H-DRF: Hierarchical Scheduling for Diverse Datacenter Workloads

Ali Ghodsi

join work with

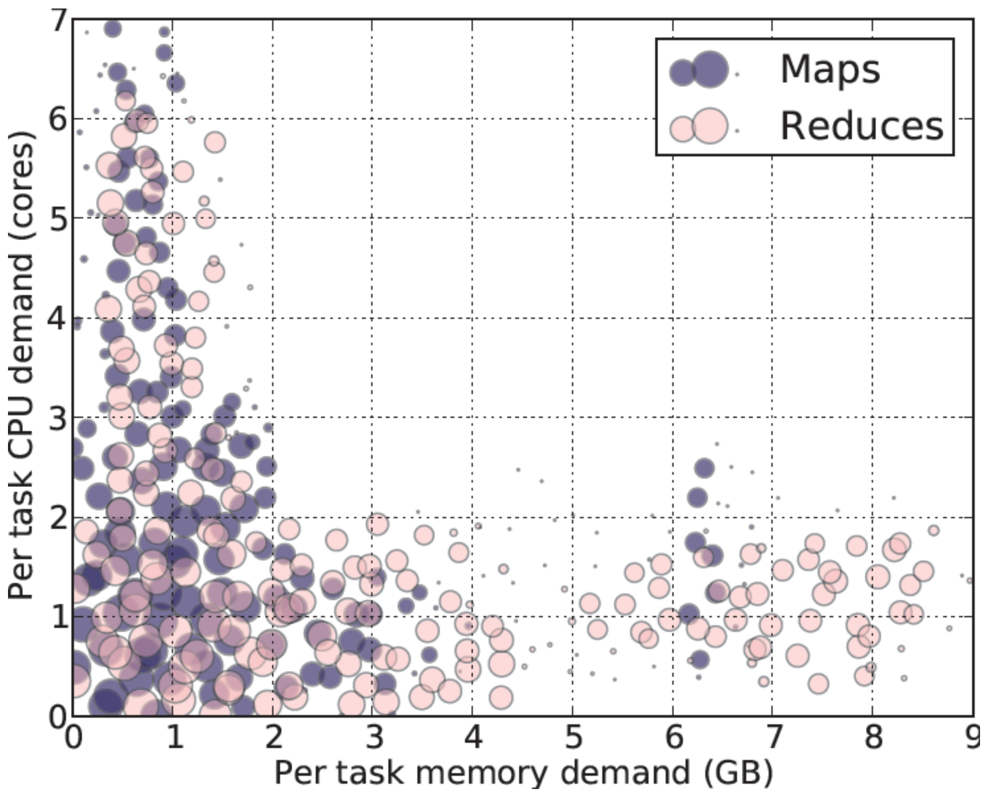
Arka Bhattacharya, David Culler, Eric Friedman, Ion Stoica, Scott Shenker

UC Berkeley

amplab UC BERKELEY

Intel Science & Technology Center for Cloud Computing

Background



- Data centers run a large mix of workloads ...leading to diverse resource requirements multi-resource scheduling necessary for isolation and efficiency

Background: multi-resource fairness

- Dominant Resource Fairness (DRF)
  - Share guarantee: guaranteed 1/n share
  - Strategy-proof: lying can only hurt you

- Well understood
  - Efficiency, extensions, limitations

- DRF now de-facto scheduler in Hadoop
  - DRF capacity scheduler (HortonWorks)
  - DRF fair scheduler (Cloudera)

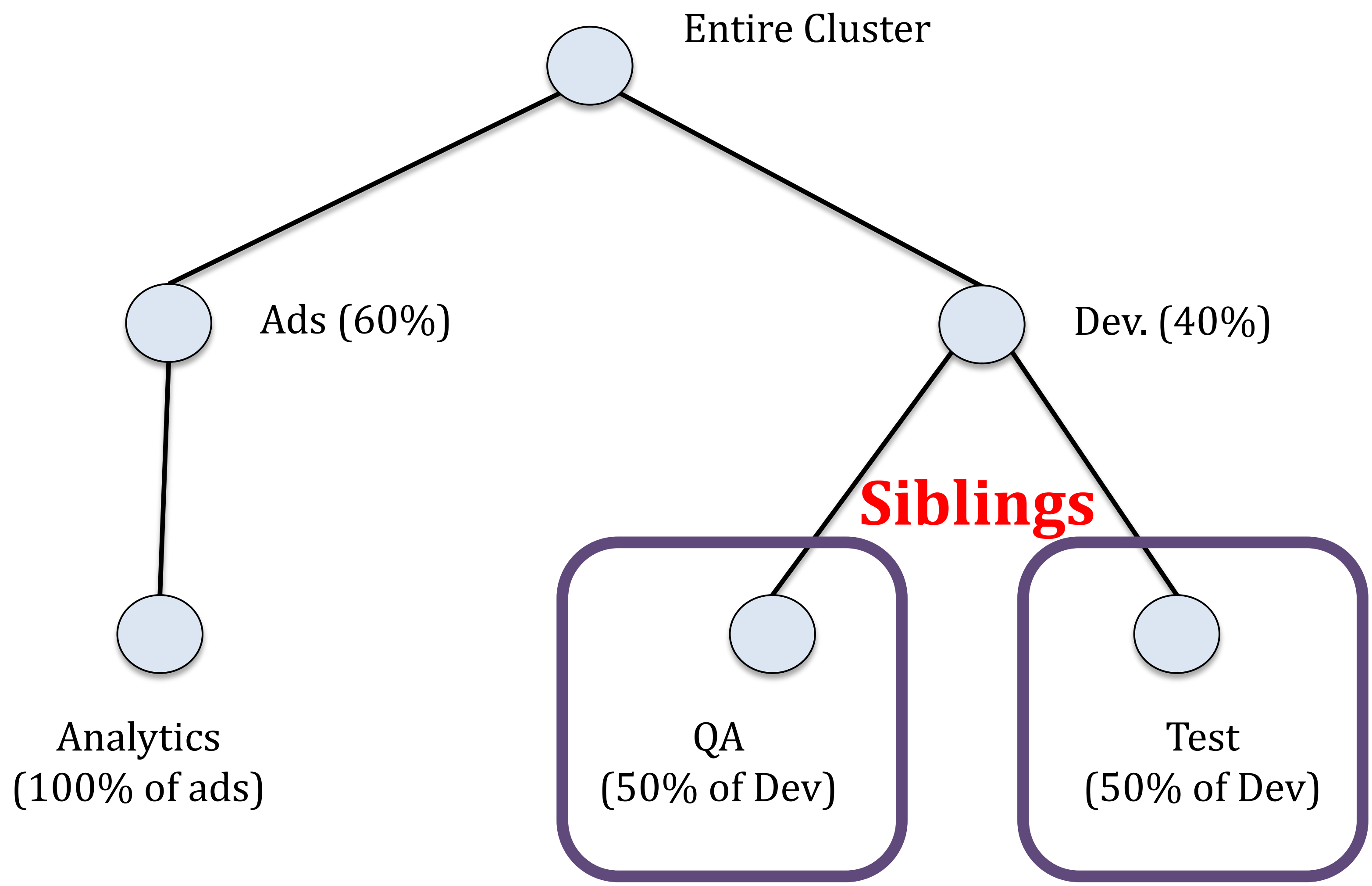
Slight problem…

- Hadoop always had hierarchical policies
  - Problem: DRF didn’t mention hierarchies

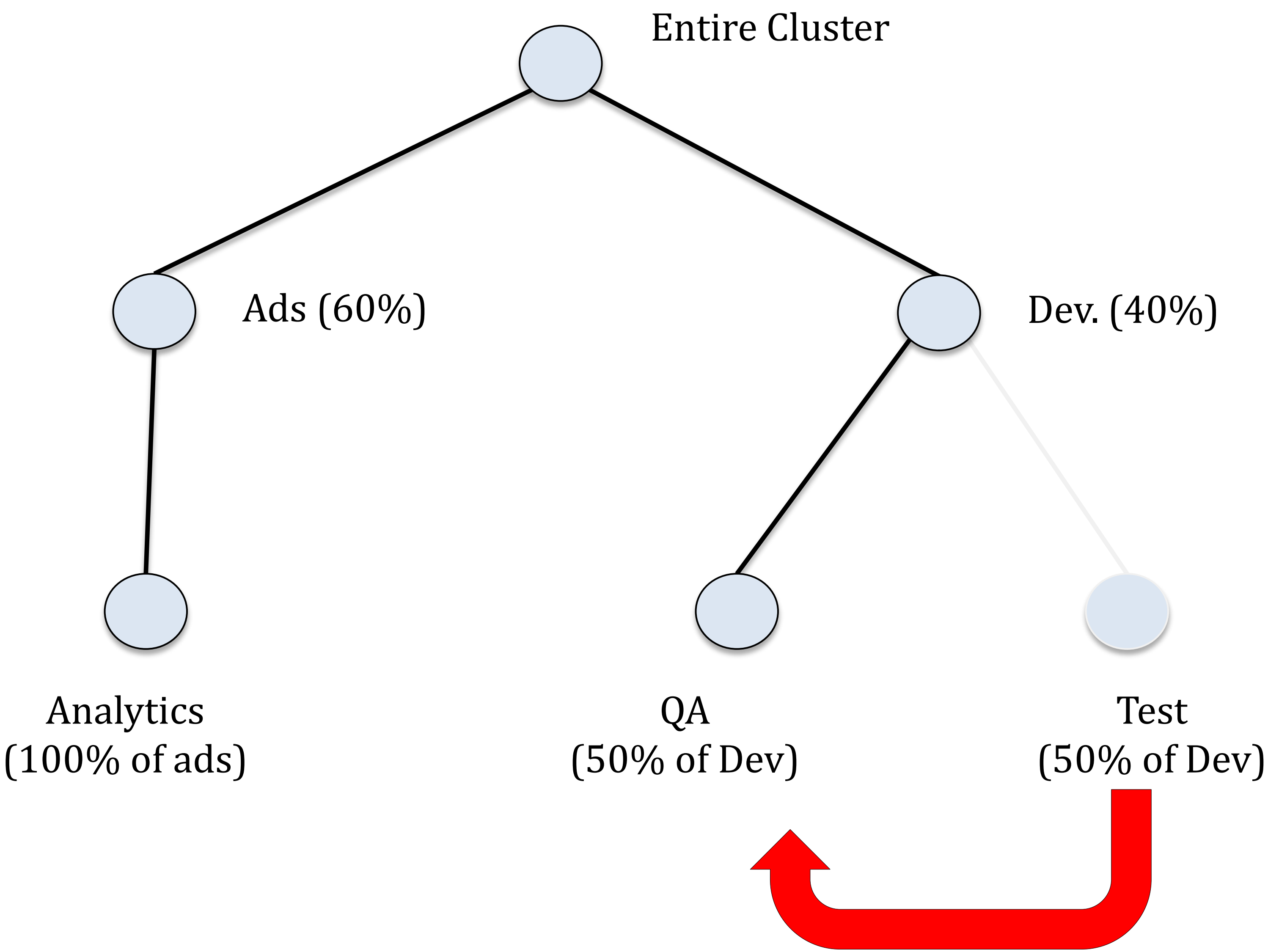
- Both industry implementations adapted DRF to support hierarchies

What's hierarchical scheduling?

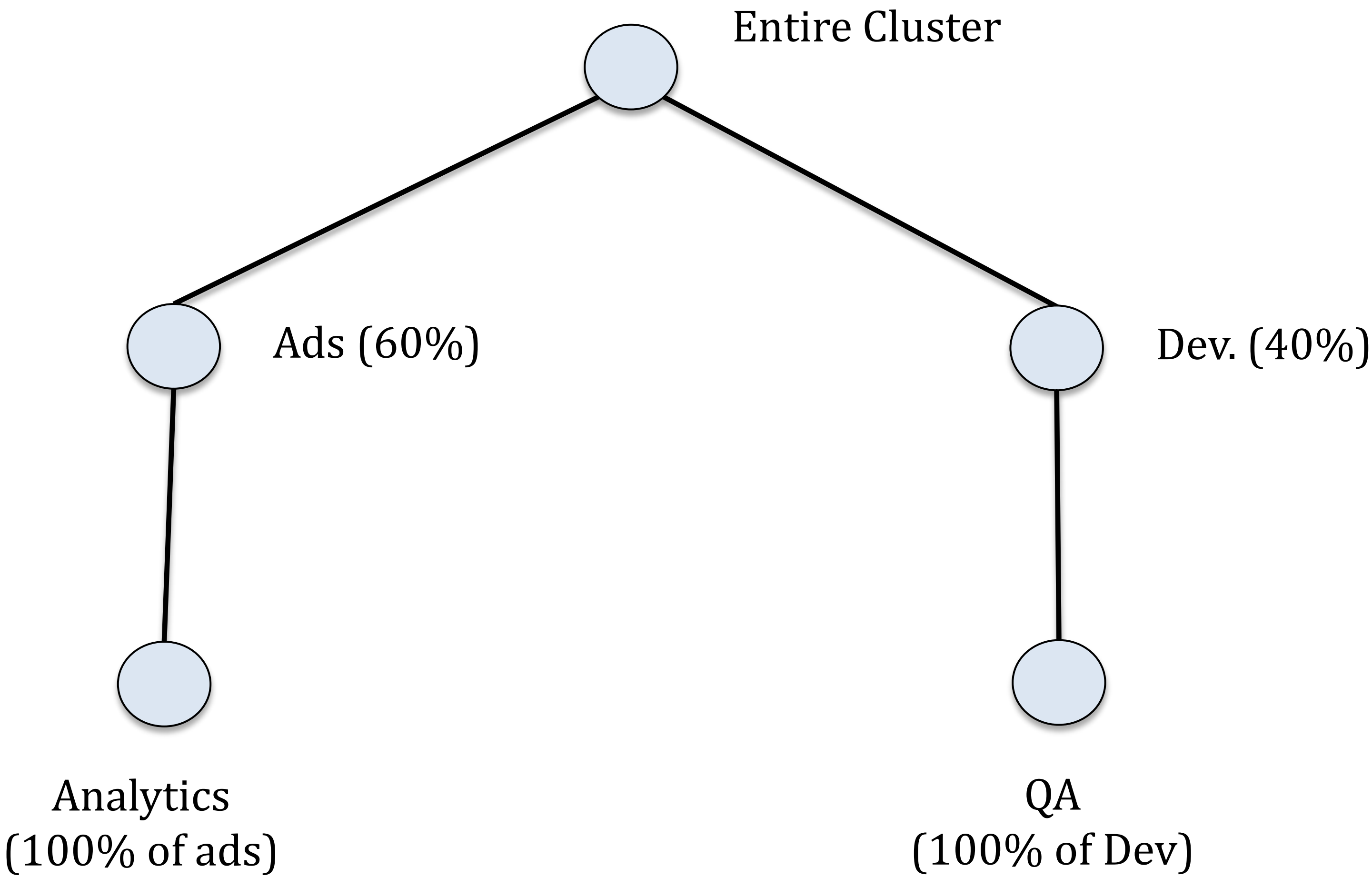
Hierarchical Scheduling



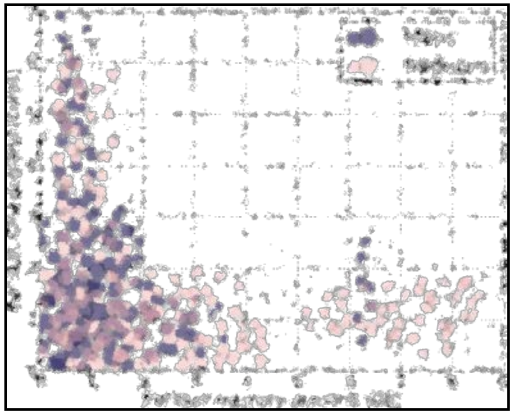
Hierarchical Scheduling



Hierarchical Scheduling

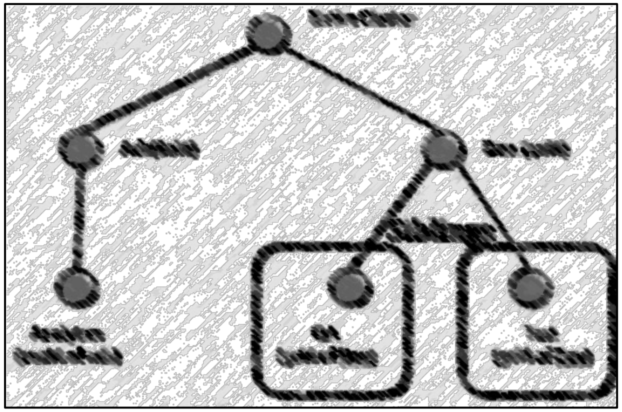


Multi-Resource Scheduling **Hierarchical Policies**



+

=



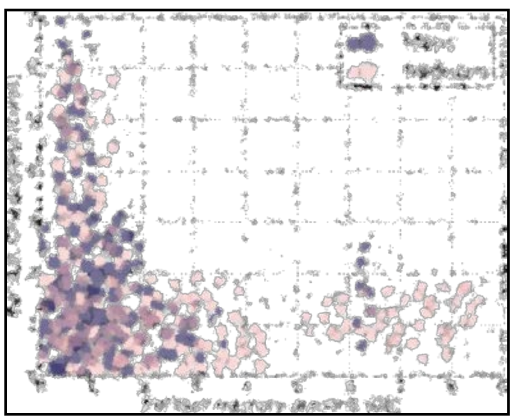
Challenging

- Hadoop DRF schedulers can break down
  - Leave resources unallocated, or
  - Starve some users

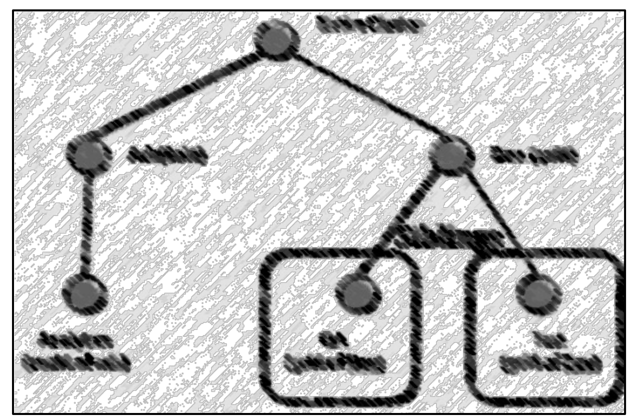
Problem Statement

How to generalize DRF to support hierarchical policies?

Dominant Resource Fairness **Hierarchical Scheduling**



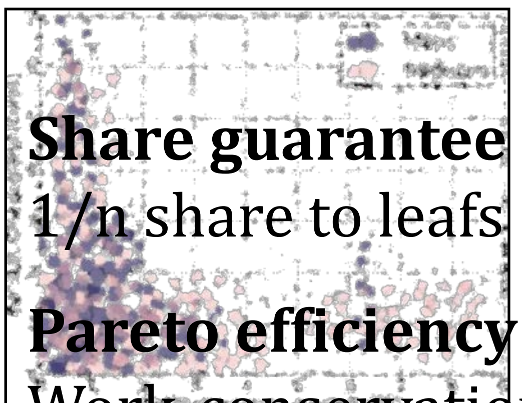
+



Problem Statement

How to generalize DRF to support hierarchical policies?

Dominant Resource Fairness



- **Share guarantee**
  1/n share to leafs
- **Pareto efficiency**
  Work-conservation

+ →

Hierarchical Scheduling

- **Hierarchical share guarantee**
  1/n to every node
- **Pareto efficiency**
  Work-conservation

Outline

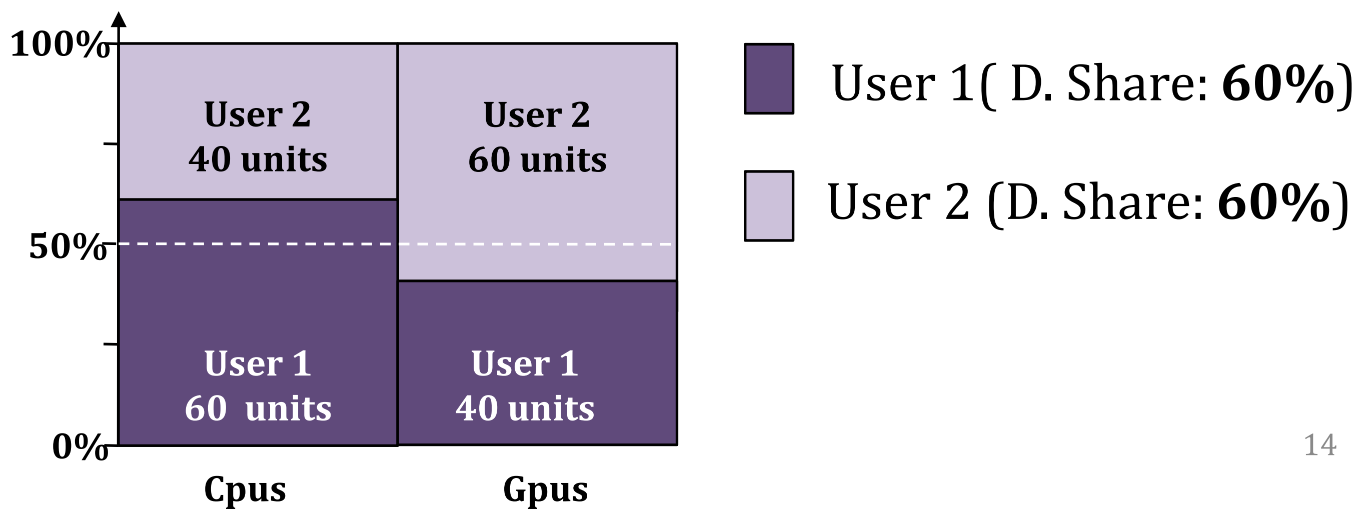
- How to schedule multi-resources? (DRF)
- Why is it challenging?
- What's our solution? (H-DRF)
- How well does it work?

Dominant Resource Fairness (DRF)

- Dominant resource of a user is the resource she has biggest share of
  - Dominant share of a user is her share of her dominant resource

Total resources:	**<100 Cpus, 100 Gpus>**	(2 types of resources)
User 1 demand:	**<3 Cpus, 2 Gpus>**	dom res: **Cpu**
User 2 demand:	**<2 Cpus, 3 Gpus>**	dom res: **Gpu**

- DRF Scheduler
  - Max-min fair allocation on dominant shares
  - ”Equalize” the dominant share of all users

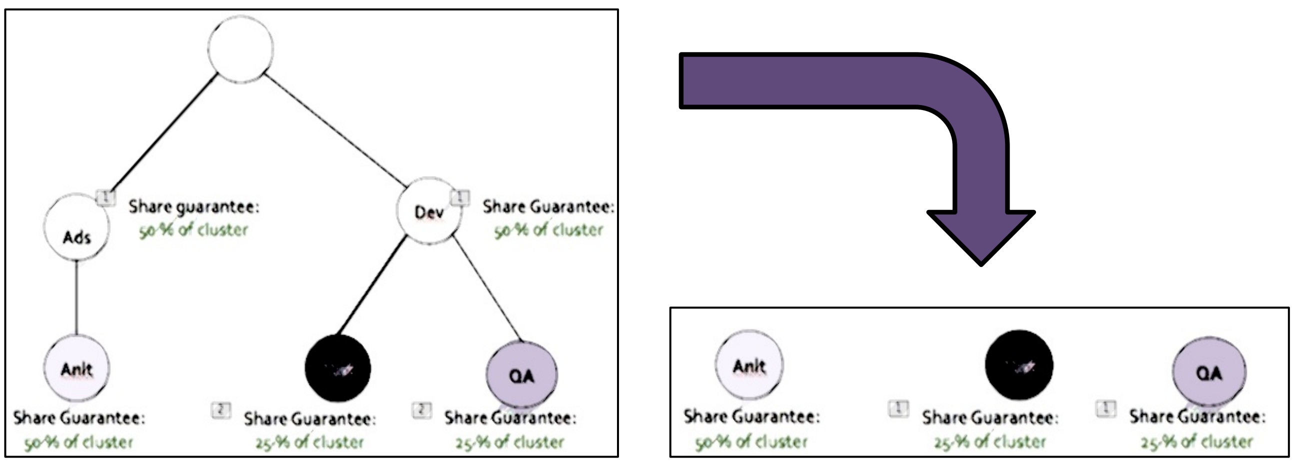


Outline

- How to schedule multi-resources? (DRF)
- Why is it challenging?
- What's our solution? (H-DRF)
- How well does it work?

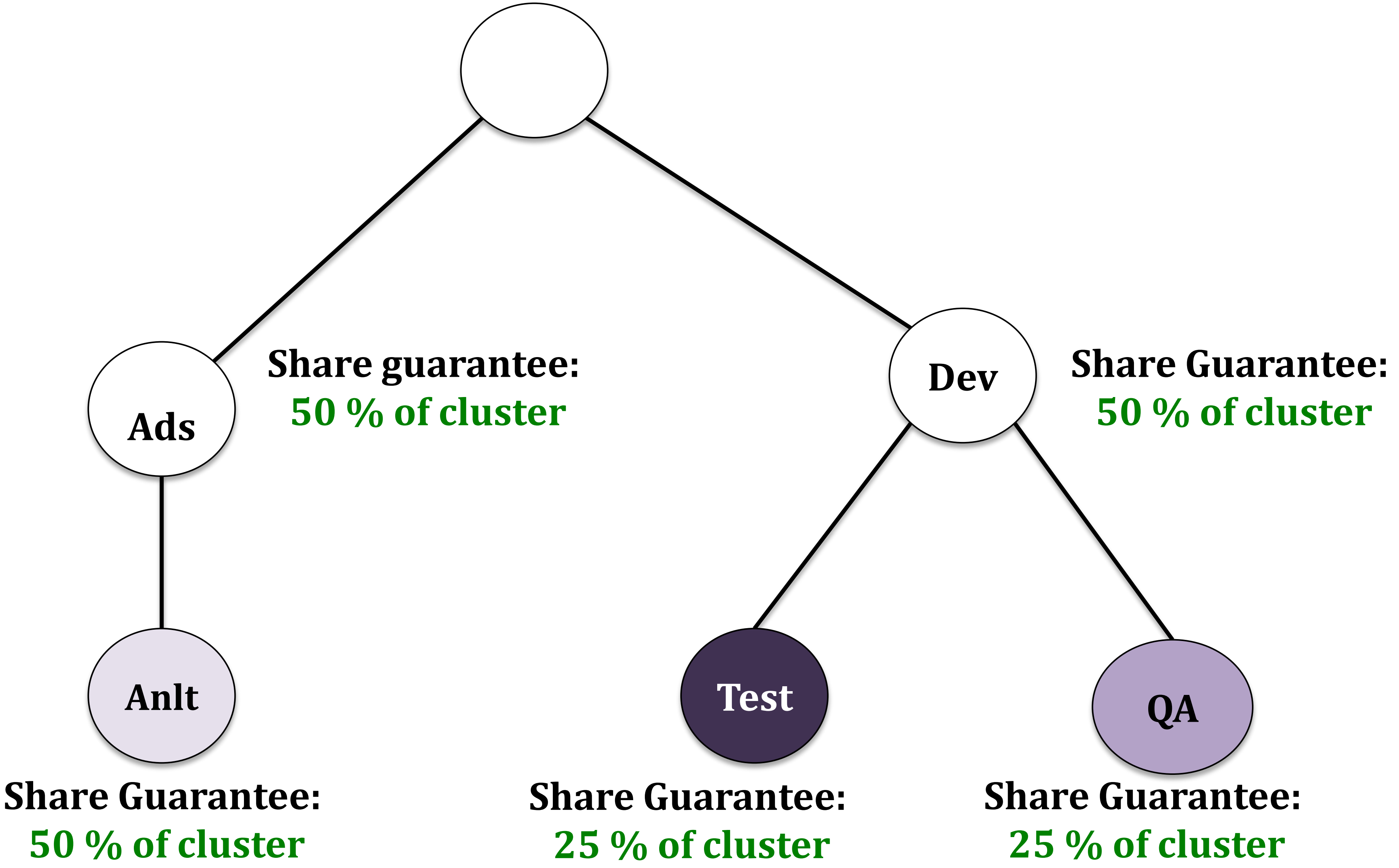
Hierarchy Flattening

- General technique
  - Compute fair share of every leaf node
  - Use weighted scheduler (weighted DRF)

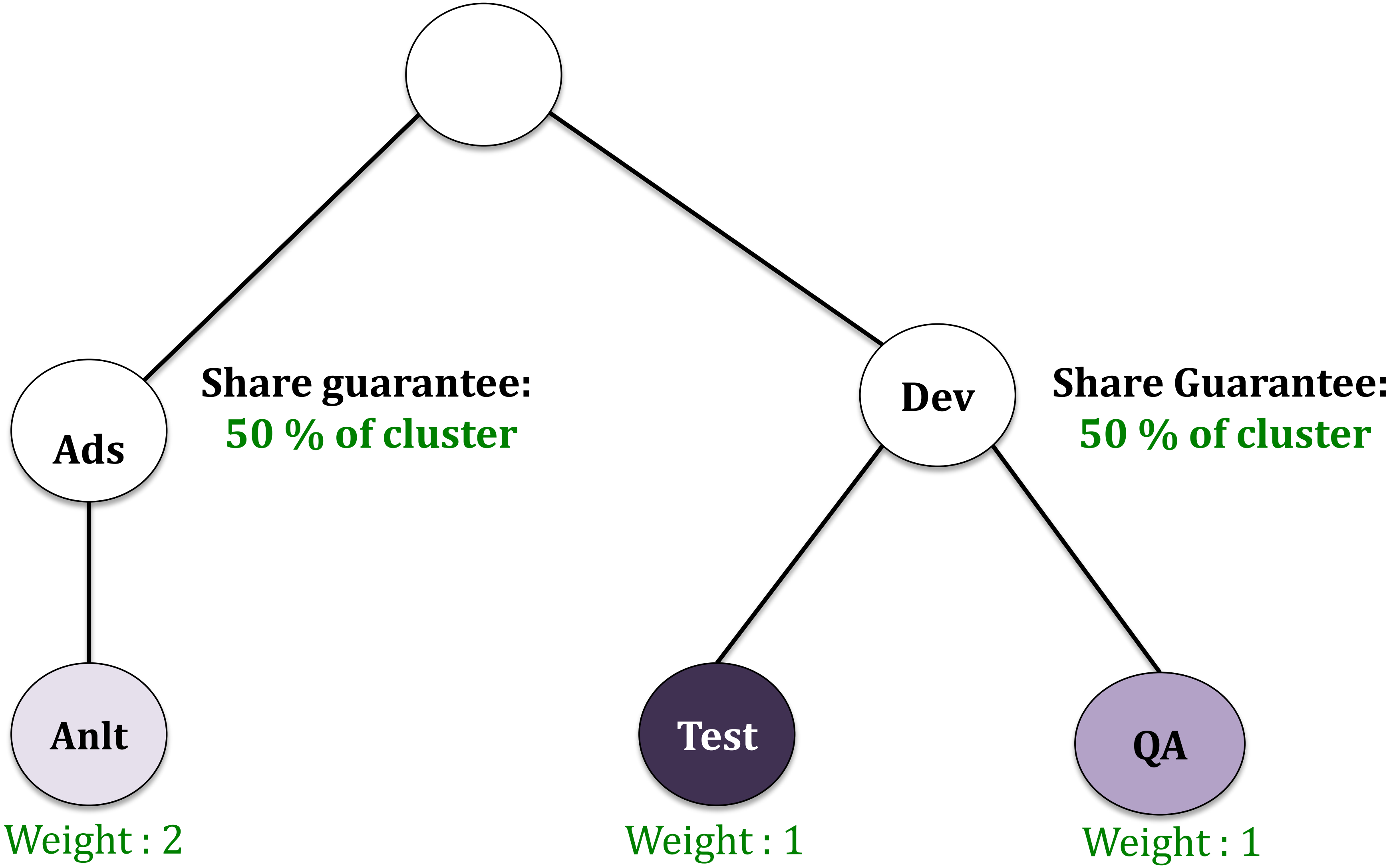


- Works for any single-resource scheduler

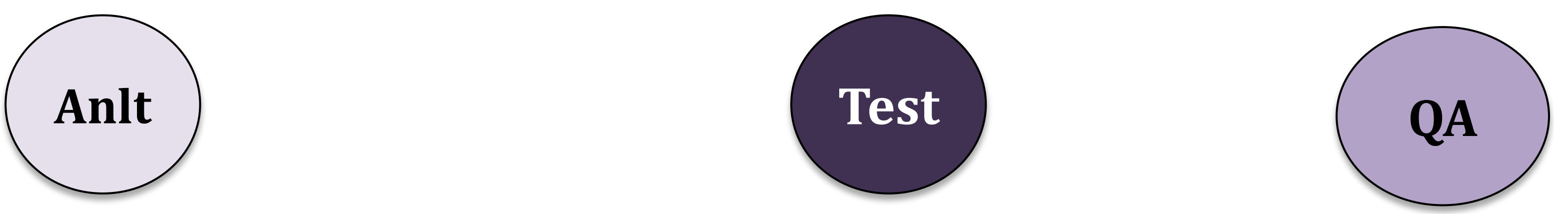
Hierarchy Flattening



Hierarchy Flattening



Total resources: <100 Cpus, 100 Gpus>

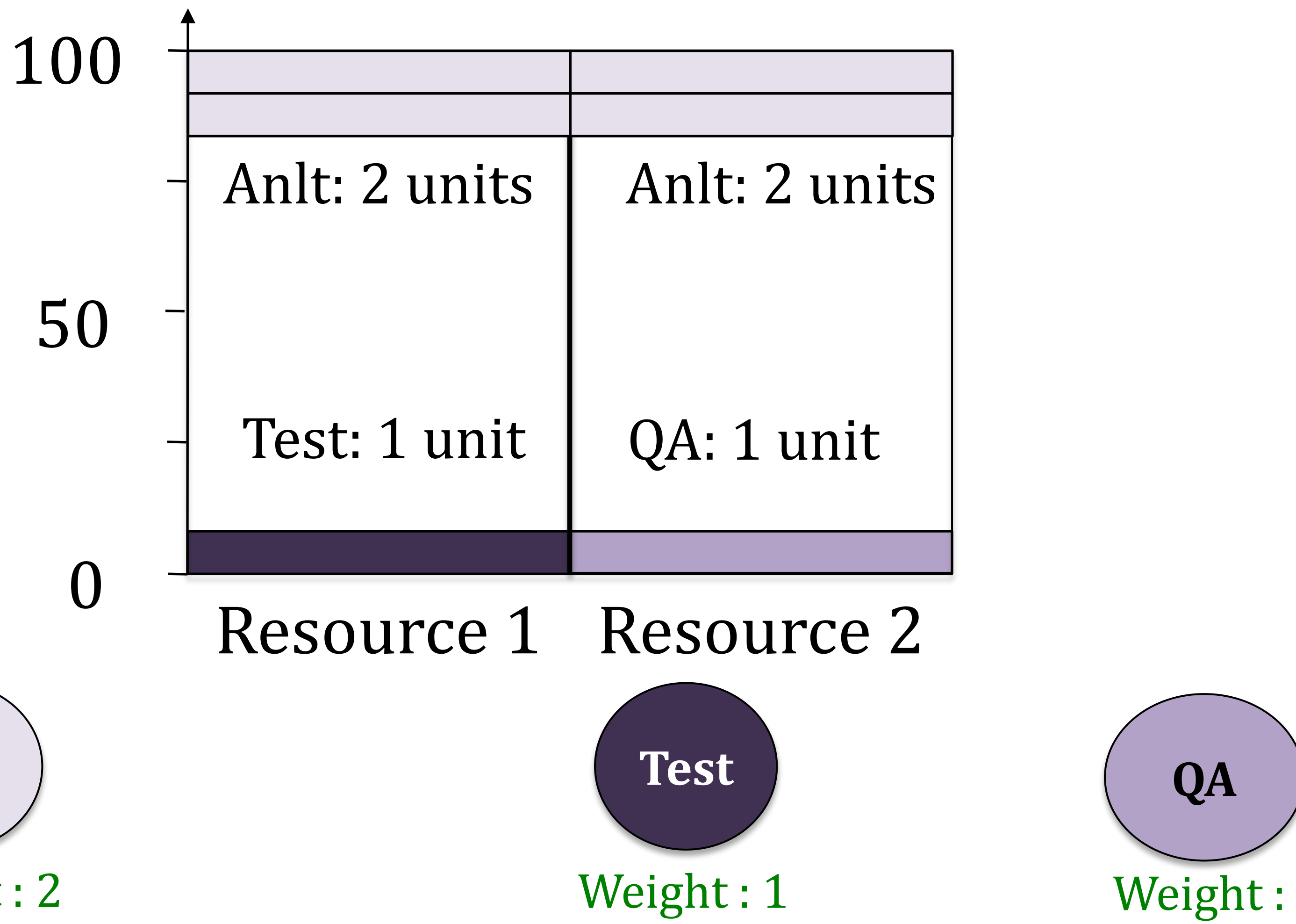


Weight : 2
Demand: <1, 1>

Weight : 1
Demand: <1,0>

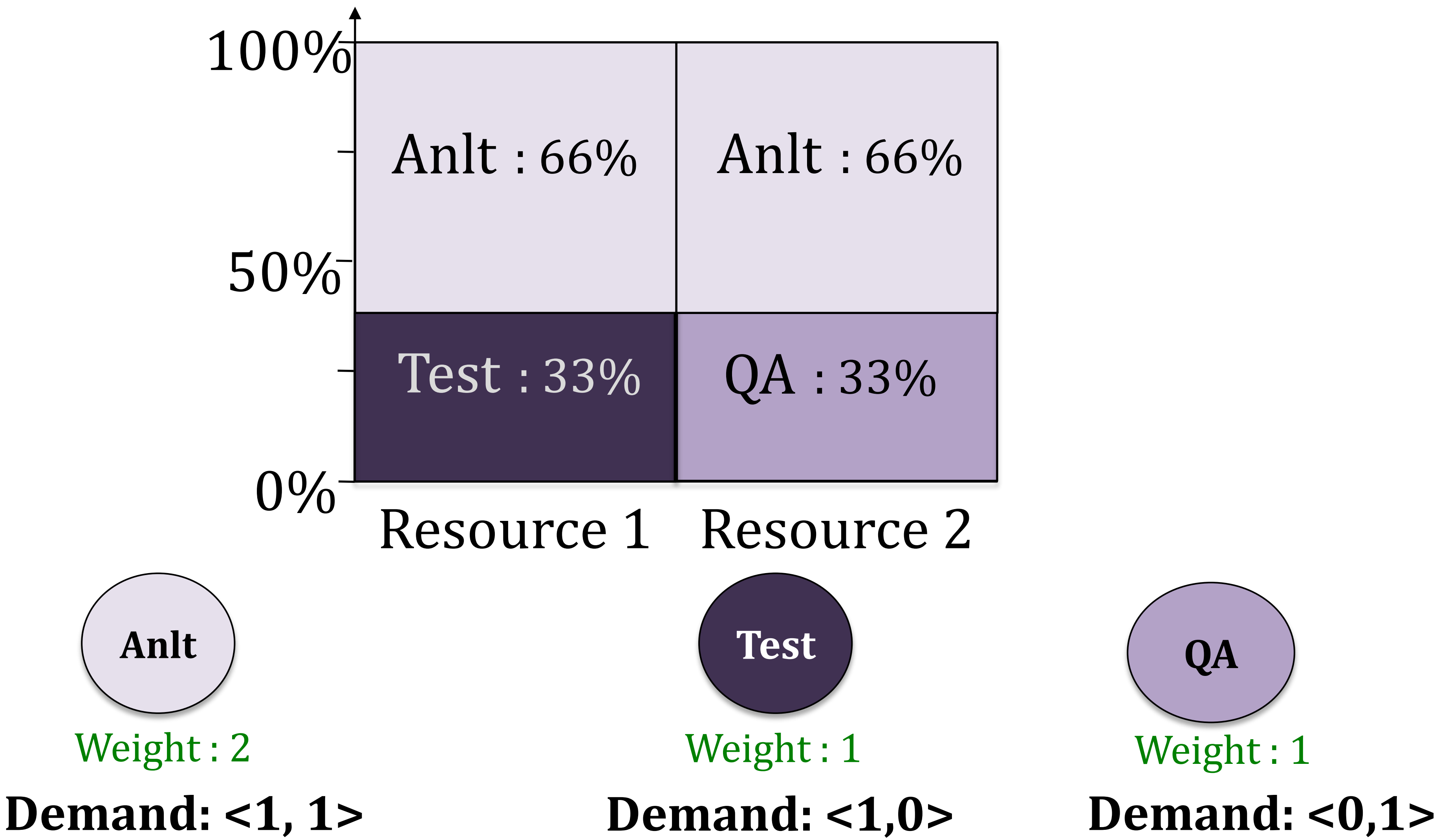
Weight : 1
Demand: <0,1>

Initial Allocation

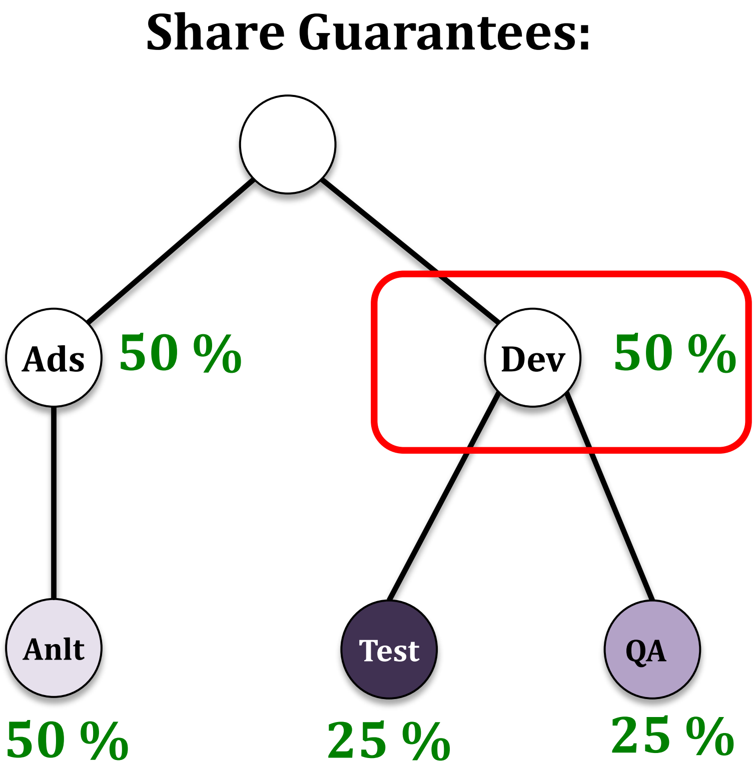


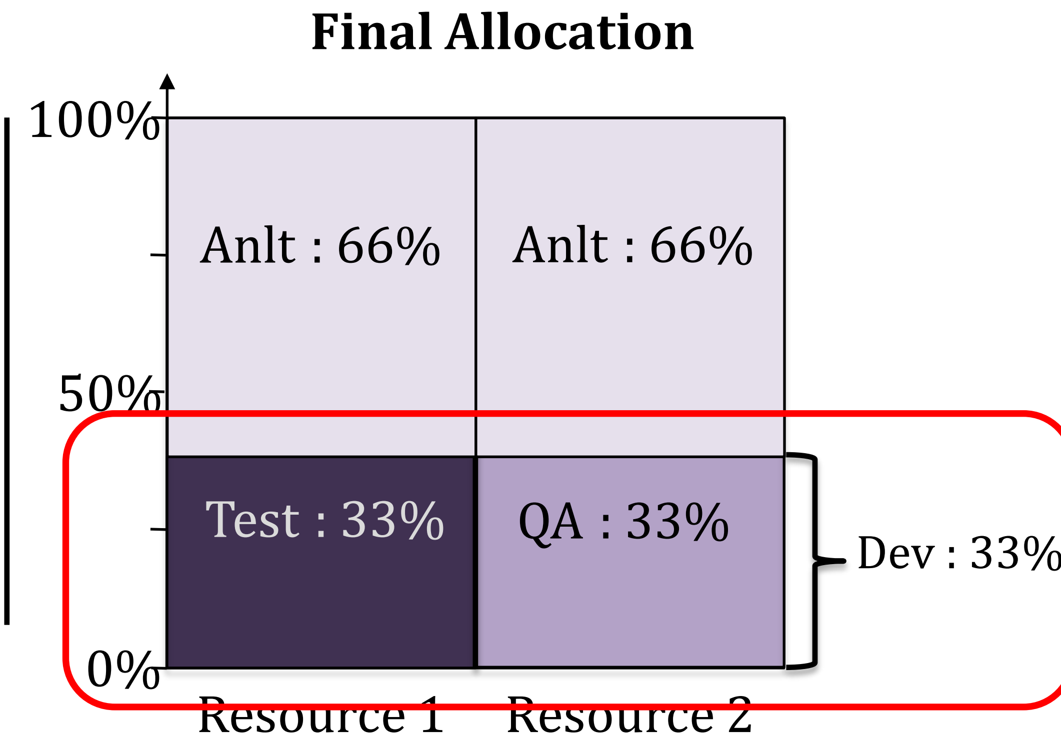
Demand: <1, 1> **Demand: <1,0>** **Demand: <0,1>**

Final Allocation



Hierarchical Share Guarantee Violated



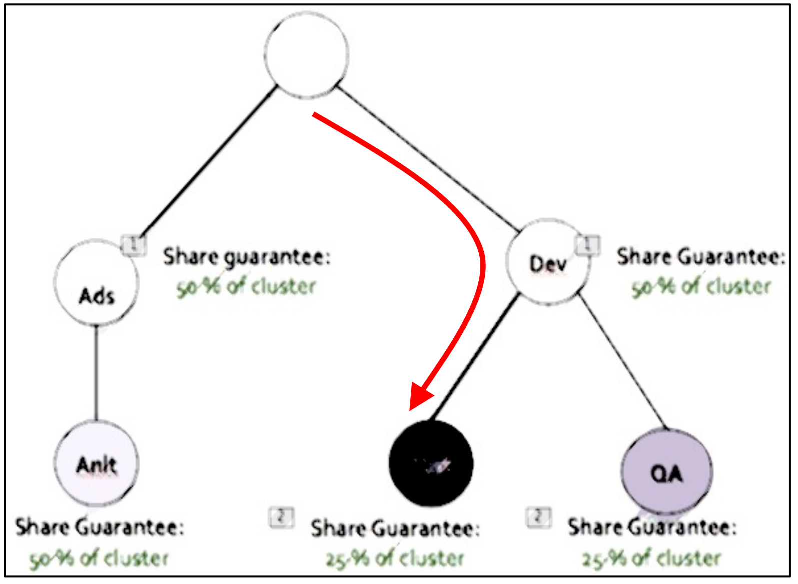


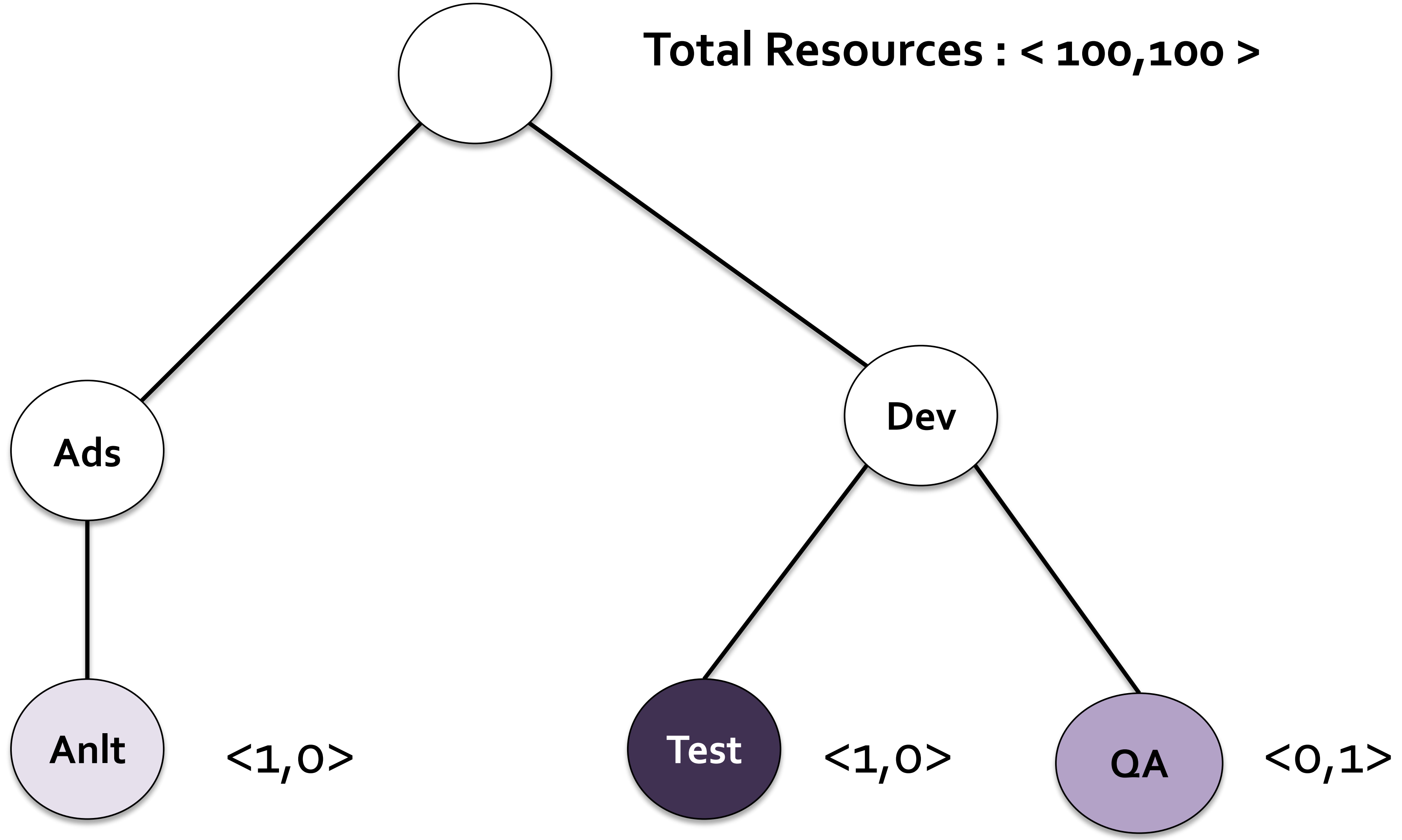
Outline

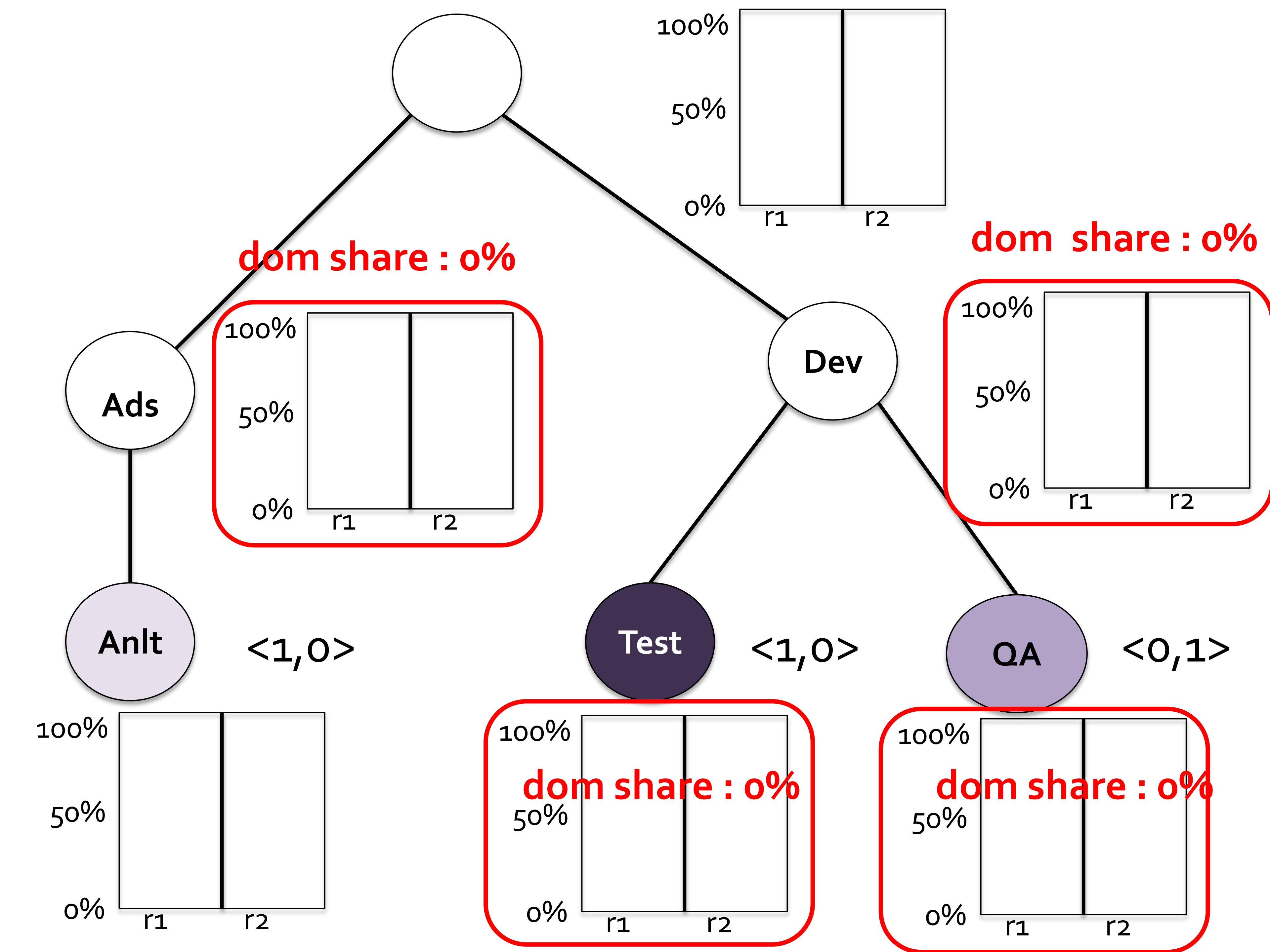
- How to schedule multi-resources? (DRF)
- Why is it challenging?
- What's our solution? (H-DRF)
- How well does it work?

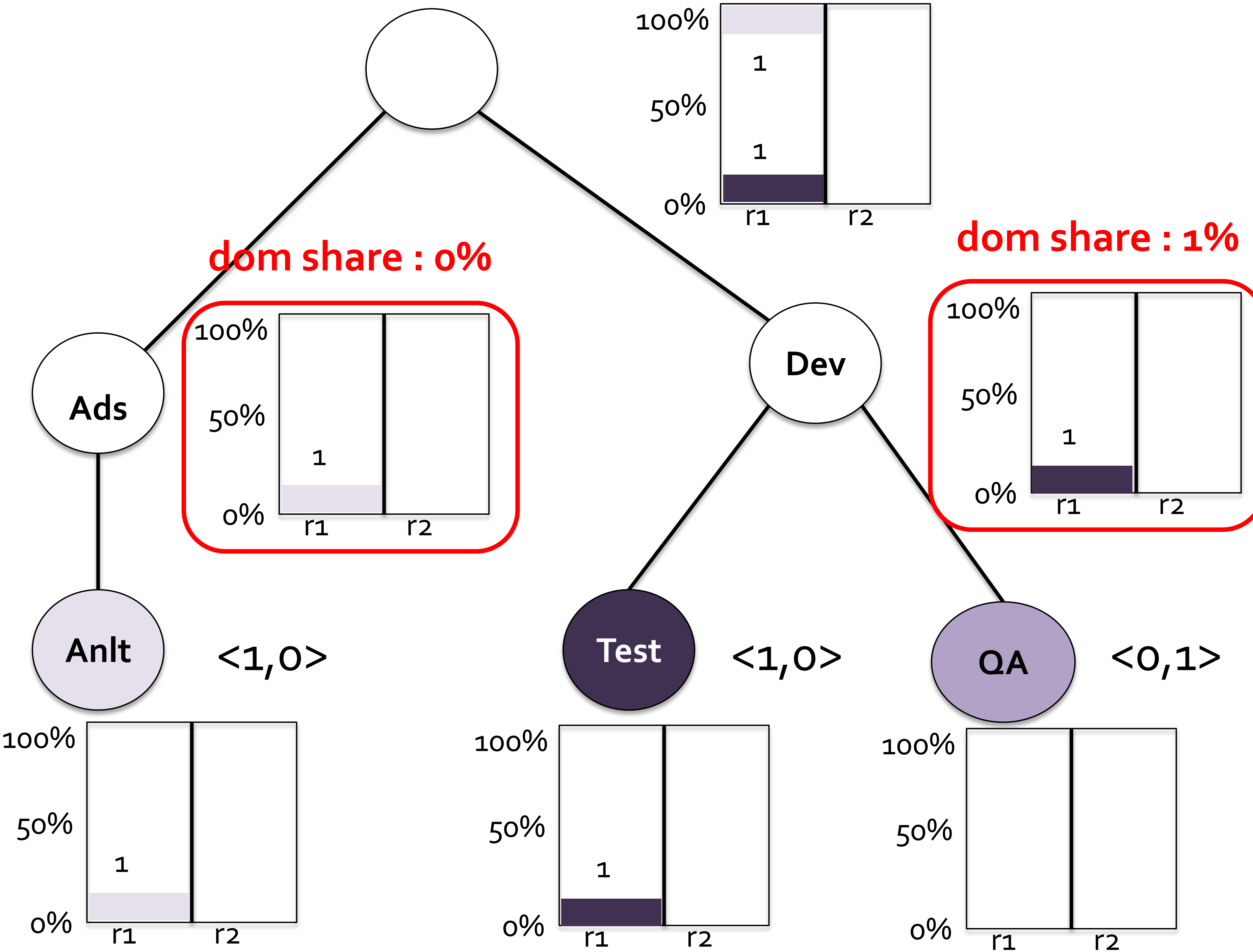
Static H-DRF

- Traverse tree top to bottom
  - Recursively pick node with smallest dom. share
  - Top-down equalize siblings

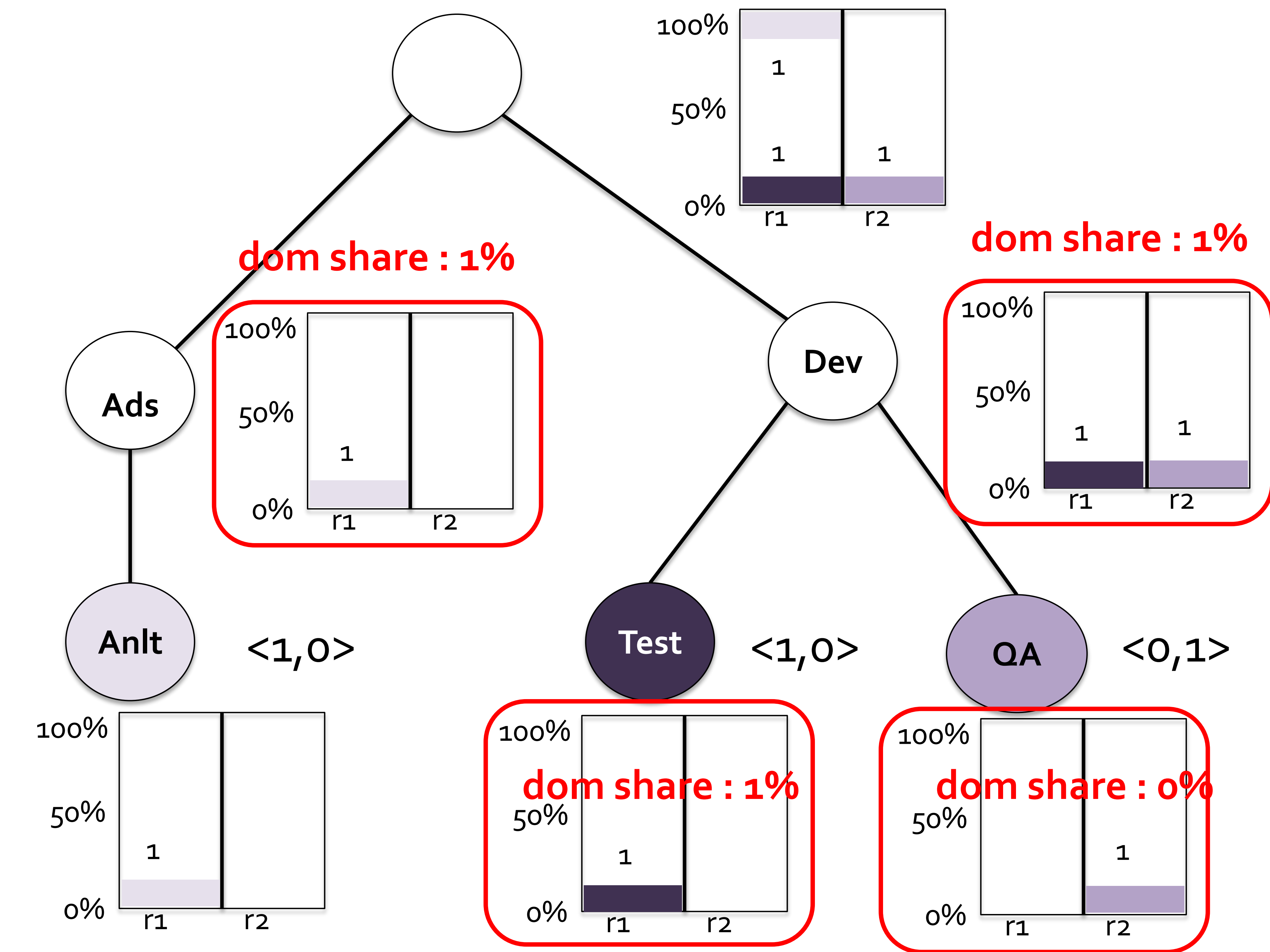



Total Resources : < 100,100 >
Ads
Dev
Anlt
<1,0>
Test
<1,0>
QA
<0,1>

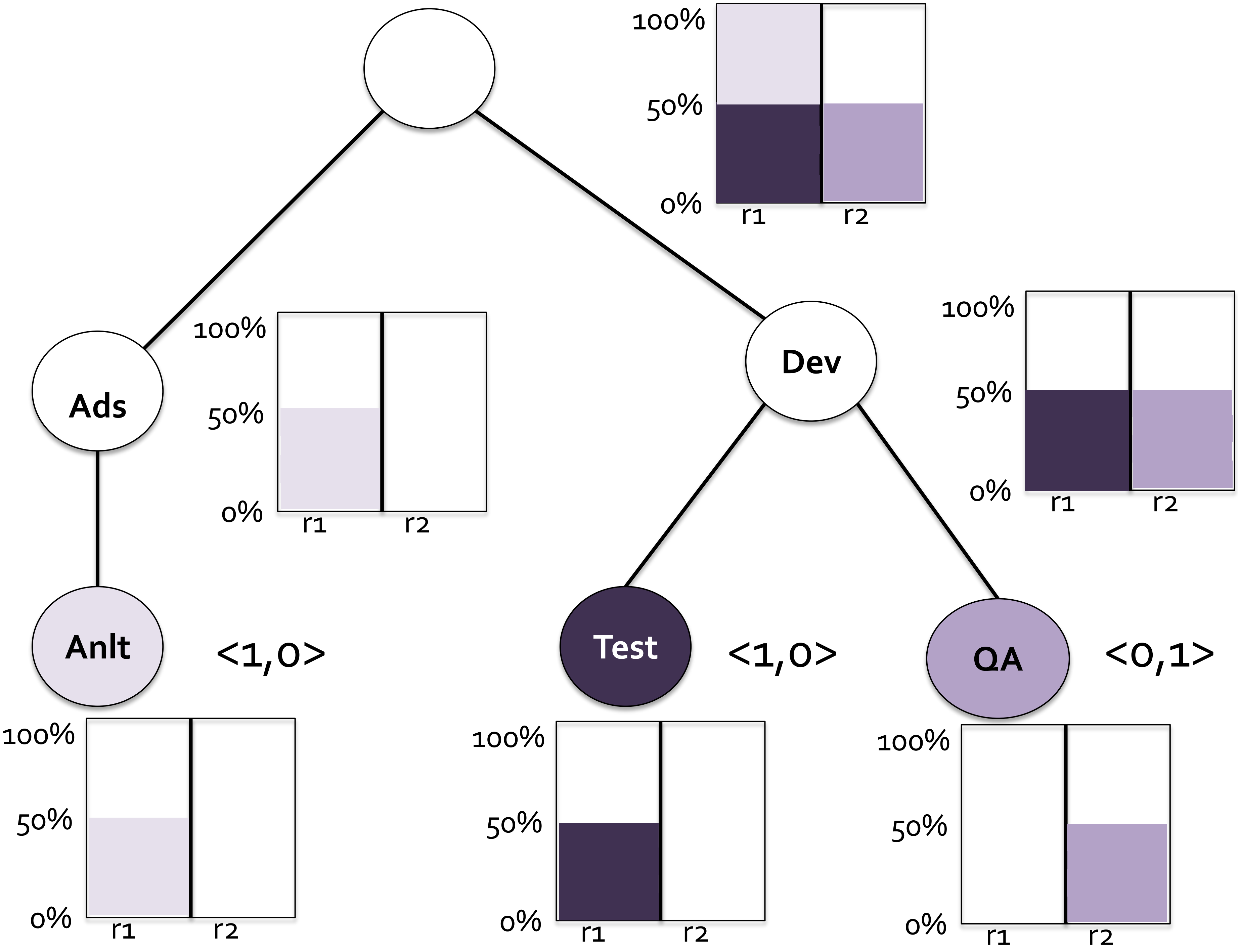

100%
50%
0%
r1
r2
dom share : 0%
Ads
100%
50%
0%
r1
r2
Dev
dom share : 0%
100%
50%
0%
r1
r2
Anlt
<1,0>
Test
<1,0>
QA
<0,1>
100%
50%
0%
r1
r2
100%
dom share : 0%
50%
0%
r1
r2
100%
dom share : 0%
50%
0%
r1
r2



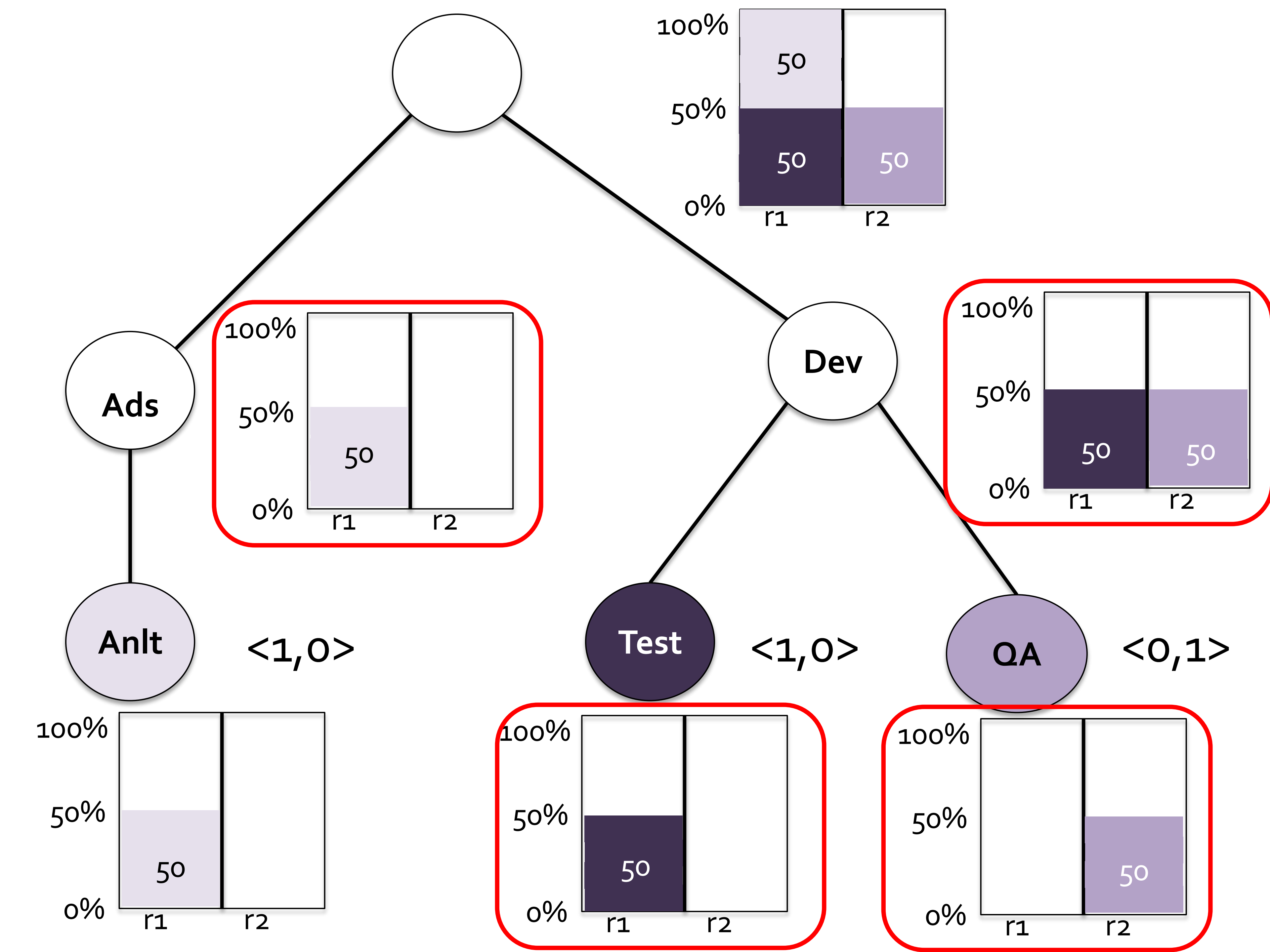
100%
50%
0%
1
1
r1
r2
dom share : 0%
dom share : 1%
Ads
Dev
Anlt
Test
QA
<1,0>
<1,0>
<0,1>

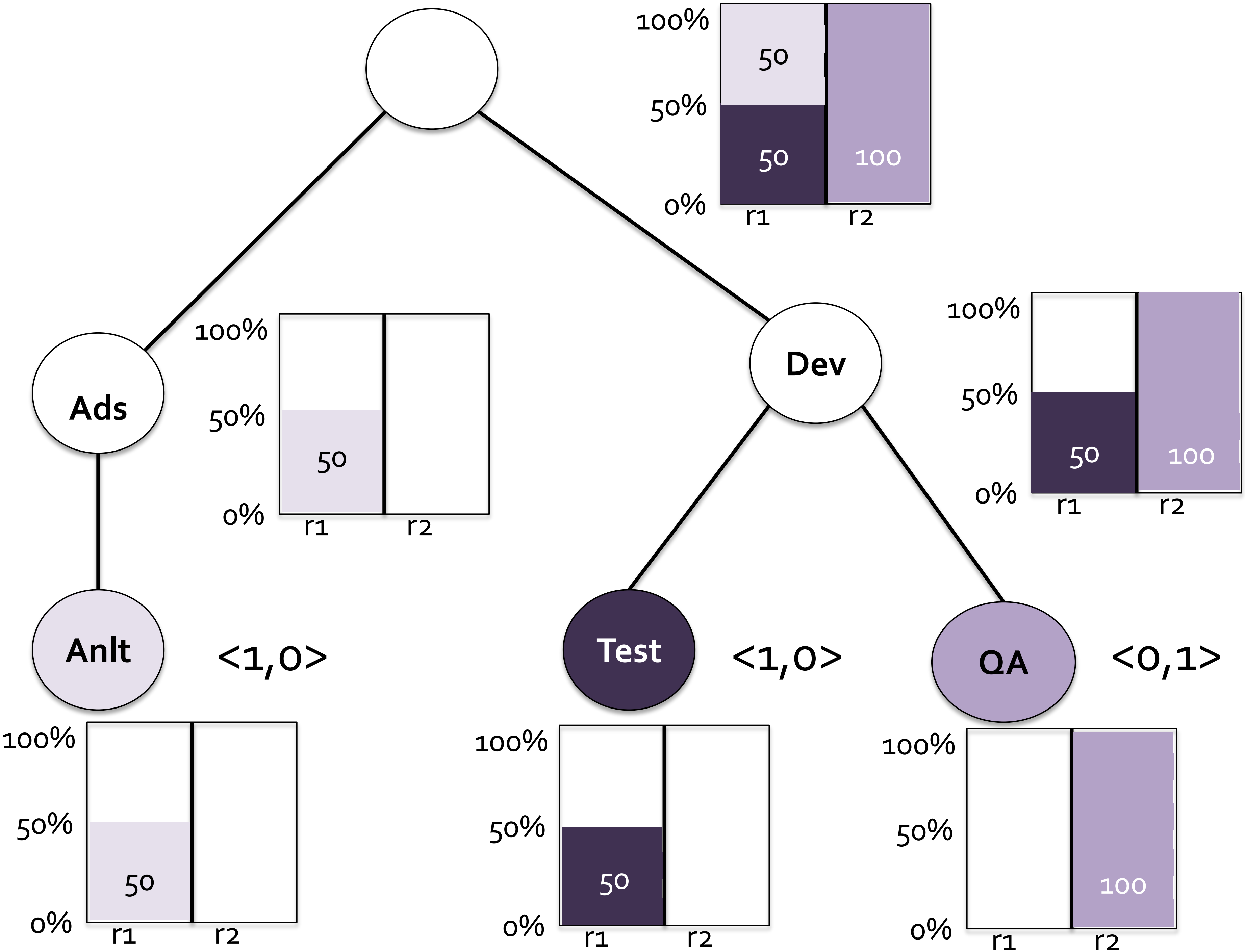


100%
50%
0%
1
1
1
r1
r2
dom share : 1%
Ads
Dev
100%
50%
0%
1
r1
r2
dom share : 1%
100%
50%
0%
1
1
r1
r2
Anlt
<1,0>
Test
<1,0>
QA
<0,1>
100%
50%
0%
1
r1
r2
dom share : 1%
100%
50%
0%
1
r1
r2
dom share : 0%
100%
50%
0%
1
r1
r2



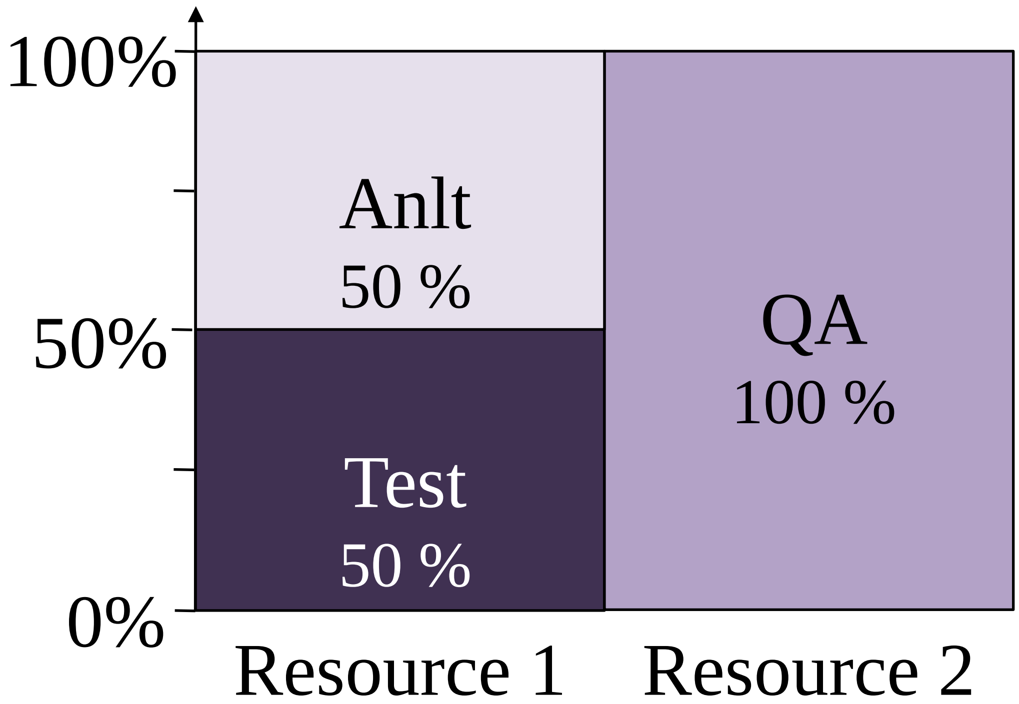
100%
50%
0%
r1
r2
Ads
Dev
Anlt
<1,0>
Test
<1,0>
QA
<0,1>

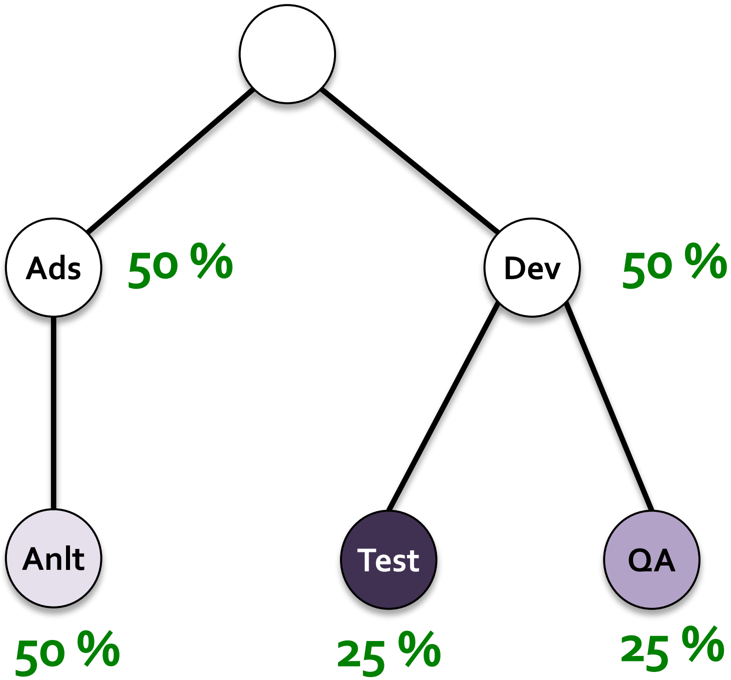

100%
50%
0%
50
50
50
r1
r2
Ads
100%
50%
0%
50
r1
r2
Dev
100%
50%
0%
50
50
r1
r2
Anlt
<1,0>
Test
<1,0>
QA
<0,1>
100%
50%
0%
50
r1
r2
100%
50%
0%
50
r1
r2
100%
50%
0%
50
r1
r2



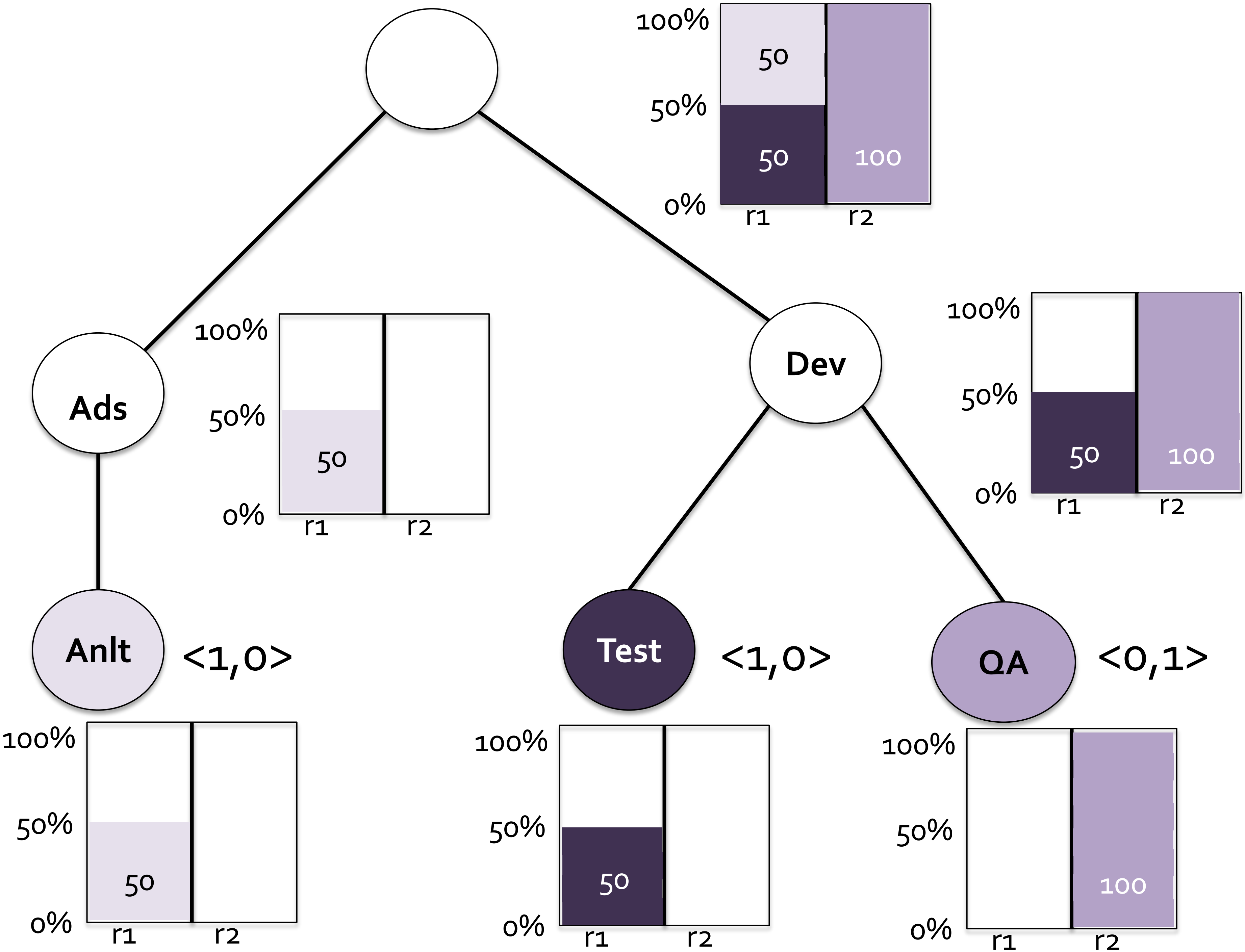
100%
50%
0%
50
50
100
r1
r2
Ads
Dev
100%
50%
0%
50
r1
r2
100%
50%
0%
50
100
r1
r2
Anlt
<1,0>
Test
<1,0>
QA
<0,1>
100%
50%
0%
50
r1
r2
100%
50%
0%
50
r1
r2
100%
50%
0%
100
r1
r2

Ideal DRF allocations in hierarchy

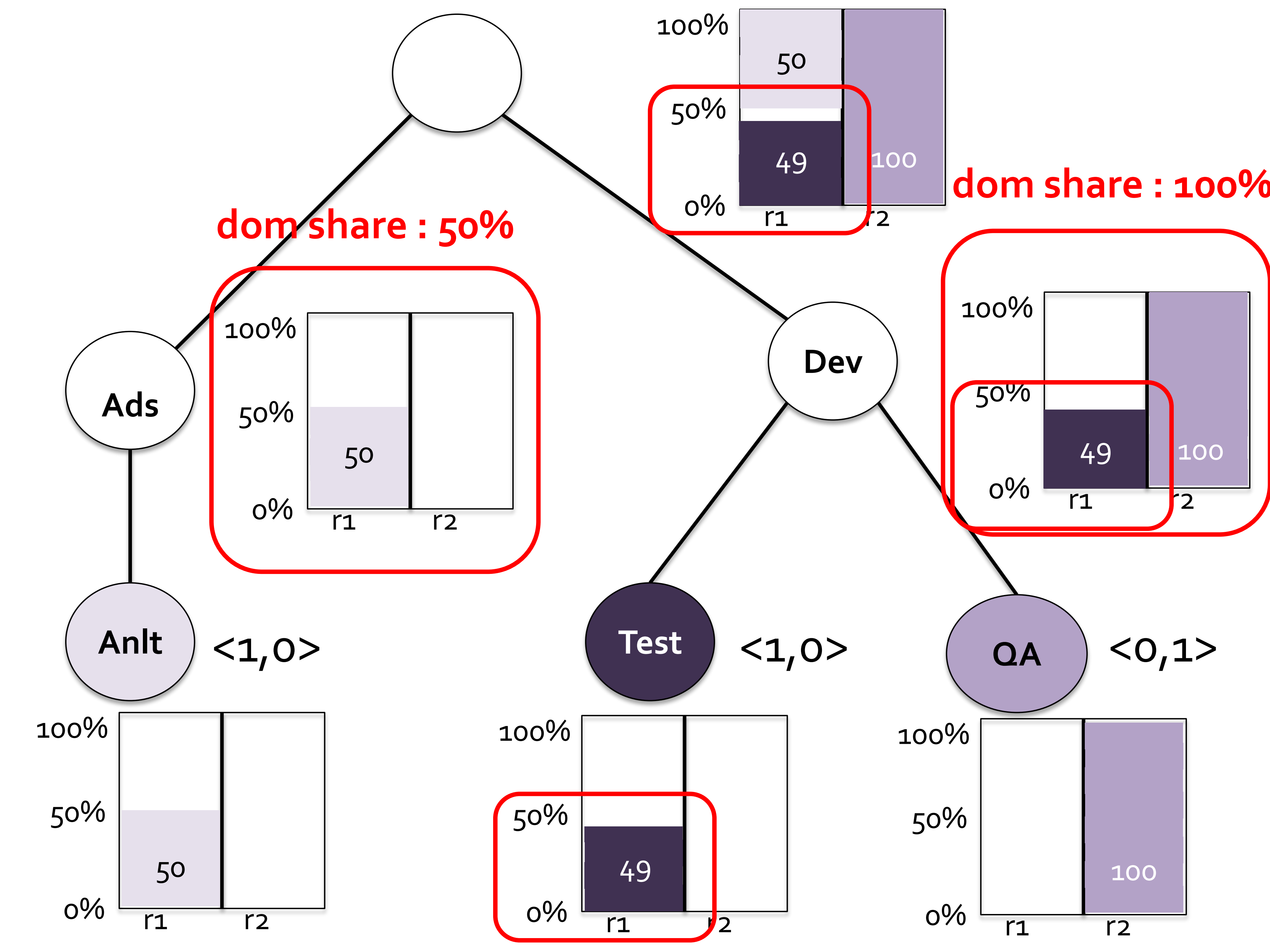




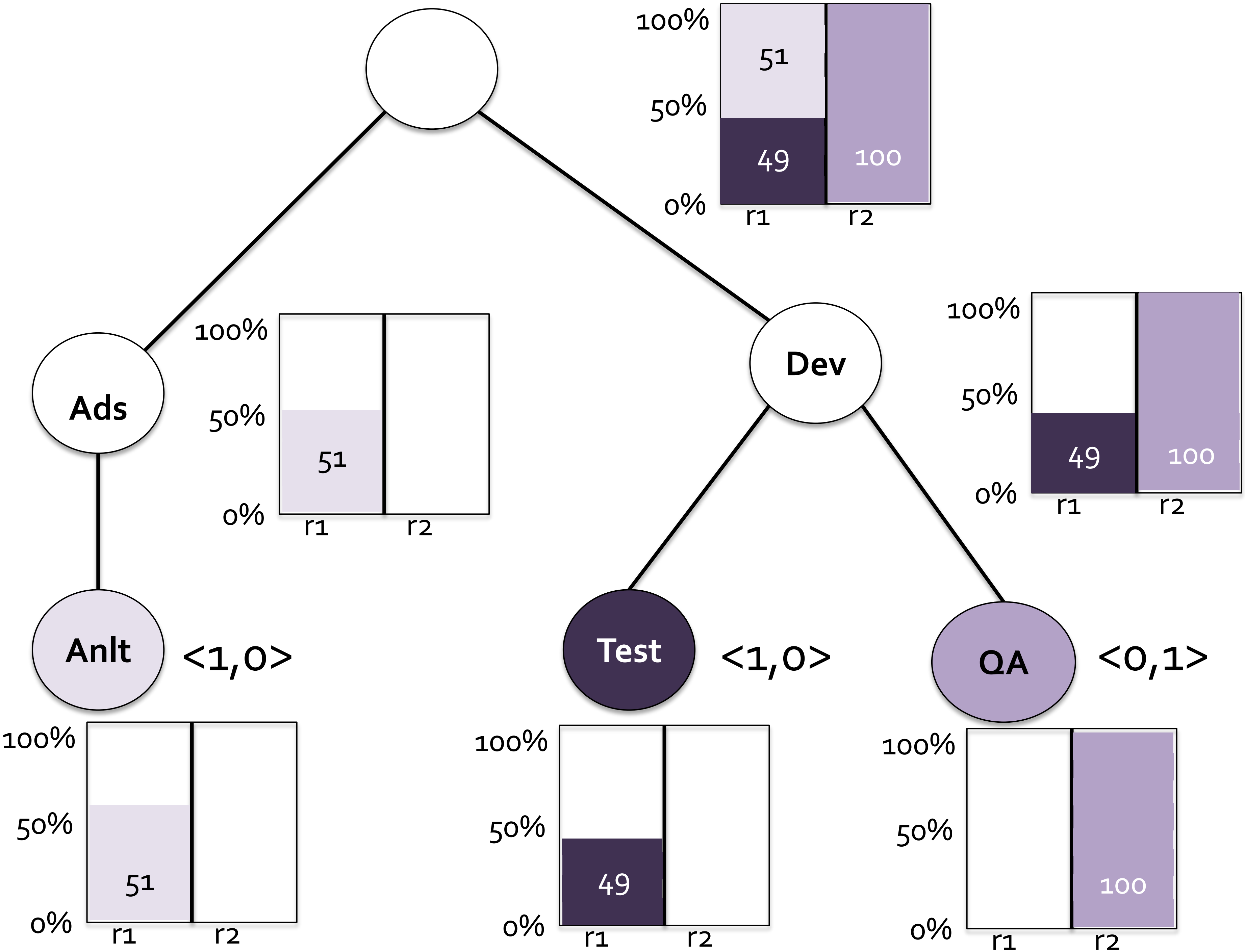
Hierarchical share guarantees for every node



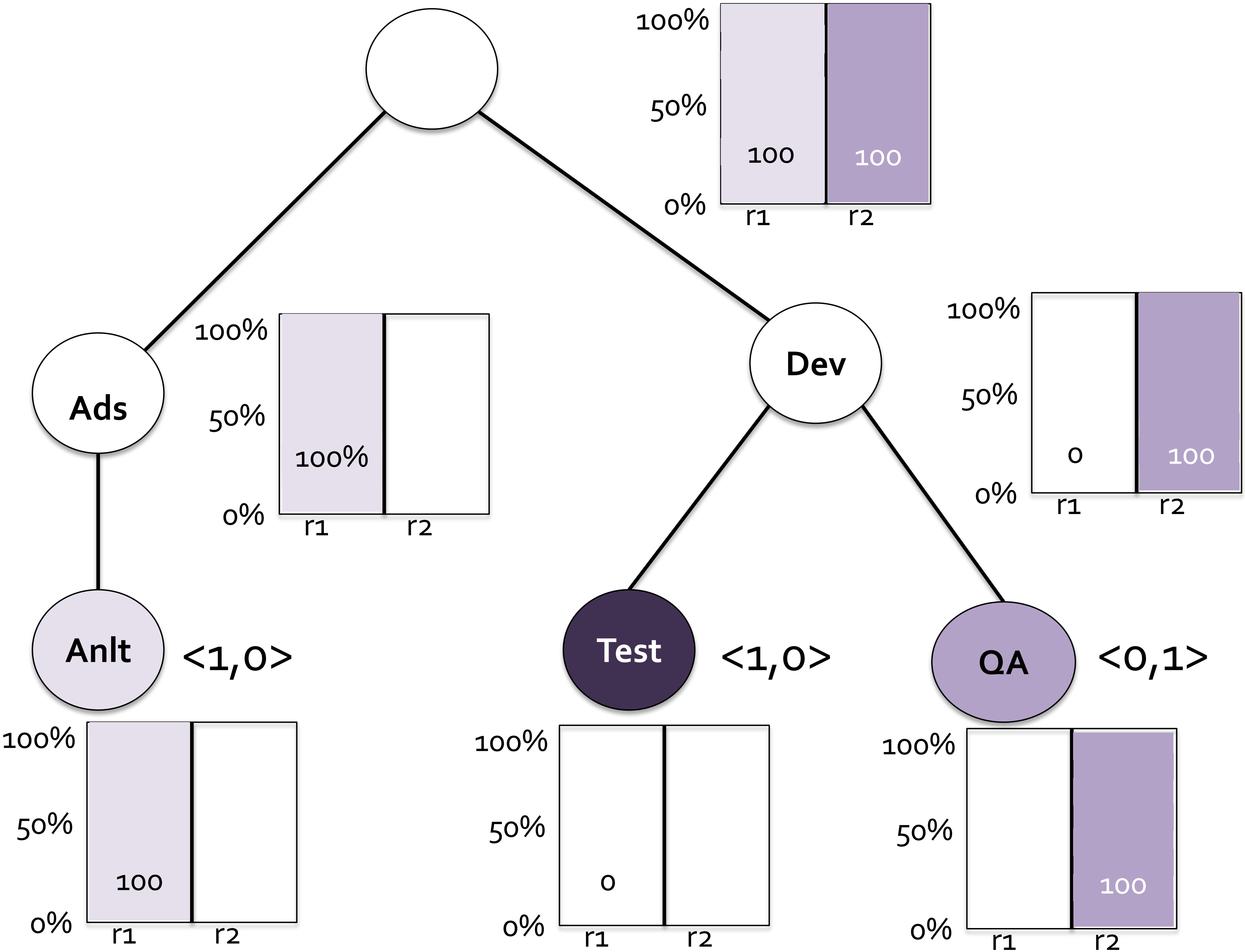
100%
50%
0%
50
50
100
r1
r2
Ads
Dev
Anlt
<1,0>
Test
<1,0>
QA
<0,1>



100%
50
50%
49
100
0%
r1
r2
dom share : 100%
dom share : 50%
100%
50%
50
0%
r1
r2
Ads
Dev
100%
50%
49
100
0%
r1
r2
Anlt
<1,0>
Test
<1,0>
QA
<0,1>
100%
50%
50
0%
r1
r2
100%
50%
49
0%
r1
r2
100%
50%
100
0%
r1
r2

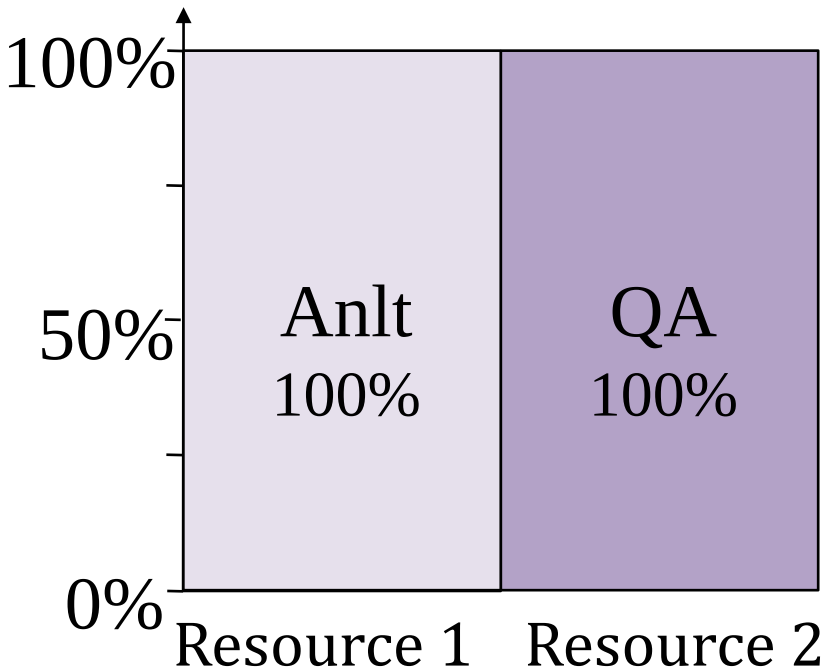


100%
50%
0%
51
49
100
r1
r2
Ads
Dev
Anlt
Test
QA
<1,0>
<1,0>
<0,1>

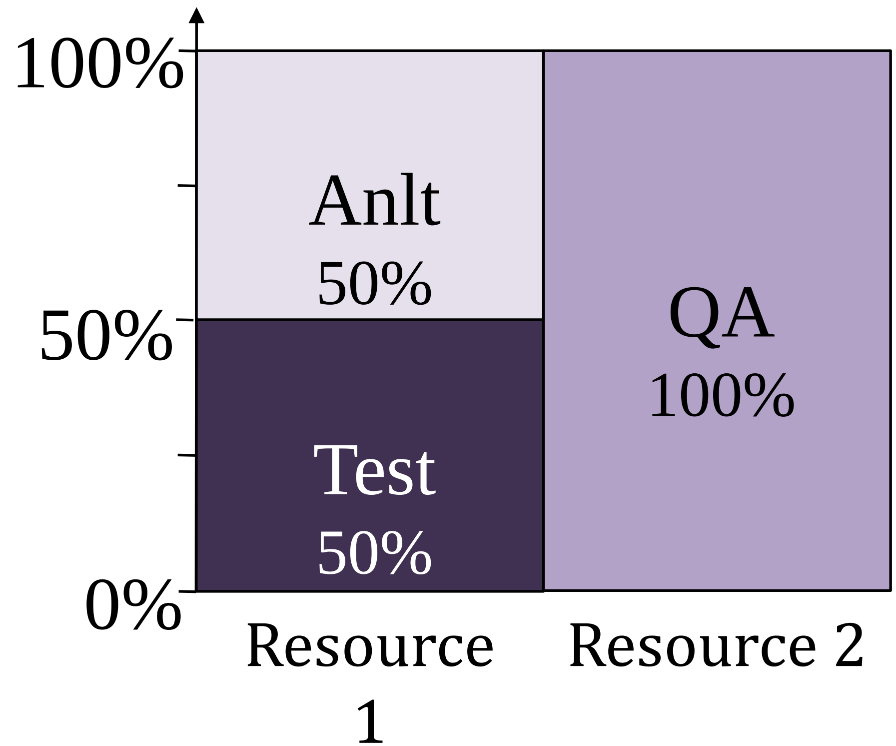

100%
50%
0%
100
100
r1
r2
Ads
100%
50%
0%
100%
r1
r2
Dev
100%
50%
0%
0
100
r1
r2
Anlt
<1,0>
Test
<1,0>
QA
<0,1>
100%
50%
0%
100
r1
r2
100%
50%
0%
0
r1
r2
100%
50%
0%
100
r1
r2

Starvation

Allocation in a dynamic cluster



Ideal allocation

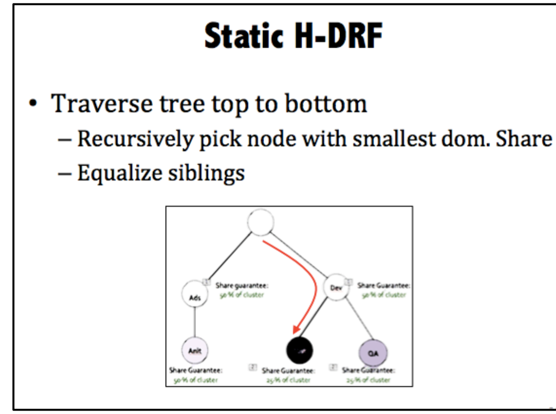


Outline

- How to schedule multi-resources? (DRF)
- Why is it challenging?
- What's our solution? (H-DRF)
- How well does it work?

Hierarchical DRF (H-DRF)

- Leverage Static H-DRF



- Add two invariants
  - Re-scale consumption vectors
  - Ignore terminated/blocked nodes

$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ current consumed resources
W resources to allocate ▷ Assumption: $R - C > W$
Y set of nonzero resources in W
A (demanding), set of leaf nodes that use only resources in Y or parents of demanding nodes
n_r ▷ root node in hierarchy tree
$C(n)$ ▷ children of any node n
$s_i \ \ (i = 1 \ldots n)$ ▷ dominant shares
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ \ (i = 1 \ldots n)$ ▷ "scaled" resources
Recompute s**:** $UpdateS(n_r)$
Allocate the resources: Alloc(W)

function (recursive) *UpdateS*(n_i)
if n_i is a leaf node **then**
  $s_i = \max U_{ij}/R_j$ for $j \in Y$
  return U_i
else
  Q = set of U_j's from *UpdateS*(n_j) for children of n_i
  f = maximum dominant share from Q restricting to nodes in A and resources in Y
  Rescale demanding vectors in Q by f
  U_i = sum of vectors in Q
  $s_i = \max U_{i,j}/Rj$ for $j \in Y$
return U_i

function *Alloc*(W)
$n_i = n_r$
while n_i is not a leaf node (job) **do**
  n_j = node with lowest dominant share s_j in $C(n_i)$, which also has a task in its subtree that can be scheduled using W
  $n_i = n_j$
$D_i = \frac{W_i}{\max_j\{T_{i,j}\}} T_i$, s.t. T_i is n_i's task demand vector
$C = C + D_i$ ▷ update consumed vector
$U_i = U_i + D_i$ ▷ update leaf only

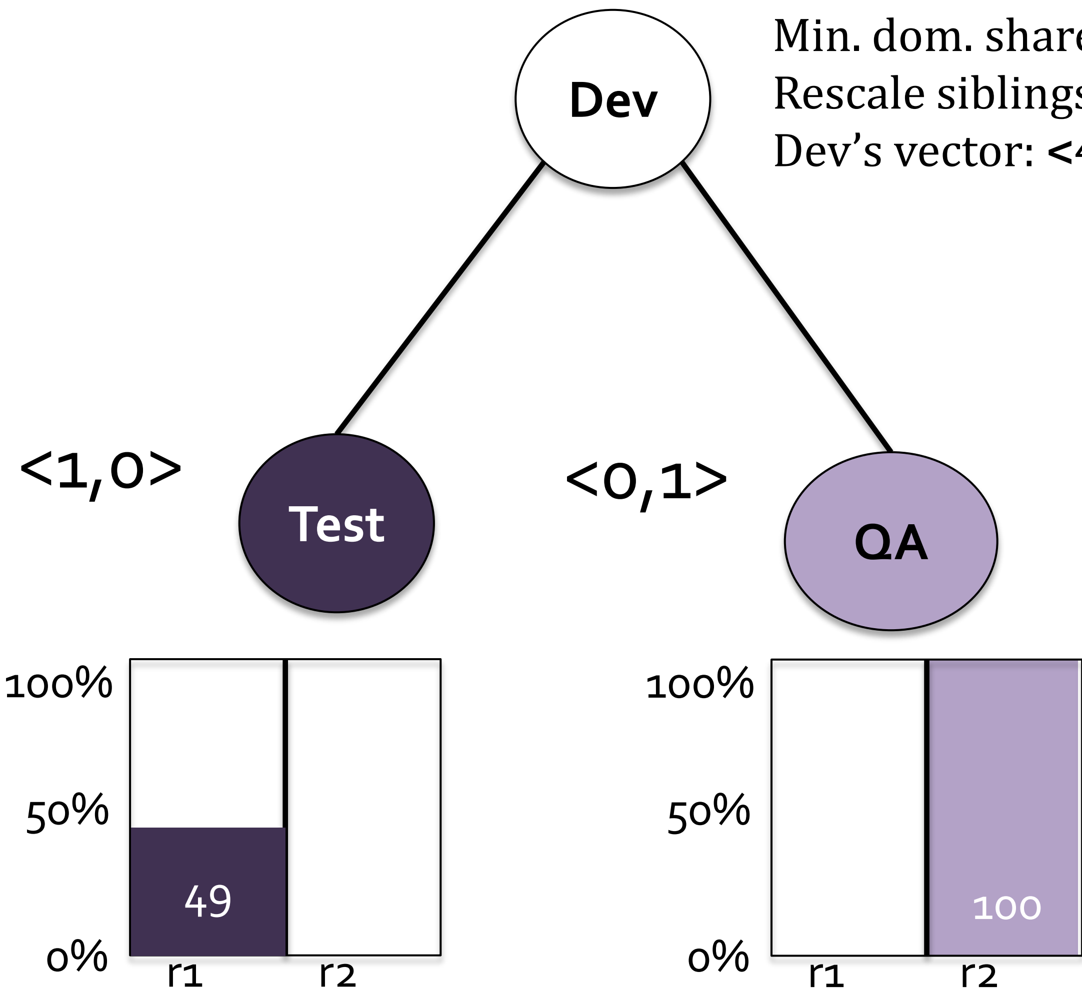
Re-scaling Consumption Vectors

- **Intuition**
  - No starvation from empty cluster
  - Rescale back as if started from empty cluster

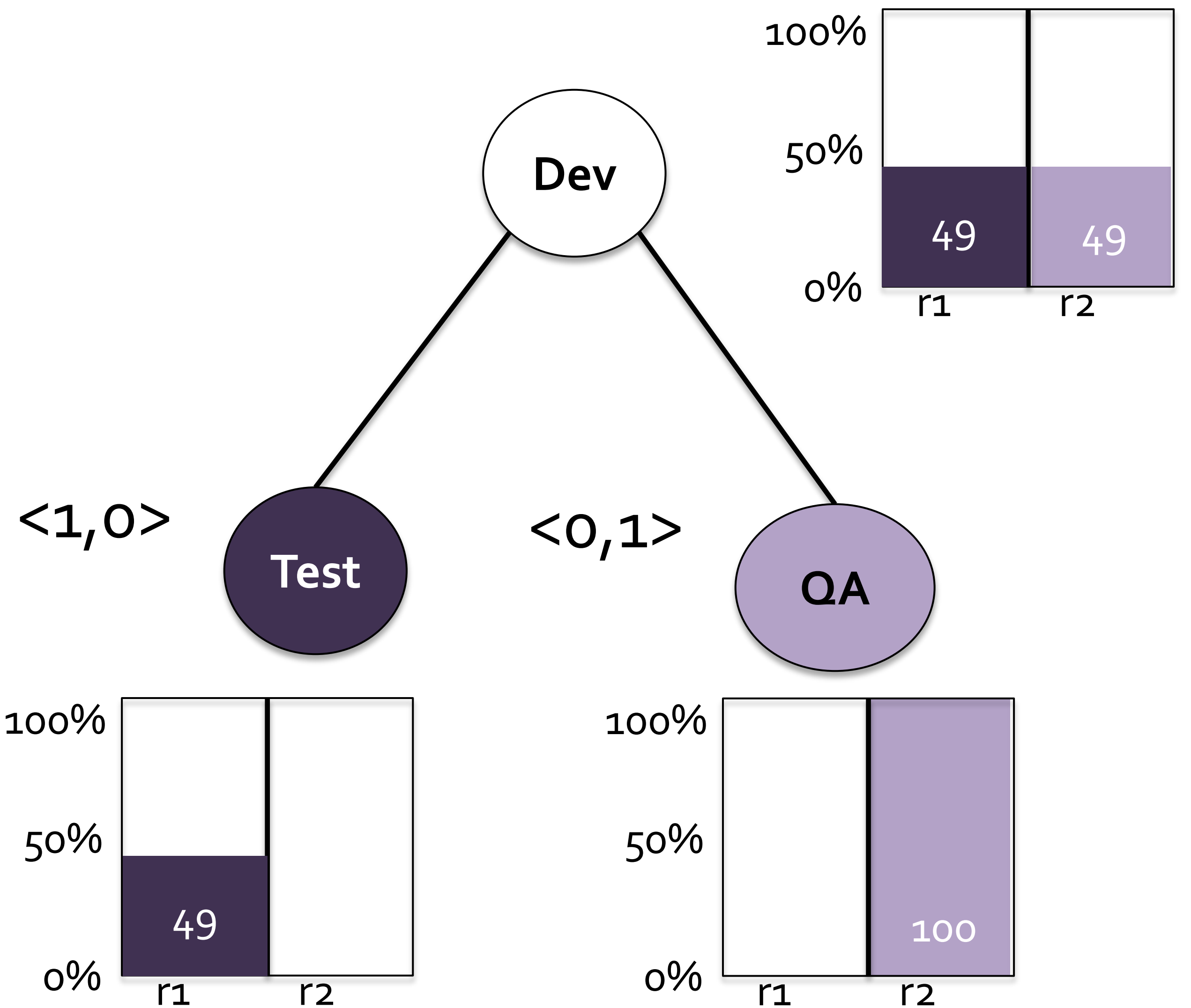
- **Re-scaling**
  - Choose sibling with lowest dominant share M
  - Rescale all siblings to have a dominant share M
  - Parent resource usage = sum of rescaled vectors

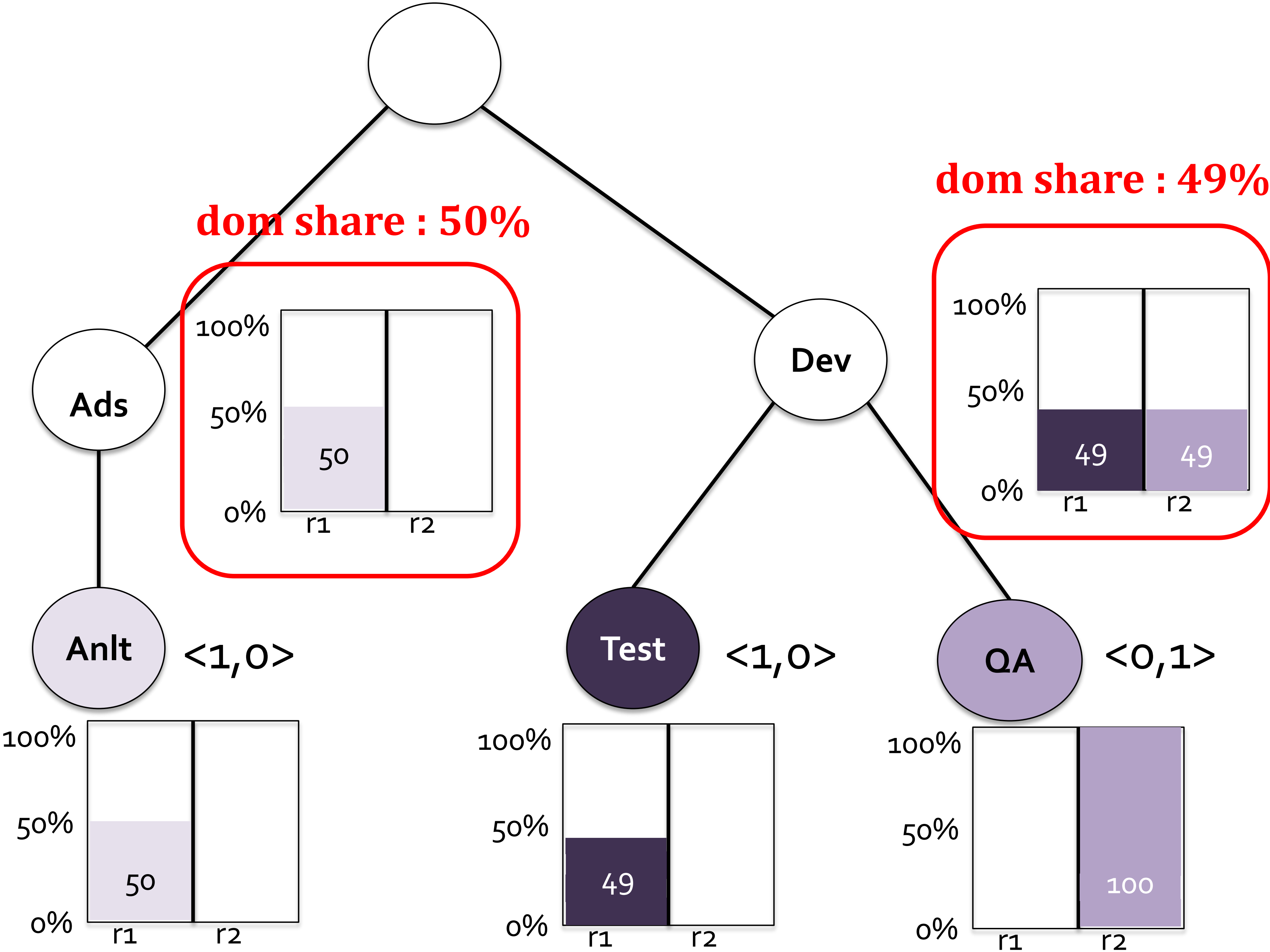
Example

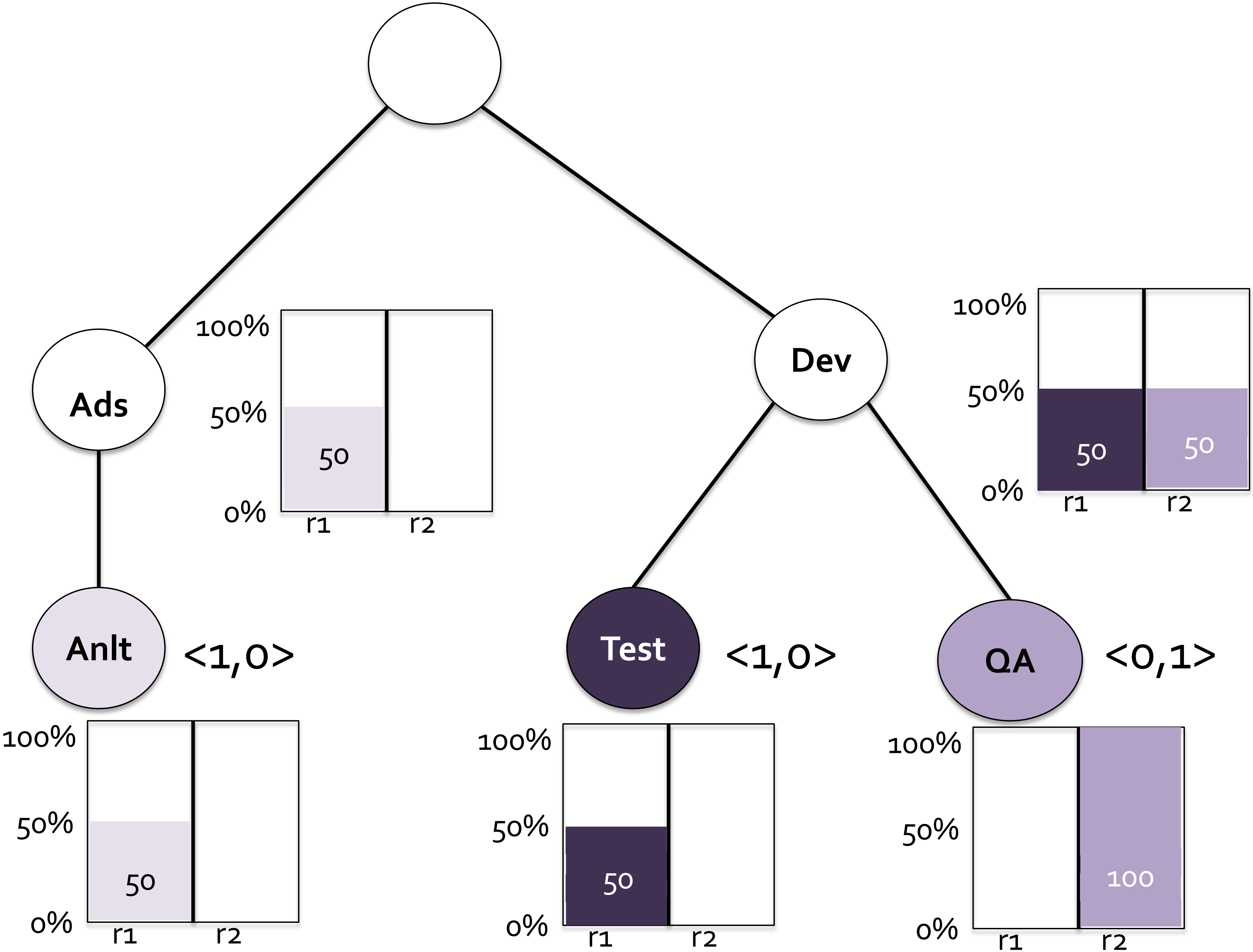
Min. dom. share: **Test (49)**
Rescale siblings: **QA <0, 49>**
Dev’s vector: **<49,0>+<0, 49> = <49,49>**



Example


100%
50%
0%
49
49
r1
r2
Dev
<1,0>
Test
<0,1>
QA
100%
50%
0%
49
r1
r2
100%
50%
0%
100
r1
r2


dom share : 50%
dom share : 49%
Ads
Dev
Anlt
Test
QA
<1,0>
<1,0>
<0,1>
100%
50%
0%
r1
r2
50
49
100



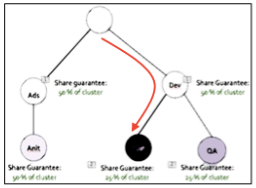
Ads
Dev
Anlt
<1,0>
Test
<1,0>
QA
<0,1>
100%
50%
0%
50
r1
r2
100%
50%
0%
50
50
r1
r2
100%
50%
0%
50
r1
r2
100%
50%
0%
50
r1
r2
100%
50%
0%
100
r1
r2

Hierarchical DRF (H-DRF)

- Leverage Static H-DRF
- Add two invariants
  - Re-scale consumption vectors
  - Ignore terminated/blocked nodes

Static H-DRF

- Traverse tree top to bottom
  - Recursively pick node with smallest dom. Share
  - Equalize siblings



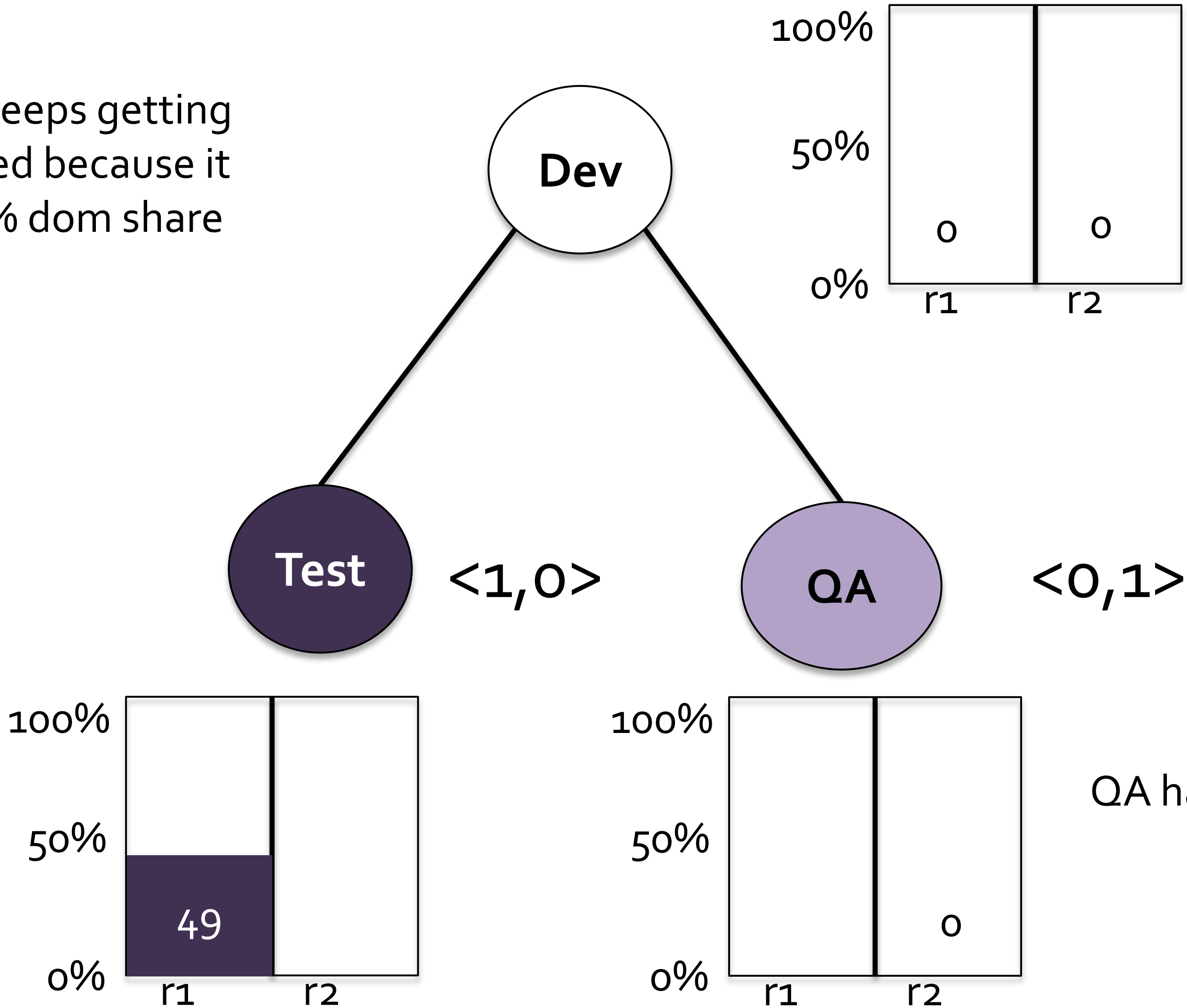
$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ current consumed resources
W resources to allocate ▷ Assumption: $R - C > W$
Y set of nonzero resources in W
A (demanding), set of leaf nodes that use only resources in Y or parents of demanding nodes
n_r ▷ root node in hierarchy tree
$C(n)$ ▷ children of any node n
$s_i \ (i = 1 \ldots n)$ ▷ dominant shares
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ (i = 1 \ldots n)$ ▷ "scaled" resources
Recompute s**:** $UpdateS(n_r)$
Allocate the resources: Alloc(W)

function (recursive) *UpdateS*(n_i)
if n_i is a leaf node **then**
  $s_i = \max U_{ij}/R_j$ for $j \in Y$
  return U_i
else
  Q = set of U_j's from *UpdateS*(n_j) for children of n_i
  f = maximum dominant share from Q restricting to nodes in A and resources in Y
  Rescale demanding vectors in Q by f
  U_i = sum of vectors in Q
  $s_i = \max U_{i,j}/Rj$ for $j \in Y$
return U_i

function *Alloc*(W)
$n_i = n_r$
while n_i is not a leaf node (job) **do**
  n_j = node with lowest dominant share s_j in $C(n_i)$, which also has a task in its subtree that can be scheduled using W
  $n_i = n_j$
$D_i = \frac{W_i}{\max_j\{T_{i,j}\}} T_i$, s.t. T_i is n_i's task demand vector
$C = C + D_i$ ▷ update consumed vector
$U_i = U_i + D_i$ ▷ update leaf only

Example

‘Dev’ keeps getting selected because it has 0% dom share



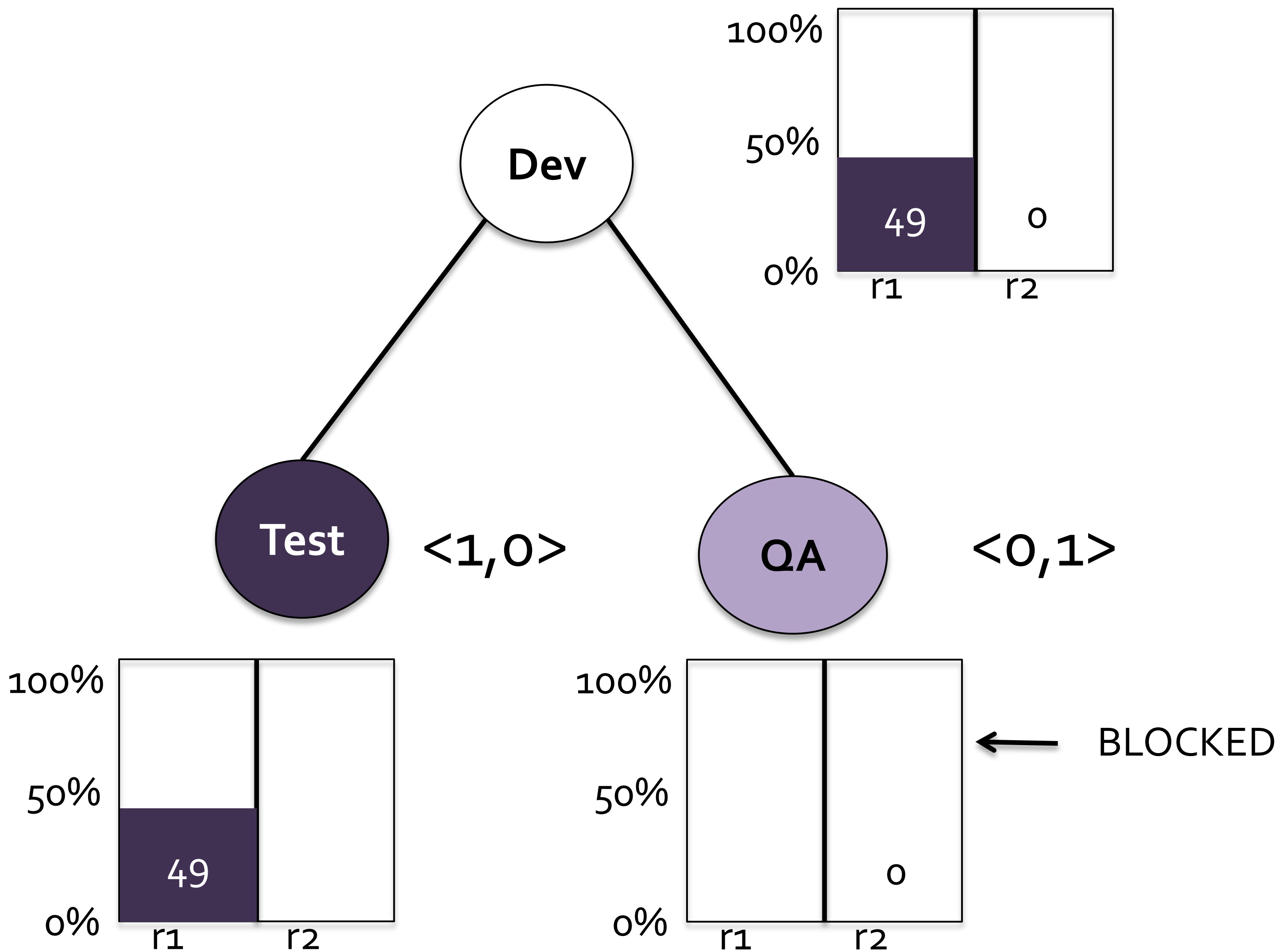
QA has no tasks

Ignore Blocked Nodes

- **A node is blocked iff**
  - No more demand
  - Cannot be allocated more resources
  - All its children are blocked

- **Ignore blocked nodes**
  - Only look at non-blocked siblings for min M
  - Rescale non-blocked nodes to dominant share M

Ignoring terminated/ blocked nodes



Outline

- How to schedule multi-resources? (DRF)
- Why is it challenging?
- What's our solution? (H-DRF)
- How well does it work?

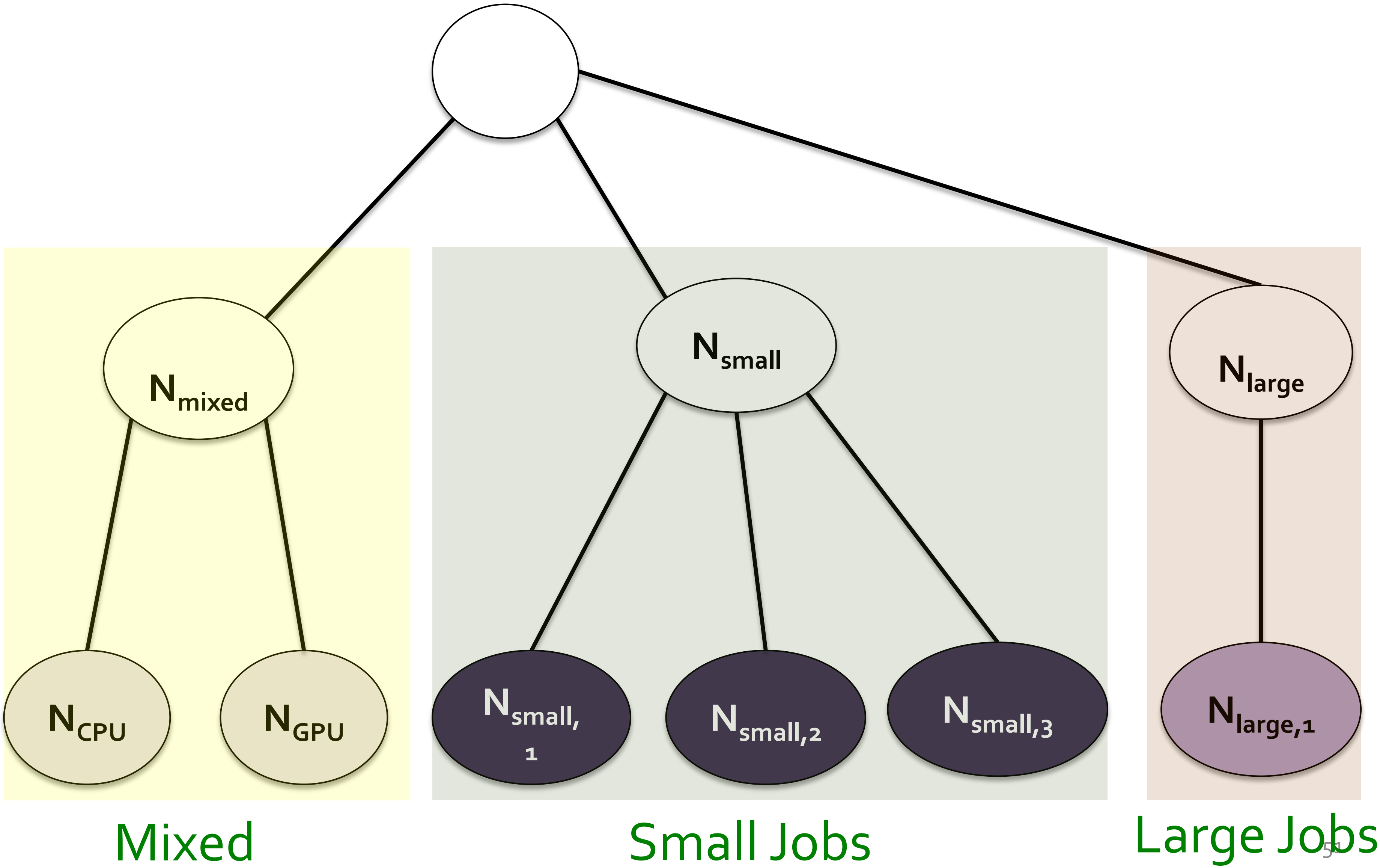
Evaluation

- 50 EC2 nodes having 6 GB memory, 4 CPUs and 1 GPU each.

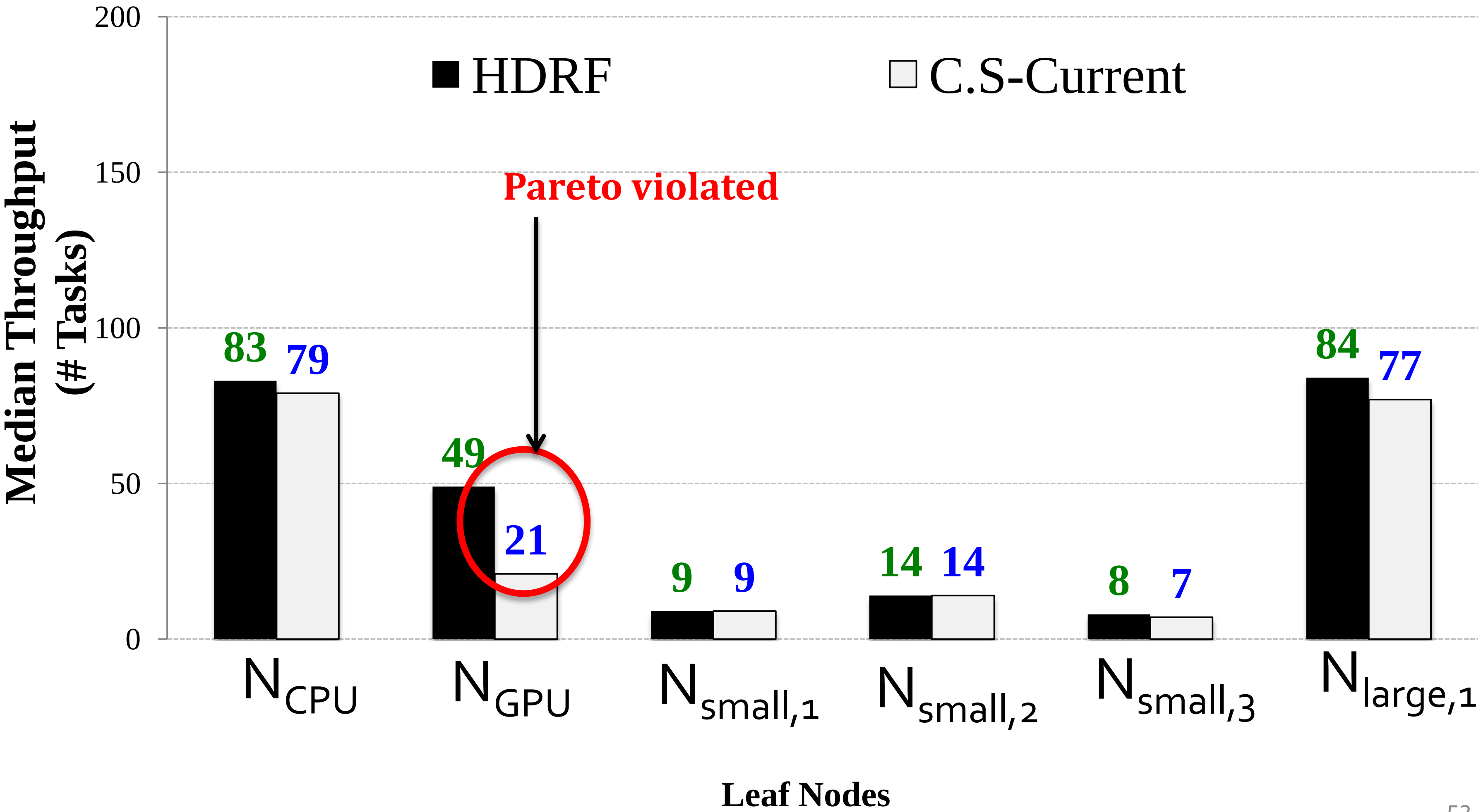
- Evaluated against
  - Hadoop Capacity Scheduler (not **Pareto**)
  - Hadoop Capacity Scheduler (Pareto added)

- Input : A 100-job schedule containing a mix of large and small jobs

Hierarchy Used

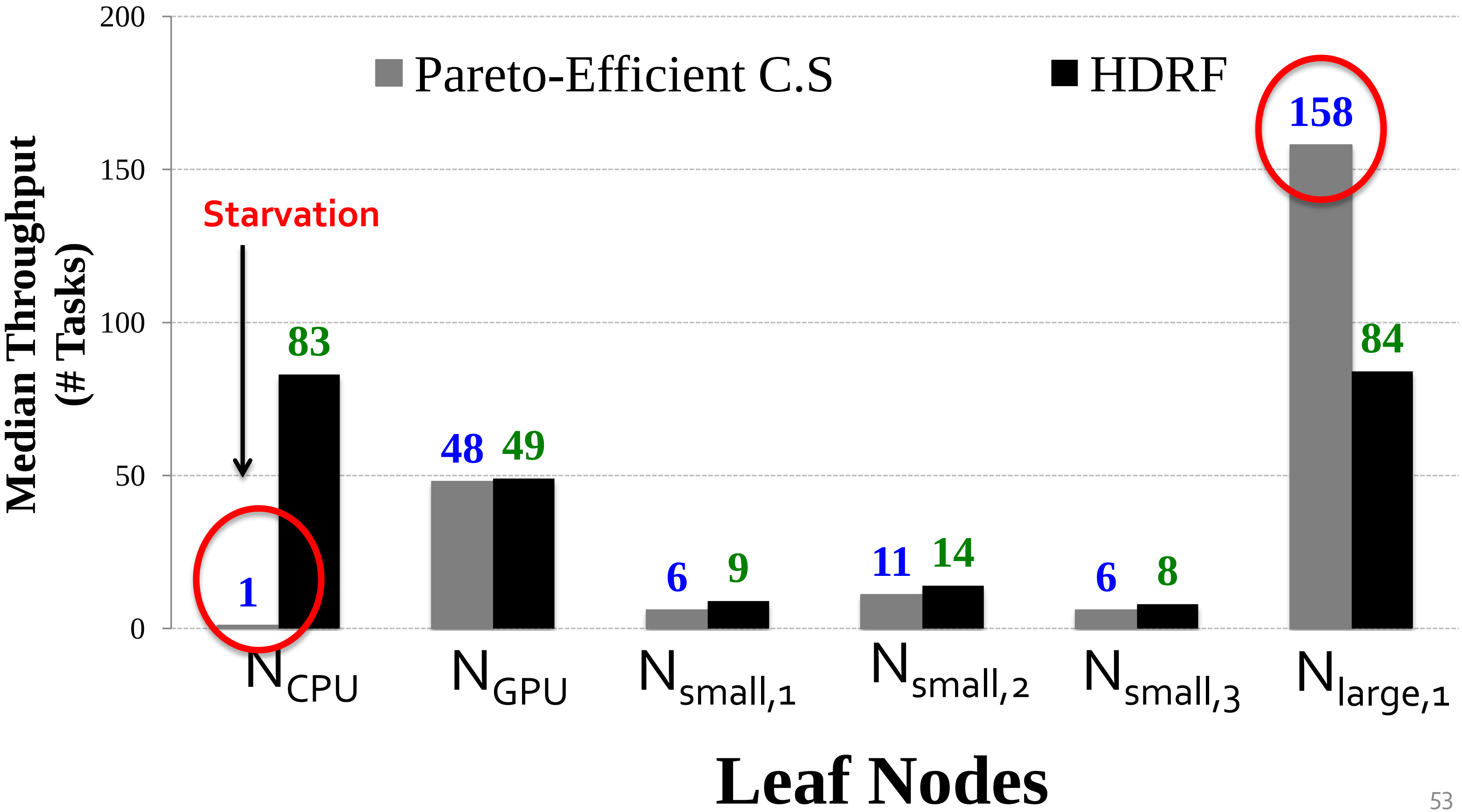


Throughput



Median Throughput (# Tasks)
200
150
100
50
0
HDRF
C.S-Current
Pareto violated
83
79
49
21
9
9
14
14
8
7
84
77
N_{CPU}
N_{GPU}
$N_{small,1}$
$N_{small,2}$
$N_{small,3}$
$N_{large,1}$
Leaf Nodes

Throughput



Median Throughput (# Tasks)
200
150
100
50
0
Pareto-Efficient C.S
HDRF
Starvation
1
83
48
49
6
9
11
14
6
8
158
84
N CPU
N GPU
N small,1
N small,2
N small,3
N large,1
Leaf Nodes

Conclusion

- Hierarchical scheduling policies important

- Hierarchical + Multi-resource = Challenging
  - Starvation, or violation of share guarantees

- Proposed *H-DRF*
  - *Generalization* of DRF to hierarchies
  - Guards against starvation
  - Provides hierarchical share guarantee

Thank you

Algorithm

$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ current consumed resources
W resources to allocate ▷ Assumption: $R - C > W$
Y set of nonzero resources in W
A (demanding), set of leaf nodes that use only resources in Y or parents of demanding nodes
n_r ▷ root node in hierarchy tree
$C(n)$ ▷ children of any node n
$s_i \ (i = 1 \ldots n)$ ▷ dominant shares
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ (i = 1 \ldots n)$ ▷ "scaled" resources
Recompute s**:** $UpdateS(n_r)$
Allocate the resources: Alloc(W)

function (recursive) *UpdateS*(n_i)
if n_i is a leaf node **then**
  $s_i = \max U_{ij}/R_j$ for $j \in Y$
  return U_i
else
  Q = set of U_j's from *UpdateS*(n_j) for children of n_i
  f = maximum dominant share from Q restricting to nodes in A and resources in Y
  Rescale demanding vectors in Q by f
  U_i = sum of vectors in Q
  $s_i = \max U_{i,j}/Rj$ for $j \in Y$
return U_i

function *Alloc*(W)
$n_i = n_r$
while n_i is not a leaf node (job) **do**
  n_j = node with lowest dominant share s_j in $C(n_i)$, which also has a task in its subtree that can be scheduled using W
  $n_i = n_j$
$D_i = \frac{W_i}{\max_j\{T_{i,j}\}} T_i$, s.t. T_i is n_i's task demand vector
$C = C + D_i$ ▷ update consumed vector
$U_i = U_i + D_i$ ▷ update leaf only