disjoint constraint

Constrain every host to contain only disjoint partitions. For a given host, h, :

$$\begin{aligned} HostDisjointConstraint(h) = \{(\mathcal{X}, \mathcal{R}) \mid \mathcal{X} &= \{X_{i_1}, X_{i_2}, \ldots X_{i_k}\}, \\ \mathcal{D} &= D_{i_1} \times D_{i_2} \times \ldots D_{i_k}, \\ \mathcal{R} &= \{v \mid \in D \text{ if } \forall_{X_i, X_j \in X \cap locationList(h), i \neq j} v_i \neq v_j\} \end{aligned}$$

It is easy to see that $HostDisjointConstraint$ automatically satisfies $DiskDisjointConstraint$

$C = \{ReplicaConstraint(p) \mid p \in Partitions\} \bigcup \{DiskDisjointConstraint(d) \mid d \in Disks\} \bigcup \{HostDisjointConstraint(h) \mid h \in Hosts\}$

3.1.4 Solving partition allocation CSP

Since CSP constrains are only at the host level, we can ignore hierarchy and only look at the host-level allocation.

Search strategy

- Variable selector: Choose the location variables prioritized by the host size. Since the larger hosts are the more constrained than smaller ones(MostConstrained).
- Value selector: Choose the least value in the domain that satisfies its constraints(MinValue)

Consider partition allocation on host layout:

$$\{1:6, 2:4, 3:2, 4:3, 5:3\} \tag{3.1}$$

	Replica1	Replica 2	Replica3
Partition1	1	2	4
Partition2	1	2	4
Partition3	1	2	4
Partition4	1	2	5
Partition5	1	5	3
Partition6	1	5	3

Figure 3.1 – Partition allocation- MostConstrained + MinValue

	Replica1	Replica 2	Replica3
Partition1	3	4	5
Partition2	3	4	5
Partition3	4	5	2
Partition4	2	1	
Partition5	2	1	
Partition6	2	1	

Figure 3.2 – Partition allocation - LeastConstrained + MinValue

However this method is not completely fail-safe. Consider host layout

$$\{1:6, 2:4, 3:2, 4:3, 5:3, 6:3\} \tag{3.2}$$

Allocation strategies in 3.1 fails for this case.

	Replica1	Replica 2	Replica3
Partition1	1	2	4
Partition2	1	2	4
Partition3	1	2	4
Partition4	1	2	5
Partition5	1	5	6
Partition6	1	5	6
Partition7	6	3	

Figure 3.3 – Partition allocation- MostConstrained + MinValue

We observer that condensed packing of consecutive chosen variables leaves multiple replicas to be allotted to the last host.
- Value selector: we can try to spread the consecutive variables with Least-recently-used(LRU) value selection.

	Replica1	Replica 2	Replica3
Partition1	1	2	5
Partition2	1	2	5
Partition3	1	2	6
Partition4	1	4	6
Partition5	1	4	6
Partition6	1	4	3
Partition7	2	5	3

Figure 3.4 – Partition allocation - MostConstrained + LRUValue

It is easy to prove that if there exists a solution, Search with any variable selector + LRUValue-Selector on the constrained solver will find it without backtracking

3.2 Partition allocation as COP

Partition allocation is a multi-objective COP[1]. For given cost functions, $costs = \{cost_1, cost_2, \ldots, cost_n\}$ in order of priority, we could define a single cost function as follows:

Reducing to single constraint allows 2 to do an exhaustive search on the state space to find the optimal input But for Partition allocation COP it is be extremely time-consuming to use as a cost function.
Therefore we dont use the cost function directly in our design but we instead we do multiple runs of constraint solver to optimize the preference conditions in the order of their priority

Input: $X, D, C, costs$
Output: solution
$combinedCost \leftarrow \{\}$
for $i \in 1, 2, \ldots, n$ **do**
  $combinedCost_i \leftarrow function\_def(X, solution)\{$
  return $cost_i(X, solution) \times \sum_{j=i+1}^{n} upperbound(cost_i, X, solution)$
  $\}$
  Add $combinedCost_i$ to $combinedCost$;
end
$combinedCostFunction \leftarrow function\_def(X, solution)\{$
return $\sum_{c \in combinedCost} c(X, solutions)$
$\}$
$cop_i \leftarrow (X, D, C, cost_i)$
$solution \leftarrow solve(cop_i)$
return $solution$

Algorithm 1: Multi-objective COP- Reduce to single constraint

Input: $X, D, C, costs$
Output: solution
for $i \in 1, 2, \ldots, n$ **do**
  $cop_i \leftarrow (X, D, C, cost_i)$
  $solution \leftarrow solve(cop_i)$
  $C \leftarrow C \cup fix\_cost\_constraint(solution, cost_i)$
end

Algorithm 2: Multi-objective COP- Solve by constraint priority and bound constraint

and add a new constraint for the run next run to maintain the cost of the high priority cost functions.

3.2.1 Solving partition allocation COP

Minimize local overlap

Partition allocation CSP add absolute constrains to reject allocations with local overlaps within disks and hosts. At higher levels in the topology hierarchy(chassis, rack), constraining local overlaps may/may not be possible depending upon the zone layout. For example consider the following uniform allocation with partition capacity of 24 :

rack1:
chassis1_1:
host1_1_1: 3, 3, 3
host1_1_2: 3, 3, 3
chassis1_2:
host1_1_2: 3, 3, 3
host1_2_2: 3, 3, 3
rack1:
chassis1_1:
host1_1_1: 3, 3, 3
host1_1_2: 3, 3, 3
chassis1_2:
host1_1_2: 3, 3, 3
host1_2_2: 3, 3, 3

Figure 3.5 – Example of a uniform constraint

Using 3.4 from CSP problem we will get the following allocation at the chassis level:

chassis1_1: 1-18
chassis1_2: 19- 24, 1-12
chassis2_1: 13-24, 1-6
chassis2_2: 7 -24

The local host overlaps in chassis is 0 while the local chassis overlaps in the rack is 12. This is clearly the best possible allocation that minimizes the local overlap in chassis and rack. However now consider a slightly non-uniform layout where chassis1_1 and chassis2_1 contain disks with capacity 4 instead of 3. By the same search strategy in 3.4 we get the following allocation with partition capacity of 28:

chassis1_1: 1-24
chassis2_1: 25- 28, 1-20
chassis1_2: 21-28, 1-10
chassis2_2: 11-28

Here we notice that if chassis1_2 and chassis2_2 interchange, we will get better rack-level overlap. We notice that if chassis1_2 was chosen right after chassis1_1 value allocator would have chosen the appropriate set of partitions for the rack.
- Variable selector: Start with a random variable and traverse through the hierarchy in a DFS

fashion
- Value selector: Since consecutively chosen elements are most likely in proximity to each other, allocate the least recently used value.
- Backtracking: Not required.

Minimize global overlap

We have so far only considered minimizing local overlaps. Keeping balanced overlap globally across all components - disks, hosts, chassis and rack helps in minimizing network load in case of heavy operations. Consider the following 2 partition allocations on the same topology:

$$chassis1 : \{hosts1 : \{disk\_1, disk\_2, disk\_3\},$$
$$hosts2 : \{disk\_4, disk\_5, disk\_6\},$$
$$hosts3 : \{disk\_7, disk\_8, disk\_9\}\}$$

disk_1: 1-3
disk_2: 4-6
disk_3: 7-9
disk_4: 1, 4, 6
disk_5: 2, 5, 7
disk_6: 3, 6, 9
disk_7: 1, 5, 9
disk_8: 2, 6, 7
disk_9: 3, 5, 8

Figure 3.6 – Balanced overlap allocation

disk_1: 1-3
disk_2: 4-6
disk_3: 7-9
disk_4: 1-3
disk_5: 4-6
disk_6: 7-9
disk_7: 1-3
disk_8: 4-6
disk_9: 7-9

Figure 3.7 – Unblanced overlap allocation

As we can see clearly allocation 3.6 is more balanced than 3.7. In case of disk failure of $disk\_1$, the network load will distributed over $disk\_4$-$disk\_9$ in 3.6 while only $disk\_1$ and $disk\_2$,

Input: X, S, X_i, D_i
Output: v
X: Set of location variables
S: Partial state set
X_i: Variable for which the value is being chosen
D_i: Domain of variable X_i
$overlap \leftarrow 0$
while *true* **do**
  for $v \in D_i$ **do**
    $OverlapingLocations = \{X_j \mid (X_j, v) \in S, \}$
    if $|OverlapingLocations| \leq overlap$ **then**
      return v
    end
    overlap++
  end
end

Algorithm 3: MinimumOverlap

can share the lentire network load in 3.6

The LRU value selector is passive. It does not look at the toplogy of the variables it is applied on. Therefore we invert the variable selector and the value selector:
- Variable selectors: Select the farthest element unexplored location variable from the previous variable(MostDistance variable selector)
- Value selectors: Select the least legal value (No intersections within hosts and disks) with the smallest number of overlaps(MinOverlap)
- Backtracking strategy: Unlike LRUValueSelector, this value selector may cause some failures hence a backtracking strategy is required. Here we use a combination of 2 strategies: Fast-Fail and Local-search+Repair
Fail fast - means upon certain condition the solver abandons the current search tree and starts again from the root
Local-search+Repair - the new search starts with a partial solution constructed from either the previous explored tree or the last best solution. Design of the local search techniques are discussed in the next section

Repair

When Repair is invoked, it means there must be equal-size sets of unallocated locations and unallocated partitions that are not compatible with each other. Incompatibility can be a result of local and global overlap constraints of COP.

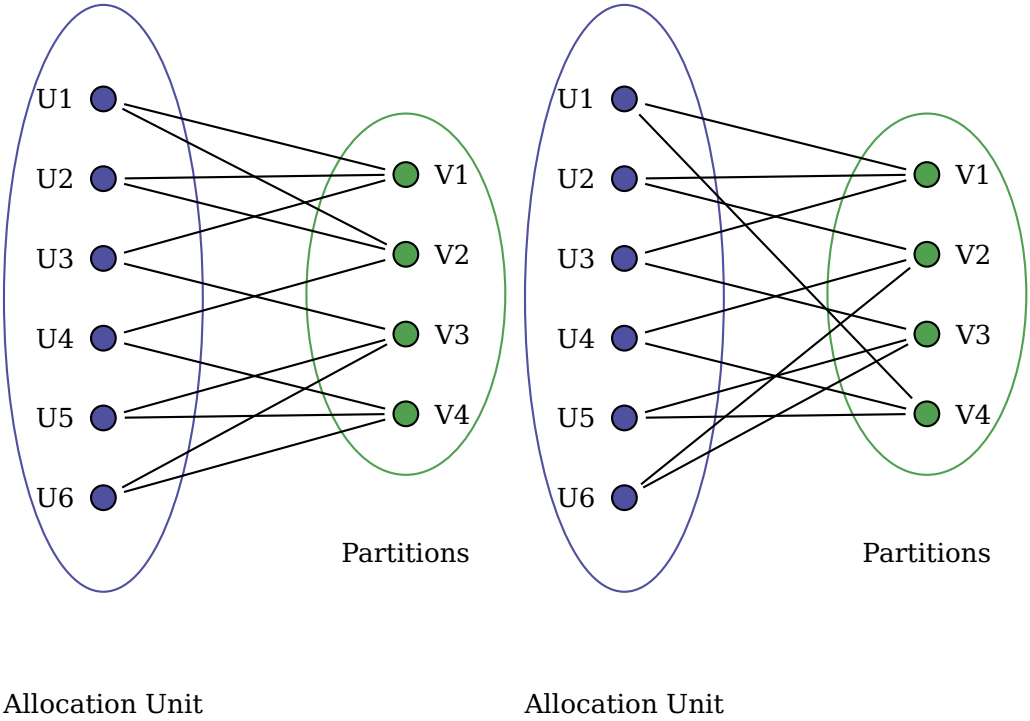

Figure 3.8 – Partition allocation as a bipartite graph

The repair logic can be explained through 3.8. Partition allocation over any allocation unit: disk, host, chassis, rack can represented as bipartite graph between the a unit set U and a partition set V. The edges represent allocation of partitions on the allocation unit(lets say disk for this example). Now we need to find a (disk and partition) pair that are not compatible.
For example: tuple $(U6, V4)$ are not compatible for overlap value=1, we find another connected tuple (Ui, Vj), preferably also non-compatible that can swap the allocation while maintaining the overlap constraints. Here $(U1, V2)$ form such a tuple. On swapping the allocations $(U6, V4), (U1, V2) \rightarrow (U6, V2), (U1, V4)$ we get compatible allocation for overlap=2.

Input: $U, V, E, overlap$
for $(U_i, V_j) \in E \mid overlap(U_i, V_j) > overlap$ **do**
  Find U_k, V_l such that
  - $(U_k, V_l) \in E$
  - overlap(U_i, V_j) + overlap$U_k, V_L \geq$ overlap(U_k, V_j) + overlapU_i, V_L
    swapn)partitions((U_i, V_j), (U_k, V_l))
end

Local Search

Local Neighbourhood Search/ Local search is executed on an already existing partial or complete allocation. Executed after the repair phase, the local search freezes some variables for the next round. Partition allocator local search maintain a growing list of frozen variables which are variables that could not be legally allocated in the previous run. Since the variable selector(MostDistanceVariableSelector) is deterministic, without local search for every run same set of variables are left out. Therefore on freezing these variables, which were usually chosen last and allotted partitions from the end of the list and are chosen first and allotted newer partitions from top of the partition list. This makes sure that allocator values are never restricted to a smaller subset of the total partition set.

Chapter 4

Implementation

Partition allocator is accessible at [4]. It is a Java based system relying on Choco constraint solver[6]. These are the main components of partition-allocator system.

Topology

This package loads and stores the zone layout data. It contains easy accessibly functions for every allocation unit in the datacenter.

Allocation

This class loads and store partition allocation data from exsting allocation in the yaml file and location variables from the constraint solver.

Allocator

This is the main running unit of the partition allocator. It is responsible for creation of Choco model, variables and constaints .

FrozenVarInputOrder

This is a variable selector that selects variables first from the frozen variables and later according to MostDistanceValueSelector.

Variable Monitor

This is a debugging monitor to follow/debug changes made to variable bounds while searching.

InitialProp

This is a constraint propagator triggered only at the beginning of a new search. InitialProp runs the repair and freeze strategy before a new run.

MinOverlap

This is constraint propagator that updates keep track of the overlaps between allocation units.

Chapter 5

Observation and Conclusion

5.1 Size of the allocation units vs number of allocation units

For the same number of partitions, topology with larger number of allocation units are easier to replicate than allocation units with larger size. For example: topo-large-2.yaml(Large topology with disk capacity 2) from the data repo can be more evenly spread and has lesser overlap with its peers. compared to topo-small-80.yaml(Small topology with disk capacity 80)

5.2 Uniformity of the topology

Allocation on a uniform/near uniform layout is faster than allocation on a non-uniform layout. Allocation on uniform layout allows for fast-failing thus prevents the solver from exploring unneccessary search spaces. With non-uniform/random layouts, the allocation problem can become NP-hard since it approaches Knapsack problem.

5.3 Local repair+search

As the numbers of allocation units grow in size, it becomes easier to find tuples to swap. Therefore repair is easier on solutions at lower level of hierarchy(disks and hosts) vs higher level of hierarchy(chassis, racks).

5.4 Partition moves

Change in the topology at the lower levels (eg: Increasing used disk capacity from 70% to 80 %) creates lesser number of partitions moves than changes in topology at higher levels(eg:

addition of new Chassis). Therefore unplanned failures at vulnerable location are easy to fix and create less load on the network and storage devices compared to planned activities like zone expansion.

Bibliography

[1] Jean-Guillaume Fages Charles Prud'homme and Xavier Lorca. *Choco Solver Documentation*. 2019. Chap. 3.

[2] Exoscale. *Exosclae Object Storage*. URL: `https://www.exoscale.com/object-storage`.

[3] Ding-Zhu Du Panos Pardalos and Ronald L Graham. *Handbook of Combinatorial Optimization*. 2013.

[4] *Partition allocator for storage partitions*. URL: `https://github.com/swarali/partition-allocator`.

[5] Stefan Pisinger David and Ropke. *Large Neighborhood Search*. 2010.

[6] Charles Prud'homme. *Choco Solver*. URL: `https://choco-solver.org`.

[7] Stuart J. Russell and Peter Norvig. *Artitificial Intelligence A modern approach*. 2010. Chap. 6.

[8] Wikipedia. *Constraint Programming*. URL: `https://en.wikipedia.org/wiki/Constraint_programming`.