Mirador: An Active Control Plane for Datacenter Storage

Jake Wires and Andrew Warfield
Coho Data

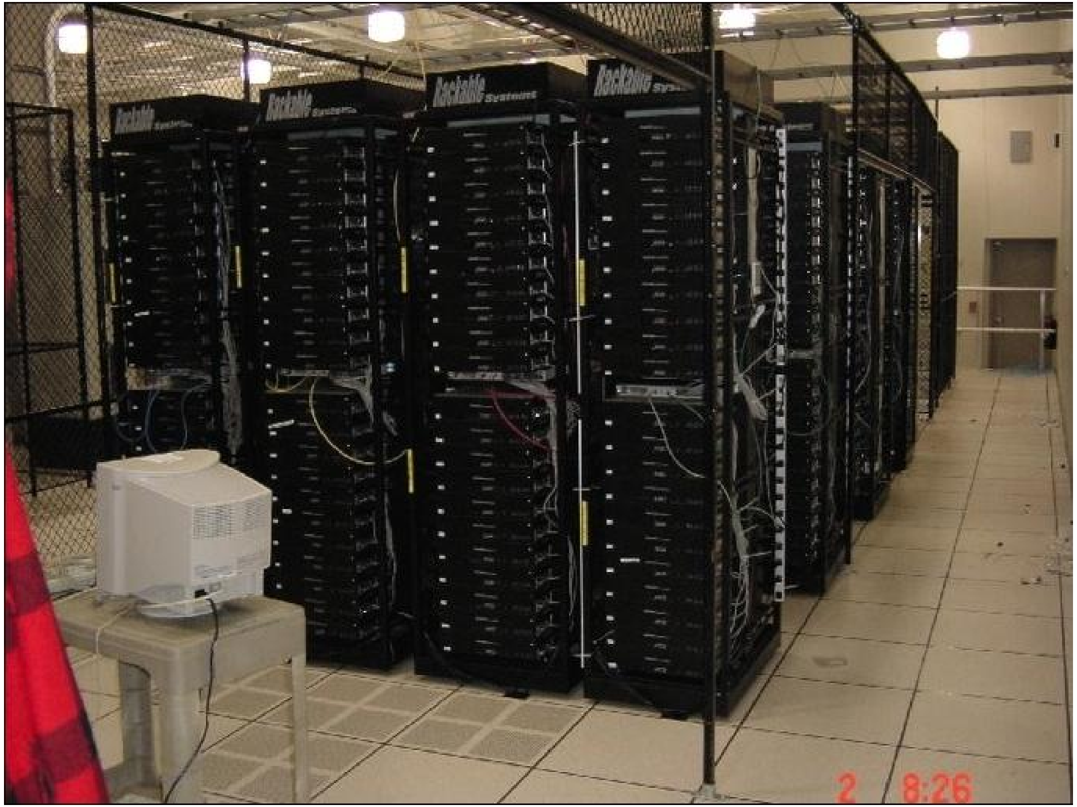
Trends

SSD	Cap / 1u	Xput per data
2 TB	64TB	312MB/s/TB
8 TB	256TB	78 MB/s/TB
32 TB	1PB	20 MB/s/TB
128 TB	4PB	5 MB/s/TB

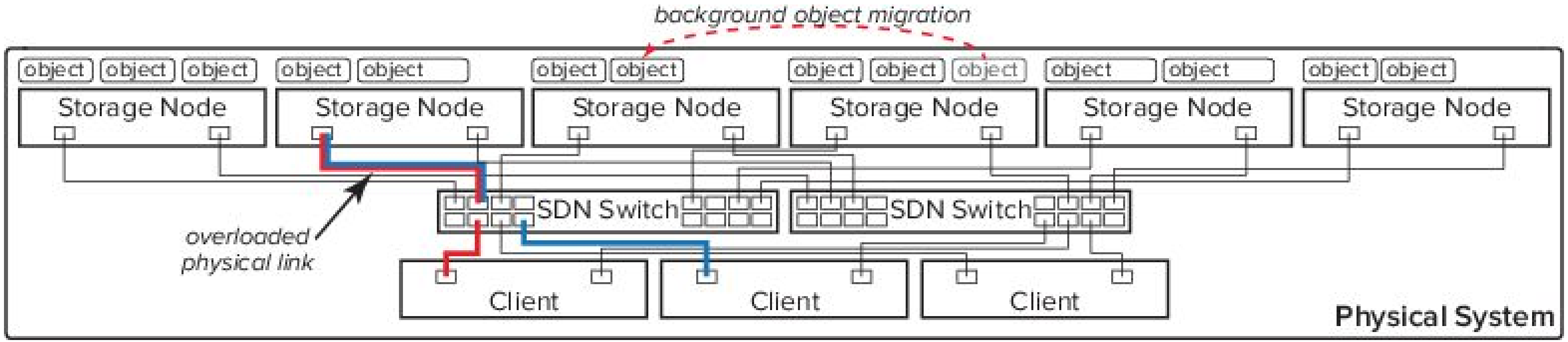
NVMe device: x4 PCIe
Broadwell CPU: 40 PCIe lanes
TOR cross-rack links typically oversubscribed at 3 or 4:1

Placement is critical

Progress



Mirador



*Mirador actively migrates **data** and **network flows** to optimize for efficiency, performance, and scale*

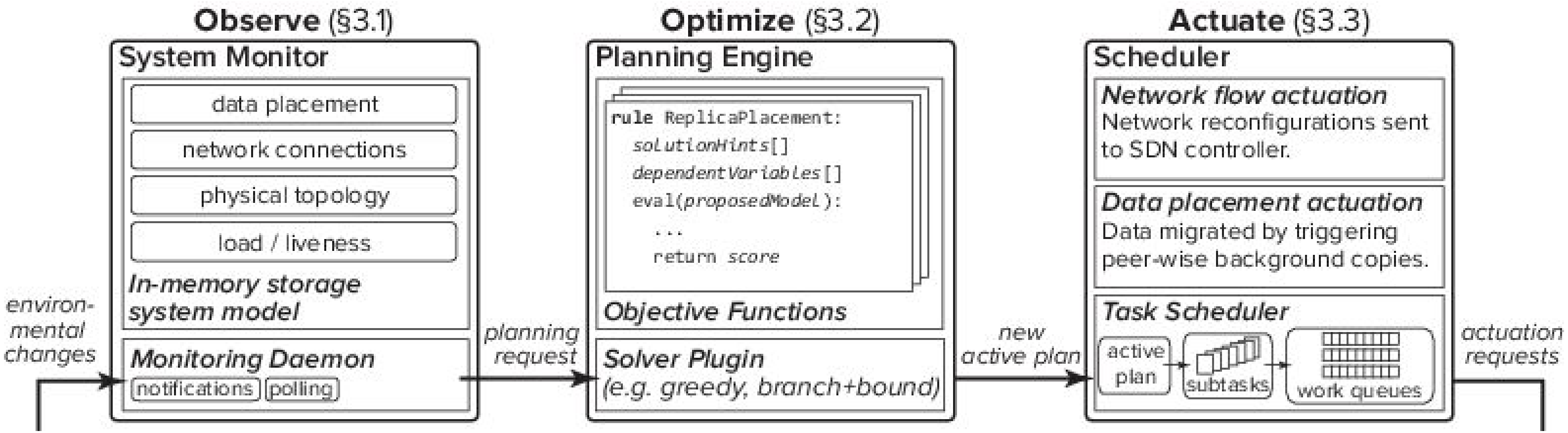
Challenges

- Software defined networking provides a nice model, but: persistence presents additional challenges
  - More constraints to satisfy
  - More dimensions to optimize
  - More expensive to reconfigure

Placement

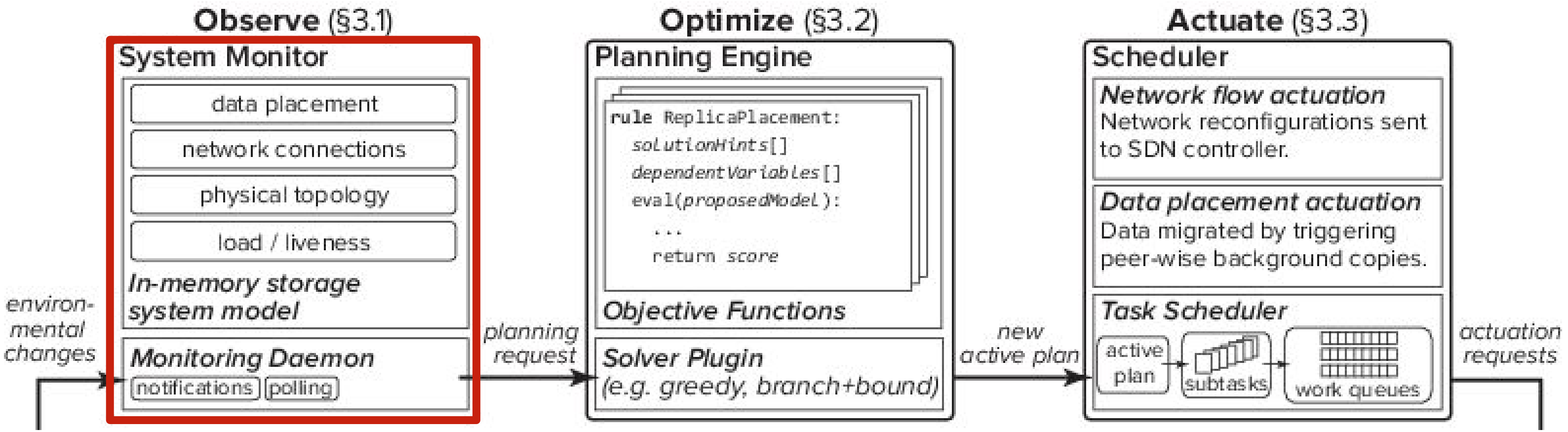
- Replicate files across failure domains
- *and* minimize cross-rack traffic
- *and* co-locate related files
- *and* stripe files across devices
- *and* respect device capacity limits
- *and* respect device performance limits
- *and* arrange for parallelizable device rebuilds
- *and* distribute load evenly across nodes
- *and* ensure exclusive caching
- *and* move cold data to cheaper media
- *and* support customer X's special requirements for multimedia files
- *and* ...

Pipeline



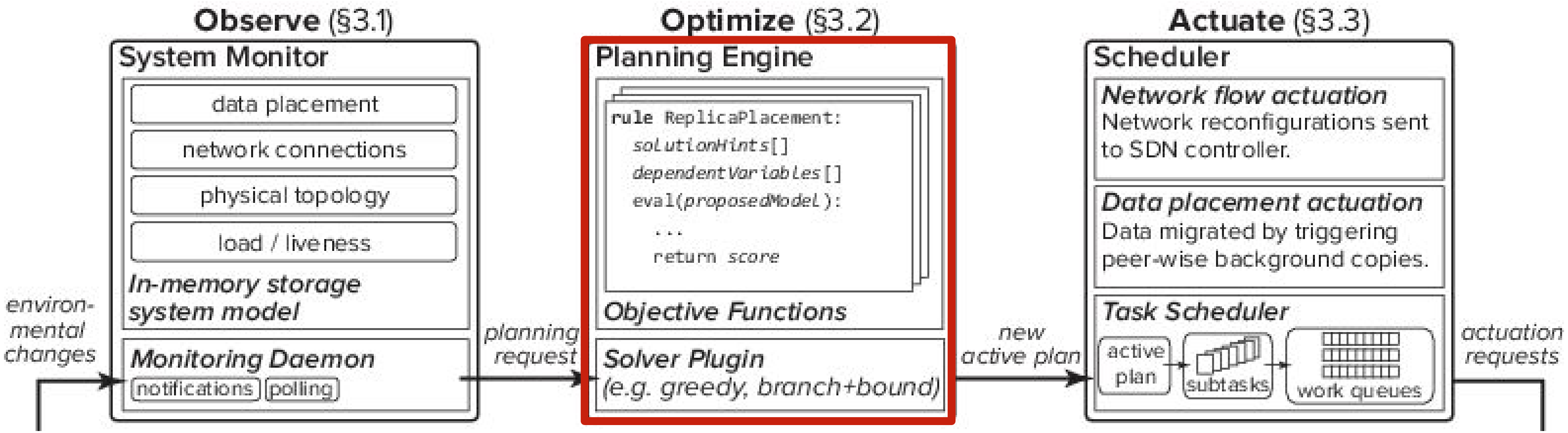
Centralized three-stage pipeline continuously optimizes placement

Pipeline



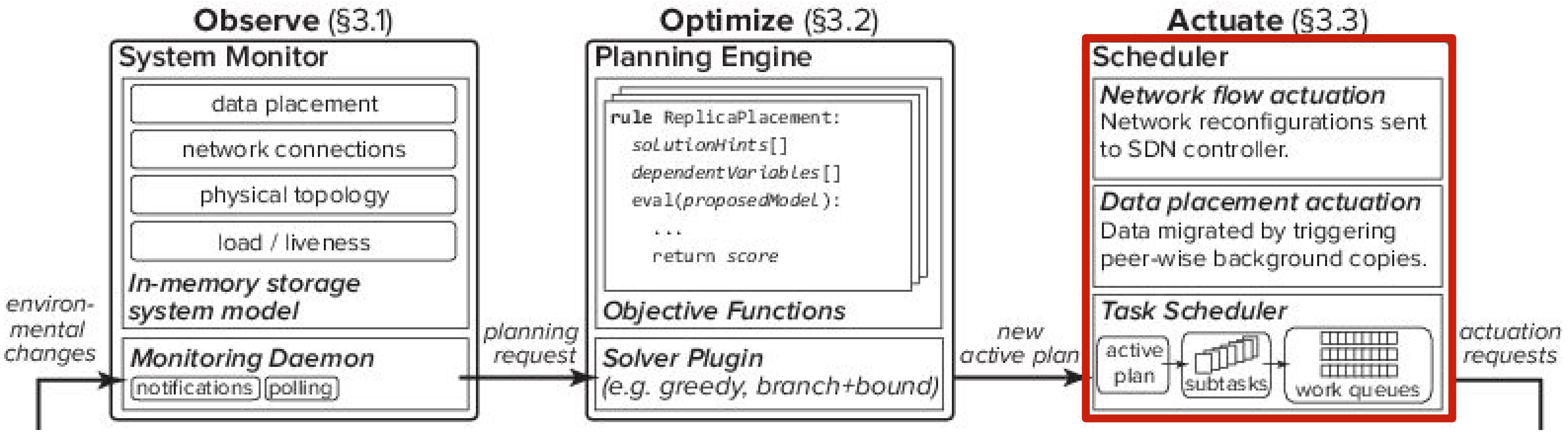
__Monitor__ collects resource utilization levels and longitudinal workload profiles

Pipeline



***Planning engine** optimizes configuration along multiple dimensions*

Pipeline



***Scheduler** coordinates migration of data and network flows*

Policy

Approach policy as a search problem:

- *Rules* (aka *objective functions*) codify intent
- *Costs* prioritize rules
- *Solvers* optimize cost

```
@rule(model.Device)
def load_balanced(fs, device, domain):
    cost, penalty = 0, DEVICE_BALANCED_COST
    # compute load of current device
    # for the current sample interval
    load = device.load()
    # compute load of least-loaded device
    minload = fs.mindevice().load()
    if load - minload > LOAD_SPREAD:
        # if the difference is too large,
        # the current device is overloaded
        cost = penalty
    return cost
```

*Rules **quantify** violations*

Optimization

- Given an existing configuration and a set of policy rules:
  - Minimize cost of violations
  - Minimize churn of reconfiguration
- Pluggable solver interface
  - Branch and bound
  - Greedy

*Solvers **search** for solutions*

Our Production Policy and Constraint Solver

7 rules governing:

- Network and storage performance and capacity balancing
- Replication across tiers and failure domains
- Device parallelism for striped files

Two-pass greedy algorithm

- Addresses highest-cost violations first
- Uses hints provided by rules to prune search space

rules.py: 219 sloc   solver.py: 128 sloc   glue.py: 1330 sloc

A Monolithic Alternative

engine.py: 2,289 sloc

Assigning Costs

- Rules do **not** eliminate the tension between conflicting goals
- They **do** provide convenient knobs for tuning the overarching policy

test/.yaml: 11,954 sloc*

A typical policy test case

```yaml
# -*- mode: yaml -*-

description: |
  distinct failure domains are prioritized over capacity balance.

fs: !fs &FS
  cfg: !config {mirrors: 2}

  nodes:
    - !node &N1 {name: N1, domains: !!set {D1}}
    - !node &N2 {name: N2, domains: !!set {D2}}

  stores:
    - !store &S1 {name: S1, state: ready, capacity: 500, used: 0, node: *N1}
    - !store &S2 {name: S2, state: ready, capacity: 500, used: 0, node: *N1}
    - !store &S3 {name: S3, state: ready, capacity: 500, used: 200, node: *N2}

files:
  - !file
    path: /file
    inode: !inode
      rplsets:
        - !rplset &R1
          ino: 0
          replicas:
            - !rpl {size: 100, state: insync, sid: S1}
            - !rpl {size: 100, state: insync, sid: S2}

preconditions:
  - [rplset_domains_unique, *R1]

postconditions:
  - [store_balanced, *S2]

plan:
  - !rplset
    ino: 0
    replicas:
      - !rpl {size: 100, state: insync, sid: S1}
      - !rpl {size: 100, state: insync, sid: S2}
      - !rpl {size: 100, state: outofsync, sid: S3}
```

Assigning Costs

- Rules do **not** eliminate the tension between conflicting goals
- They **do** provide convenient knobs for tuning the overarching policy

This complexity exists independently from policy language

A typical policy test case

```yaml
# -*- mode: yaml -*-

description: |
  distinct failure domains are prioritized over capacity balance.

fs: !fs &FS
  cfg: !config {mirrors: 2}

  nodes:
    - !node &N1 {name: N1, domains: !!set {D1}}
    - !node &N2 {name: N2, domains: !!set {D2}}

  stores:
    - !store &S1 {name: S1, state: ready, capacity: 500, used: 0, node: *N1}
    - !store &S2 {name: S2, state: ready, capacity: 500, used: 0, node: *N1}
    - !store &S3 {name: S3, state: ready, capacity: 500, used: 200, node: *N2}

files:
  - !file
    path: /file
    inode: !inode
      rplsets:
        - !rplset &R1
          ino: 0
          replicas:
            - !rpl {size: 100, state: insync, sid: S1}
            - !rpl {size: 100, state: insync, sid: S2}

preconditions:
  - [rplset_domains_unique, *R1]

postconditions:
  - [store_balanced, *S2]

plan:
  - !rplset
    ino: 0
    replicas:
      - !rpl {size: 100, state: insync, sid: S1}
      - !rpl {size: 100, state: insync, sid: S2}
      - !rpl {size: 100, state: outofsync, sid: S3}
```

Finding Solutions

Objects	Devices	Reconfigurations	Time (seconds)
1K	10	6.40 ± 2.72	0.40 ± 0.06
1K	100	145.50 ± 33.23	0.83 ± 0.08
1K	1000	220.00 ± 12.53	10.11 ± 0.49
10K	10	0.00 ± 0.00	1.61 ± 0.01
10K	100	55.70 ± 5.46	5.54 ± 0.37
10K	1000	1475.00 ± 69.70	16.71 ± 0.88
100K	10	0.00 ± 0.00	17.10 ± 0.37
100K	100	9.30 ± 4.62	22.37 ± 5.38
100K	1000	573.80 ± 22.44	77.21 ± 2.87

$O(N * \log N * \log M)$ for N objects and M devices

Workload-Aware Placement

- Policy rules informed by detailed workload profiles present new opportunities:
  - Working set size bin-packing
  - Noisy neighbor isolation
  - Workload co-scheduling
- See paper for more details!

Conclusion

- Separate control path for optimizing placement
- Active placement of data and network flows
- High dimensionality makes placement a hard problem
- Configuration as search