ParaRC: Embracing Sub-Packetization for Repair Parallelization in MSR-Coded Storage

Xiaolu Li[1], Keyun Cheng[2], Kaicheng Tang[2], **Patrick P. C. Lee[2]**, Yuchong Hu[1], Dan Feng[1], Jie Li[3], Ting-Yi Wu[3]

[1]HUST [2]CUHK [3]Huawei

USENIX FAST 2023

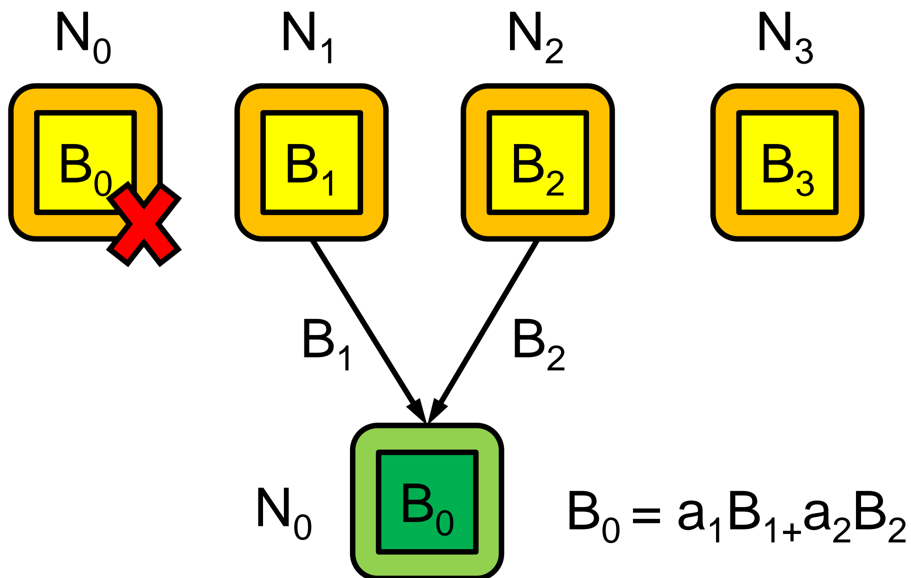
Introduction

- ➢ Erasure coding provides low-cost fault tolerance
  - Reportedly deployed in Google, Facebook, Azure, CERN
- ➢ ***(n, k)*** Reed-Solomon (RS) codes (where $k < n$)
  - Encode ***k*** uncoded blocks to ***n*** coded blocks (i.e., redundancy = n/k)
    - Each coded block is a linear combination of any *k* blocks under Galois Field arithmetic
  - **MDS**: any *k* of *n* coded blocks can recover all data, with minimum redundancy
    - (14,10) RS codes → redundancy 1.4x
    - 5x replication → redundancy 5x
    - } *Tolerate any four block failures*
  - **Drawback: High repair penalty**

Repair Penalty of RS Codes

➢ Repair penalty comes in two aspects:

- **Repair bandwidth**
  - Amount of traffic transferred over the network
- **Maximum repair load**
  - Maximum amount of traffic that a node sends or receives among all nodes



Example: (4, 2) RS code; block size = 256 MiB

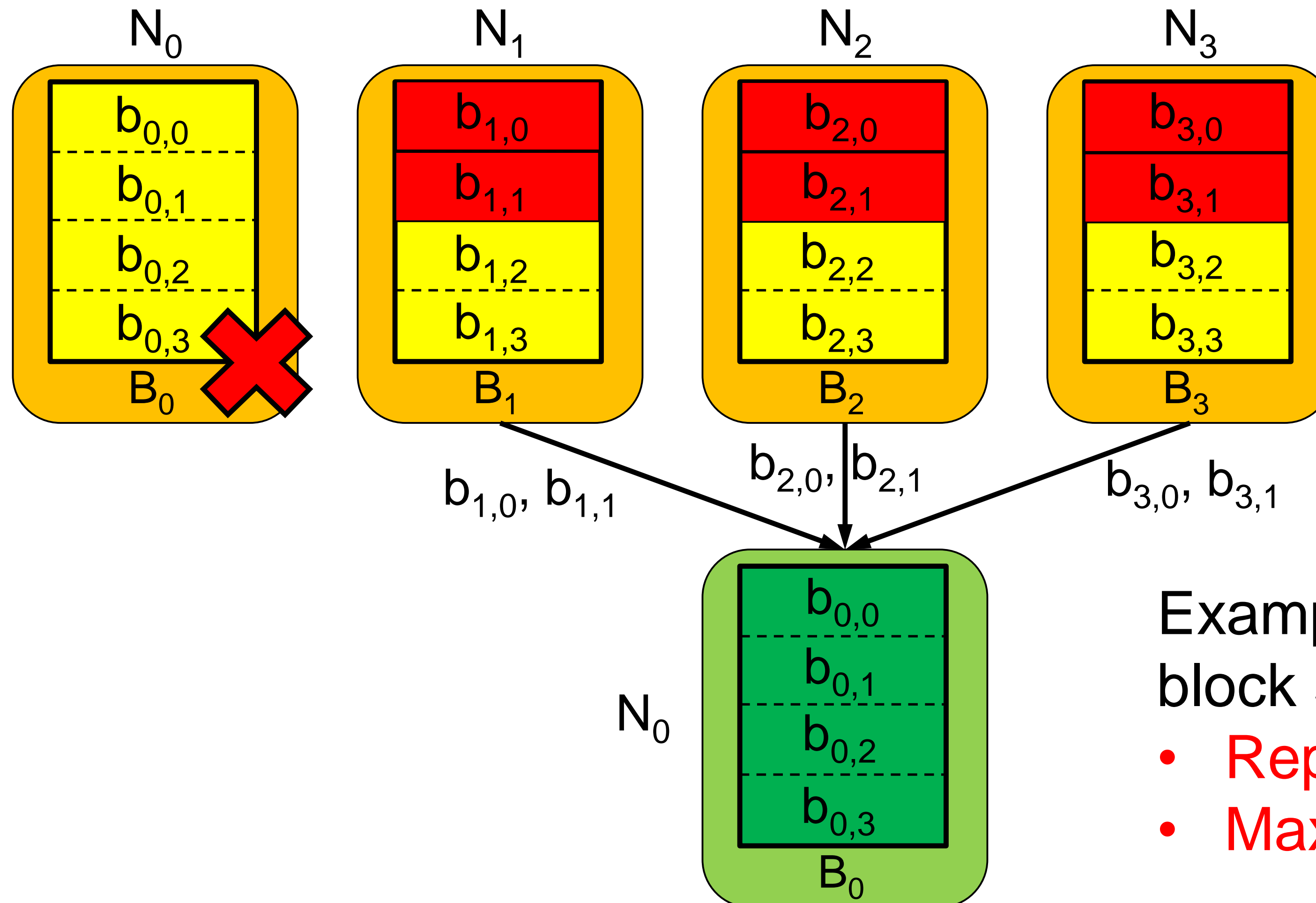
- Repair bandwidth: 512 MiB
- Maximum repair load: 512 MiB

Reducing Repair Bandwidth

- New erasure code constructions
- Minimum-storage regenerating (MSR) codes [Dimakis, TIT'10]
  - Minimum repair bandwidth, satisfying MDS
  - **Sub-packetization:** dividing a block into w sub-blocks in construction
- Examples:
  - Butterfly codes [Pamies-Juarez, FAST'16]
    - Require $n = k + 2$
  - Clay codes [Vajha, FAST'18]
    - Minimum repair bandwidth and I/O
      - Repair bandwidth = amount of I/O to local storage for repair (both are minimum)
    - General parameters n and k ($k < n$)

Clay Codes

- Centralized repair for Clay codes

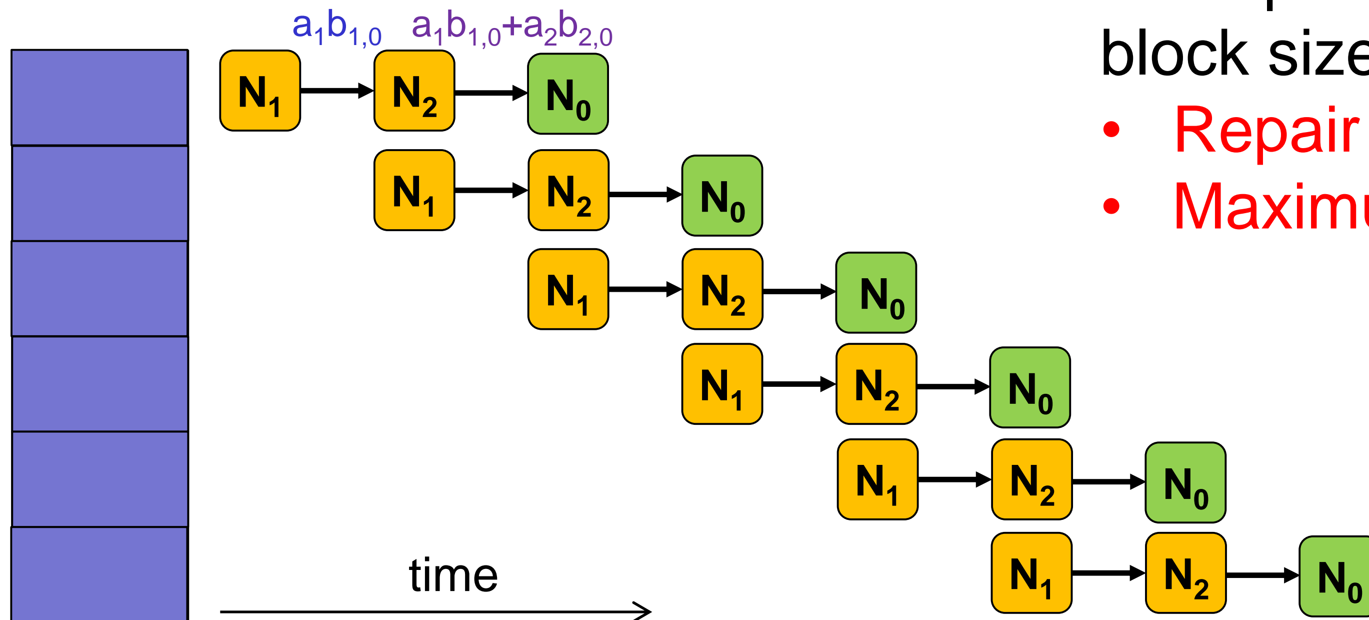


Example: (4, 2) Clay code;
block size = 256 MiB

- Repair bandwidth: 384 MiB
- Maximum repair load: 384 MiB

Reducing Maximum Repair Load

- Decomposing and parallelizing repair operations
- Repair pipelining [Li, TOS'21]
  - Divide a block into slices
  - Repair slices in parallel



Example: (4, 2) RS code;
block size = 256 MiB

- Repair bandwidth: 512 MiB
- Maximum repair load: 256 MiB

Repair Parallelization for MSR Codes

- Can we apply repair pipelining for MSR codes?
  - Unfortunately, NO.
  - Repair pipelining relies on **additive associativity** of RS codes
  - Repair of MSR codes solves a system of linear combinations

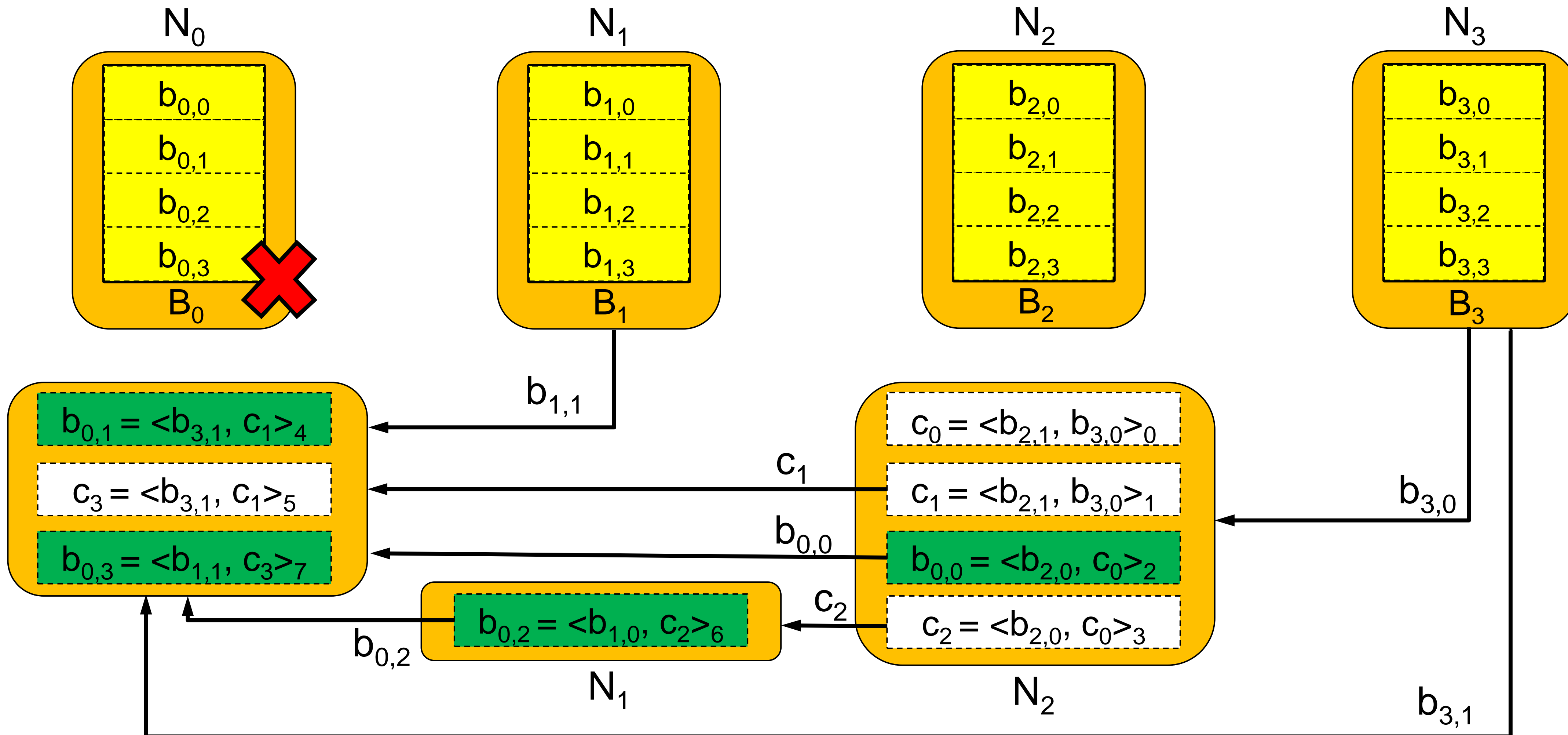
- Opportunity: exploiting sub-packetization of MSR codes
  - Repair of a sub-block requires a subset of available sub-blocks
  - We can distribute repair of sub-blocks across multiple nodes for load balancing

Motivating Example

Example: (4, 2) Clay code;
block size = 256 MiB

- Repair bandwidth: 448 MiB
- Maximum repair load: 320 MiB

- ➢ Parallel repair for Clay codes



Summary

- Comparison of repair bandwidth and maximum repair load, where $(n, k) = (4, 2)$

Repair Approach	Repair Bandwidth (MiB)	Maximum Repair Load (MiB)
RS; centralized	512 (highest)	512 (highest)
Clay; centralized	384 (lowest)	384 (high)
RS; parallel	512 (highest)	256 (lowest)
Clay; parallel	448 (medium)	320 (medium)

- **Challenge: How do we model the trade-off and design a parallel repair approach for general MSR codes?**

Our Contributions

ParaRC: A Parallel Repair Framework for MSR Codes

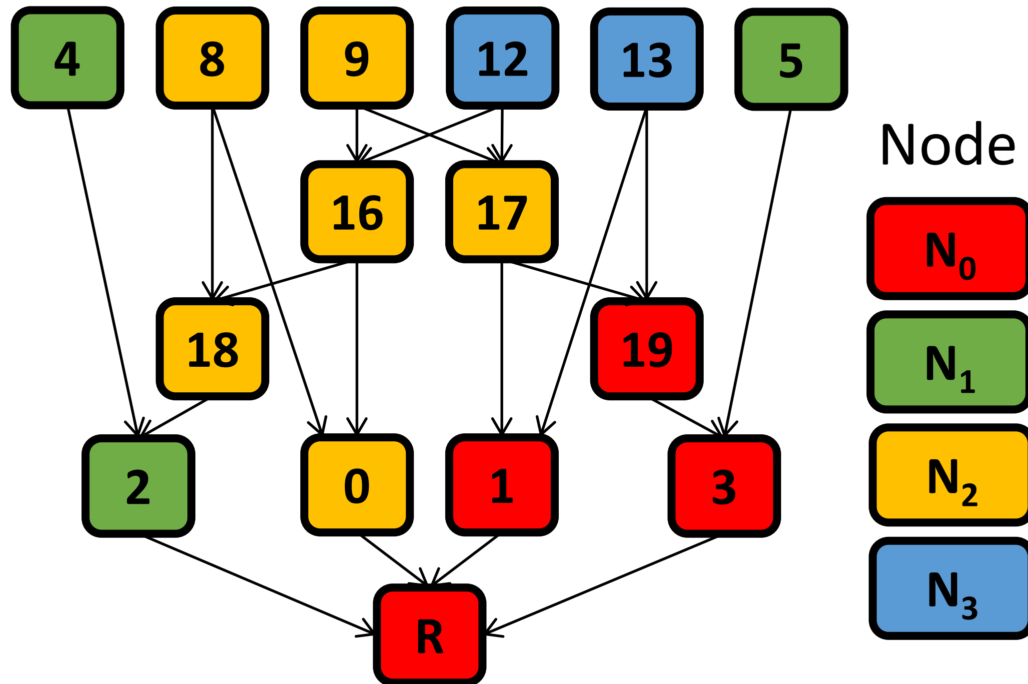
- Model parallel repair as a DAG coloring problem
  - Show the trade-off between repair bandwidth and maximum repair load
  - Identify **MLP** (min-max repair load point), which minimizes repair bandwidth given the minimum maximum repair load
- Propose a heuristic to find approximate MLP
- Prototype ParaRC atop Hadoop 3.3.4 HDFS and evaluate on Alibaba Cloud

pECDAG

- A pECDAG is a directed acyclic graph that defines a parallel repair operation
  - Built on ECDAG [Li, FAST'19]
  - Vertex v_l
    - Refers to a sub-block $b_{i,j}$ ($l = i \times w + j$) or an intermediate sub-block ($l \geq n \times w$)
  - Edge $e(v_{l1}, v_{l2})$
    - Sub-block v_{l_1} is an input to the linear combination for forming sub-block v_{l_2}
  - **Color** for each vertex
    - Refers to a node that computes or stores a sub-block

pECDAG

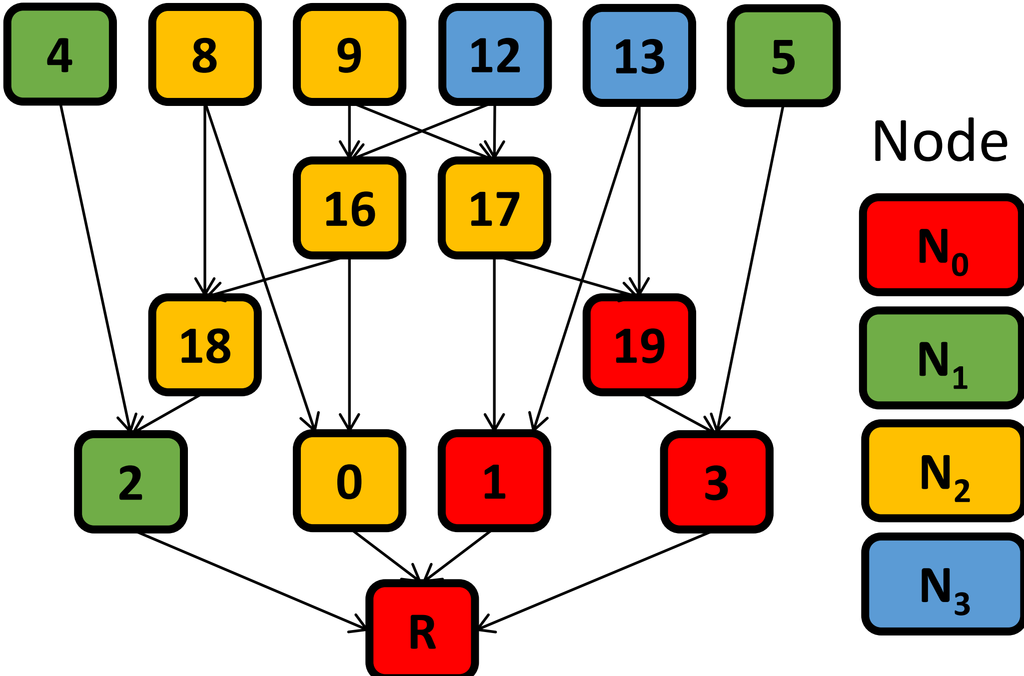
- Example of a pECDAG for repairing B_0 in (4,2) Clay code:



l	v_l	Node
4,5	$b_{1,0}$, $b_{1,1}$	Stored in N_1
8,9	$b_{2,0}$, $b_{2,1}$	Stored in N_2
12,13	$b_{3,0}$, $b_{3,1}$	Stored in N_3
16,17,18,19	c_0, c_1, c_2, c_3	Computed in N_2
0	$b_{0,0}$	Computed in N_2
2	$b_{0,2}$	Computed in N_1
1,3	$b_{0,1}$, $b_{0,3}$	Computed in N_0

Traffic Table

- **T**: traffic table that records repair load of each node
  - **T.In**: number of incoming sub-blocks
  - **T.Out**: number of outgoing sub-blocks

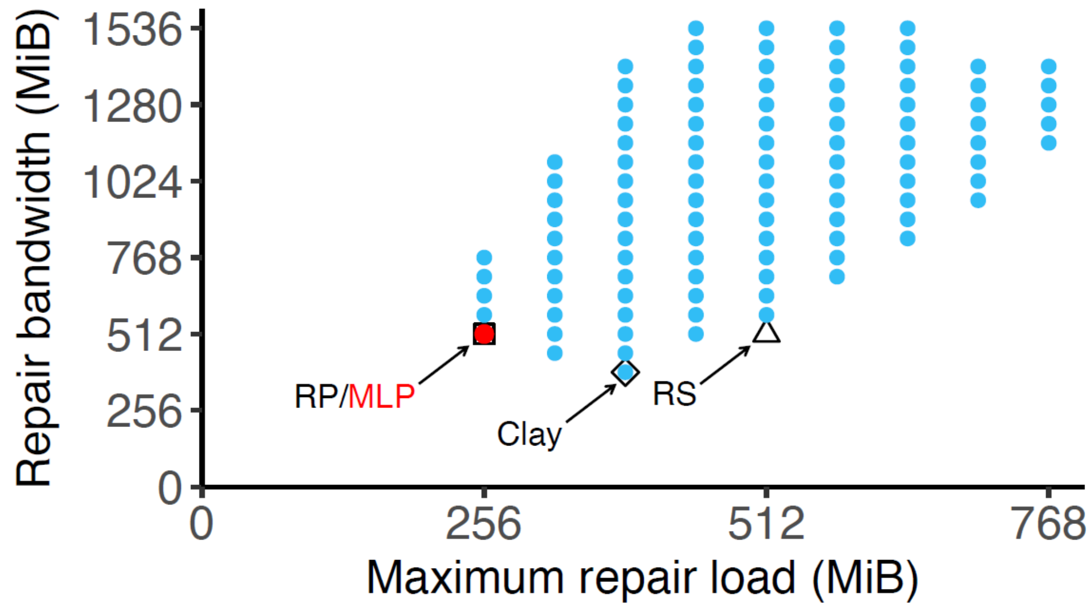


	In	Out
N_0	5	0
N_1	1	2
N_2	1	3
N_3	0	2

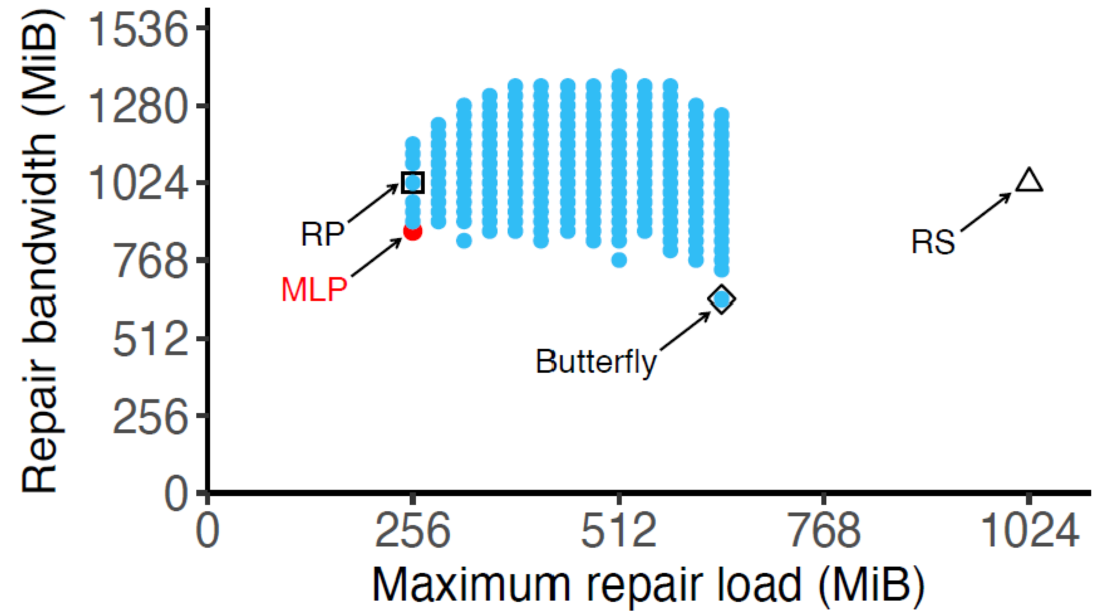
Maximum repair load: 5 sub-blocks

Repair bandwidth: 7 sub-blocks

Trade-off Analysis



(4,2) Clay code (w=4)



(6,4) Butterfly code (w=8)

- **MLP (Min-max repair load point)**
  - Minimizes repair bandwidth given the minimum maximum repair load
- Finding the MLP by brute-force is computationally infeasible

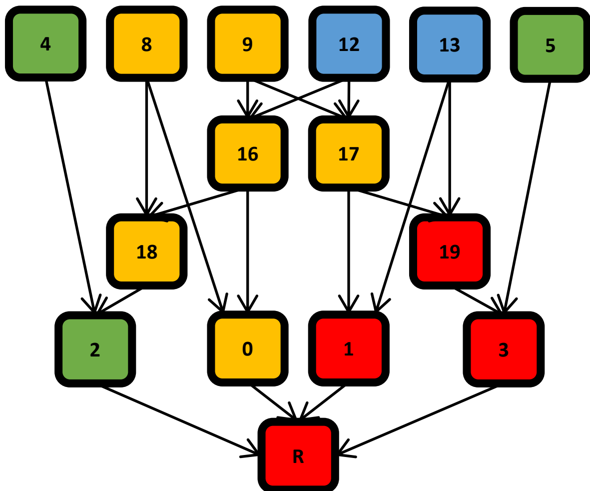
Heuristic

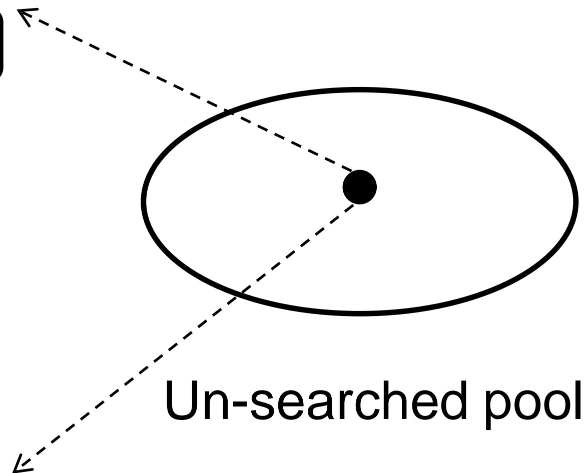
- Search all color combinations for a pEDCAG
  - Start with a pECDAG
  - Examine each of its **neighbors** (with one vertex of a different color)
  - Prune sub-optimal solutions

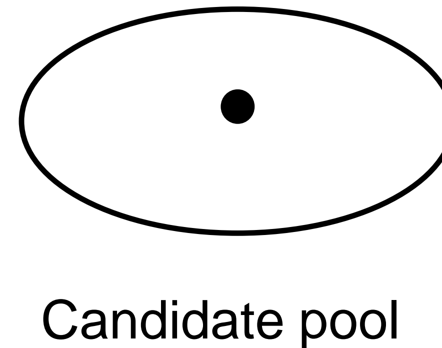
- Intuition: search on Pareto frontier and prune dominated solutions

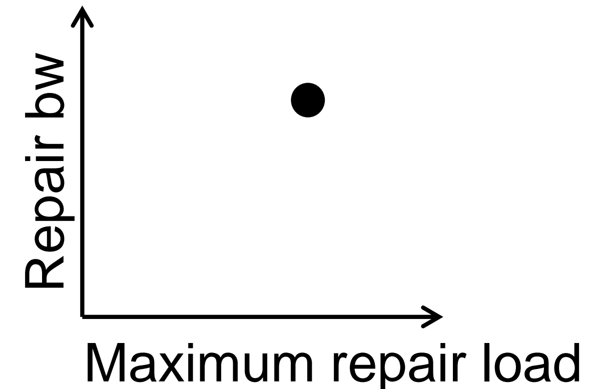
Heuristic

- Step 1: Initialization
  - Generate a pECDAG with random color combination
  - Put it into an *un-searched pool*
    - Keeps pECDAGs that will be searched
  - Put it into a *candidate pool*
    - Keeps candidate pECDAGs to be returned



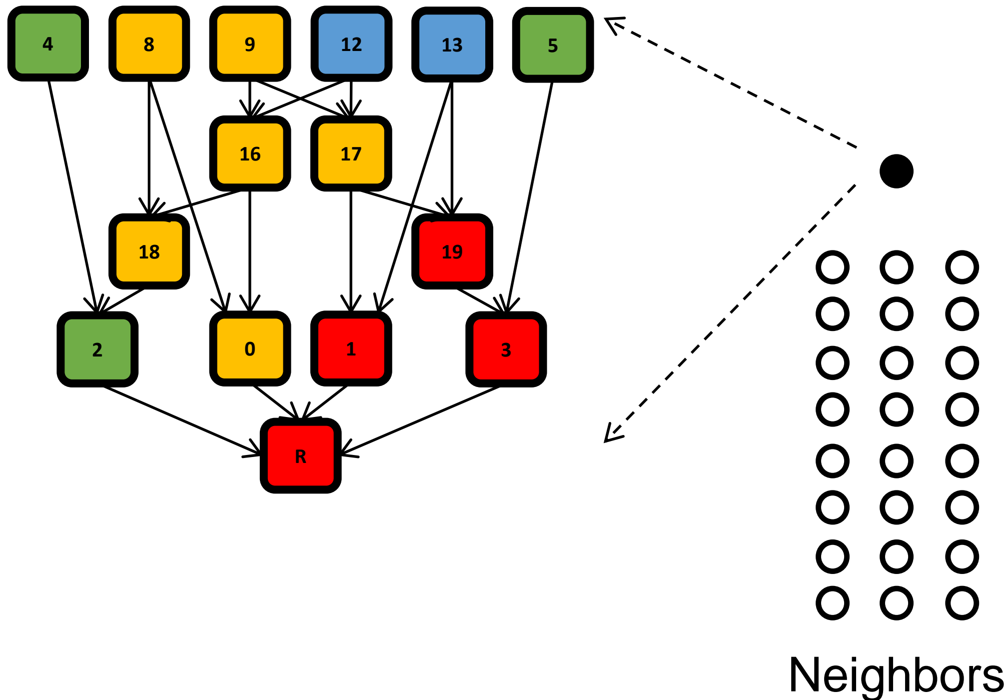


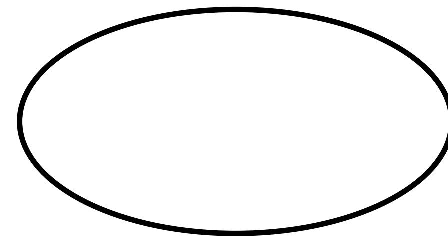


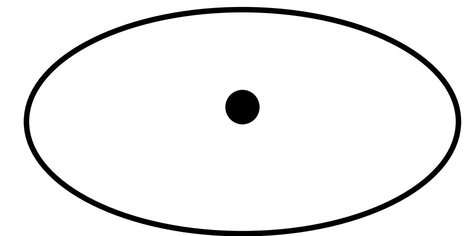


Heuristic

- Step 2: Searching
  - Retrieve a pECDAG from un-searched pool
  - Examine all its neighbors



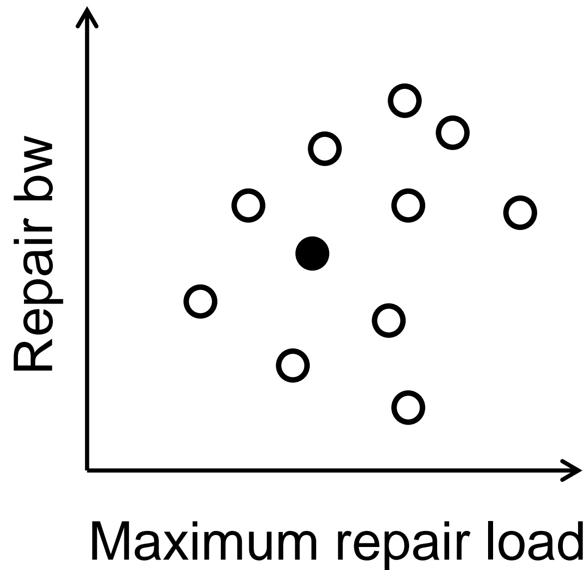


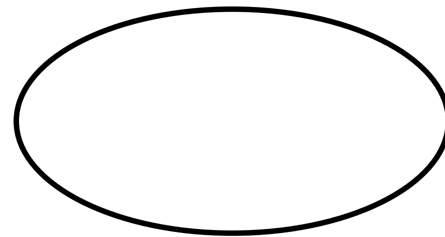


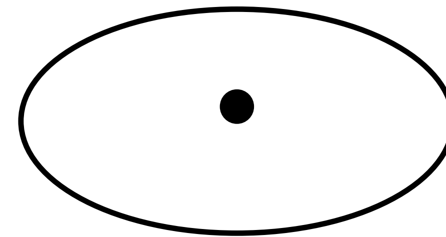
Heuristic

➢ Step 3: Pruning

- Compare each neighbor pECDAG with those in candidate pool



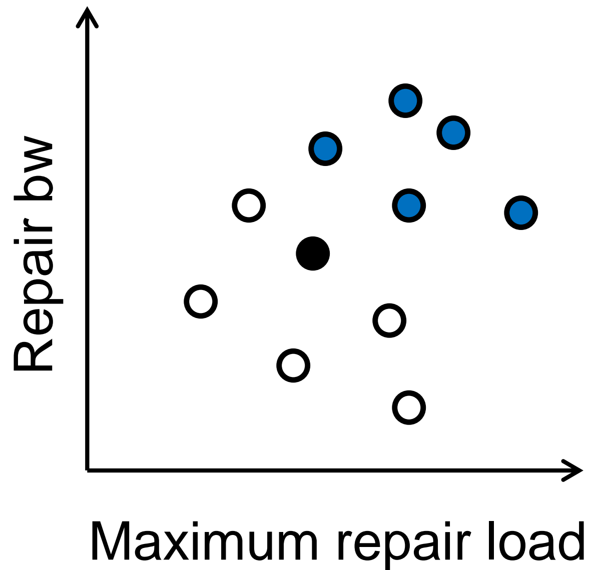


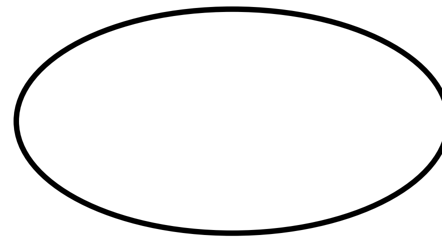


Heuristic

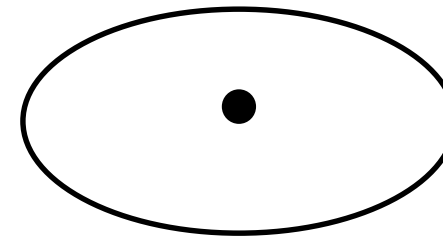
➢ Step 3: Pruning

- **Case 1:** Blue points are worse than the black point in candidate pool





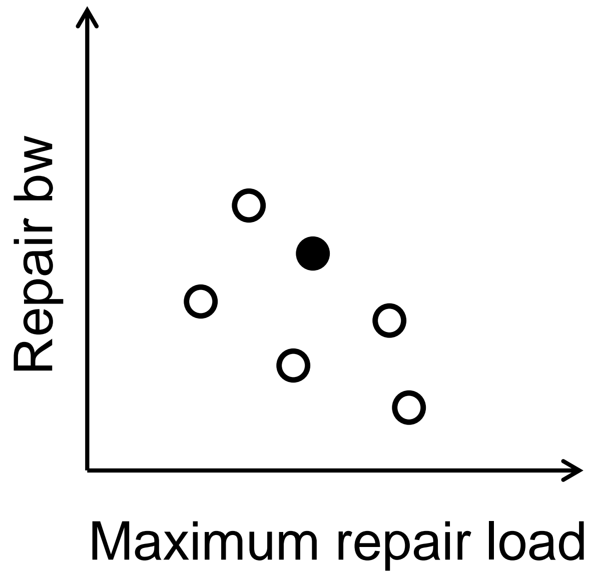
Un-searched pool

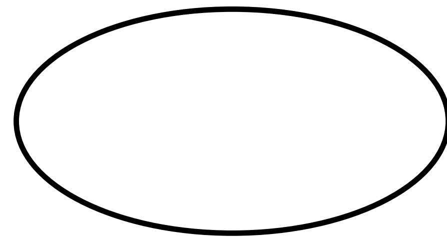


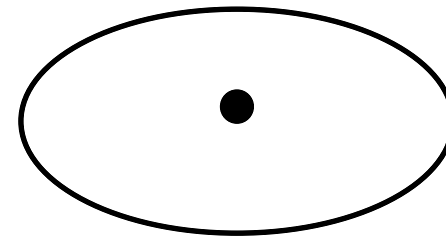
Candidate pool

Heuristic

- Step 3: Pruning
  - **Case 1:** Prune blue points





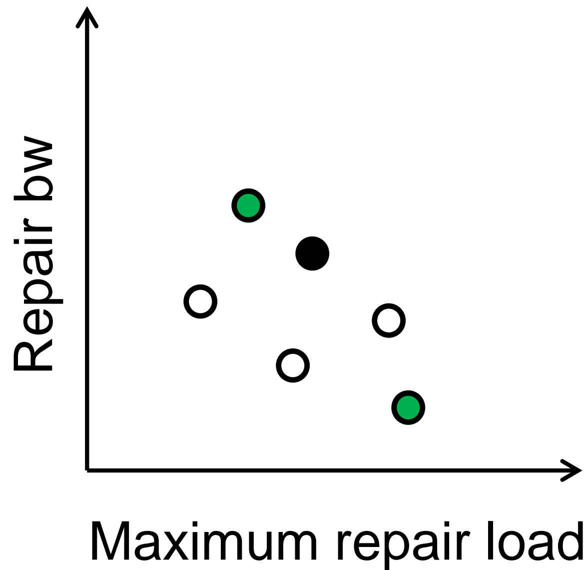


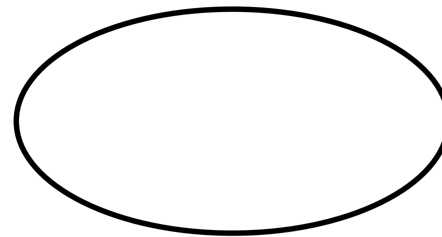
Un-searched pool

Candidate pool

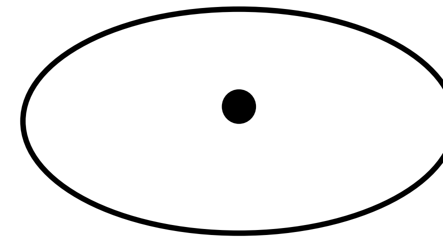
Heuristic

- Step 3: Pruning
  - **Case 2:** Green points have the least repair bandwidth or least maximum repair load compared with solutions in candidate pool





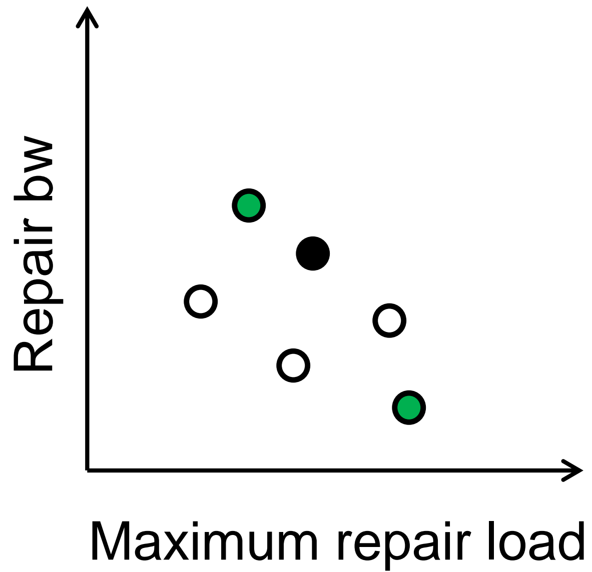
Un-searched pool

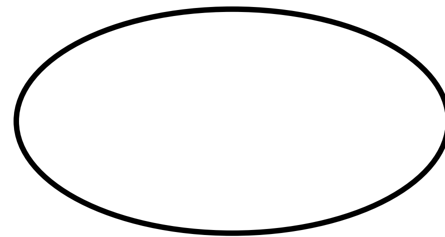


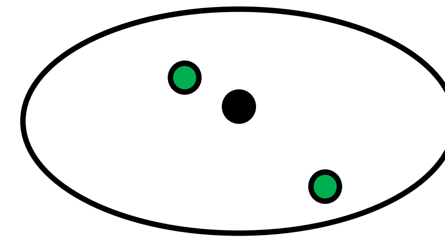
Candidate pool

Heuristic

- Step 3: Pruning
  - **Case 2:** Add green points to candidate pool

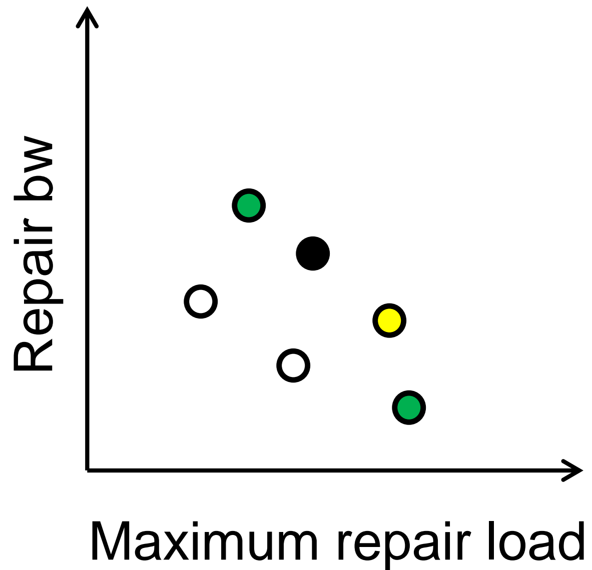


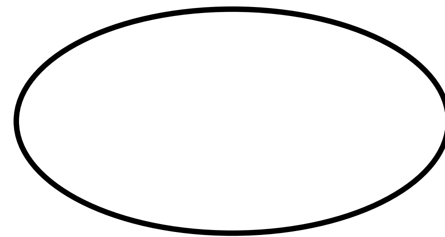

Un-searched pool


Candidate pool

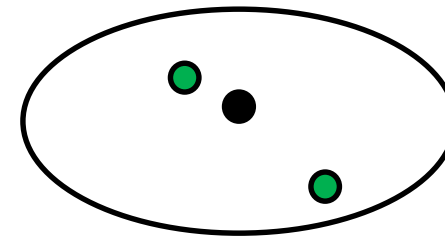
Heuristic

- Step 3: Pruning
  - **Case 3:** Yellow point lies between the solutions in candidate pool





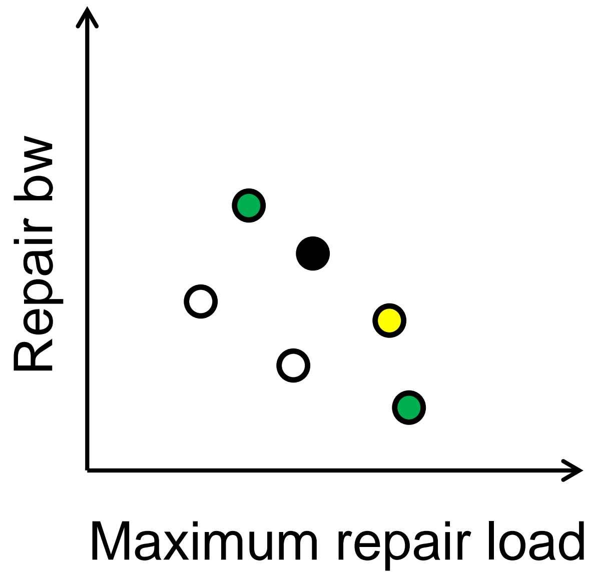
Un-searched pool

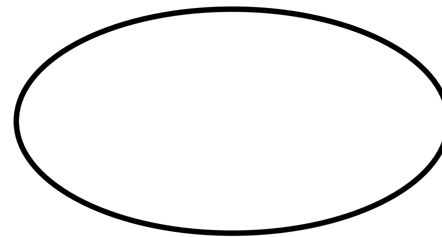


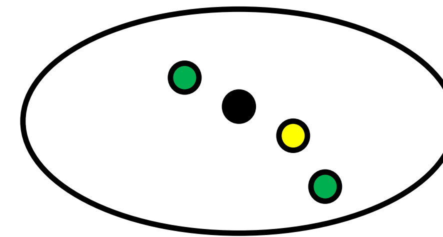
Candidate pool

Heuristic

- Step 3: Pruning
  - **Case 3:** Add yellow point to candidate pool

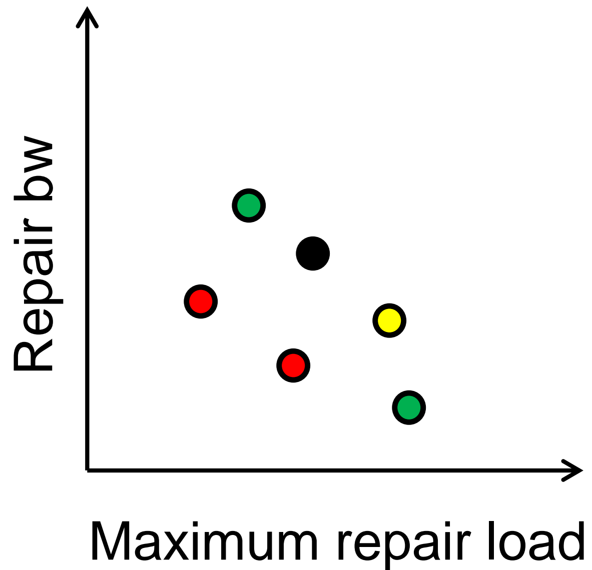


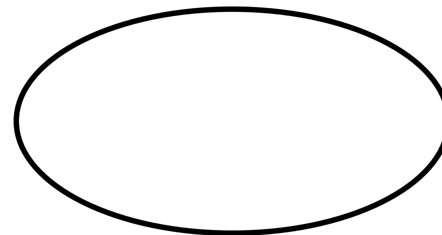


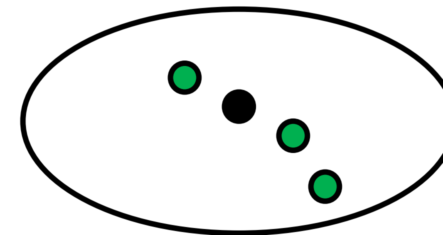


Heuristic

- Step 3: Pruning
  - **Case 4:** Red points are better than some solutions in candidate pool



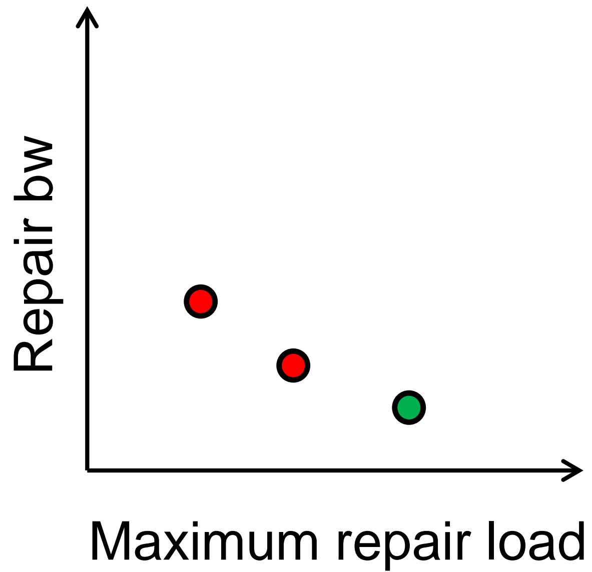


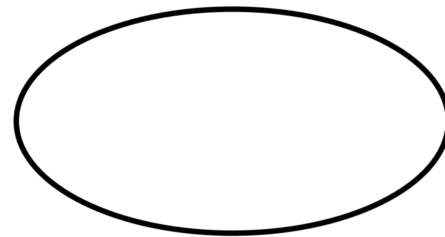


Heuristic

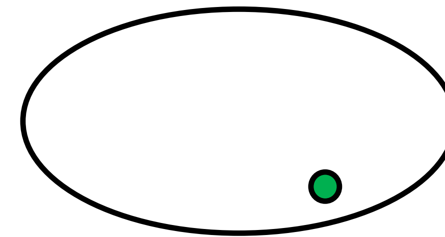
➢ Step 3: Pruning

- **Case 4:** Prune the worse candidate solutions





Un-searched pool

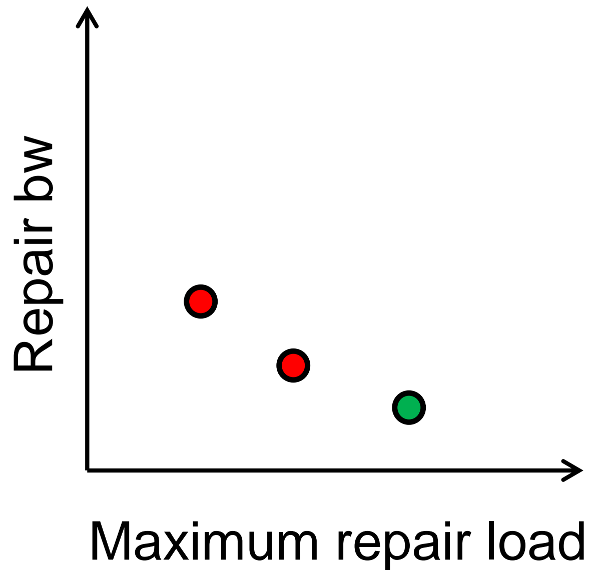


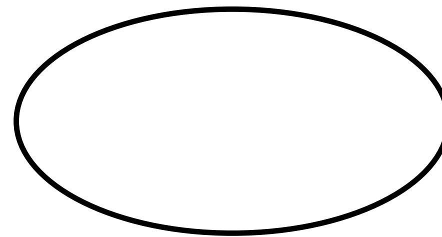
Candidate pool

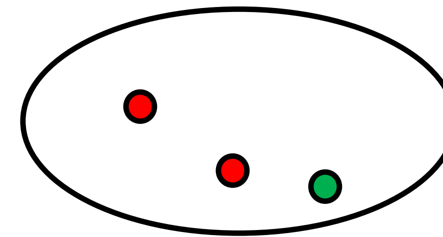
Heuristic

➢ Step 3: Pruning

- **Case 4:** Add two red points to candidate pool

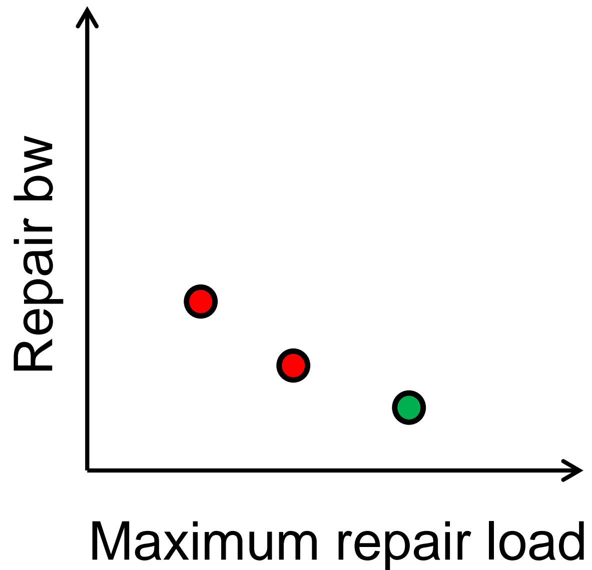


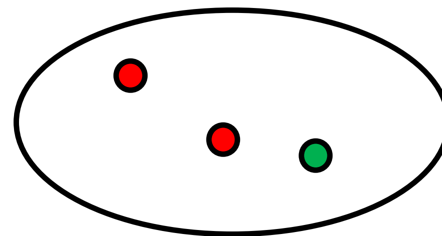


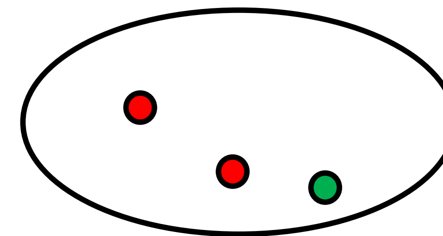


Heuristic

- Finally, add all candidate solutions to un-searched pool for the next iteration



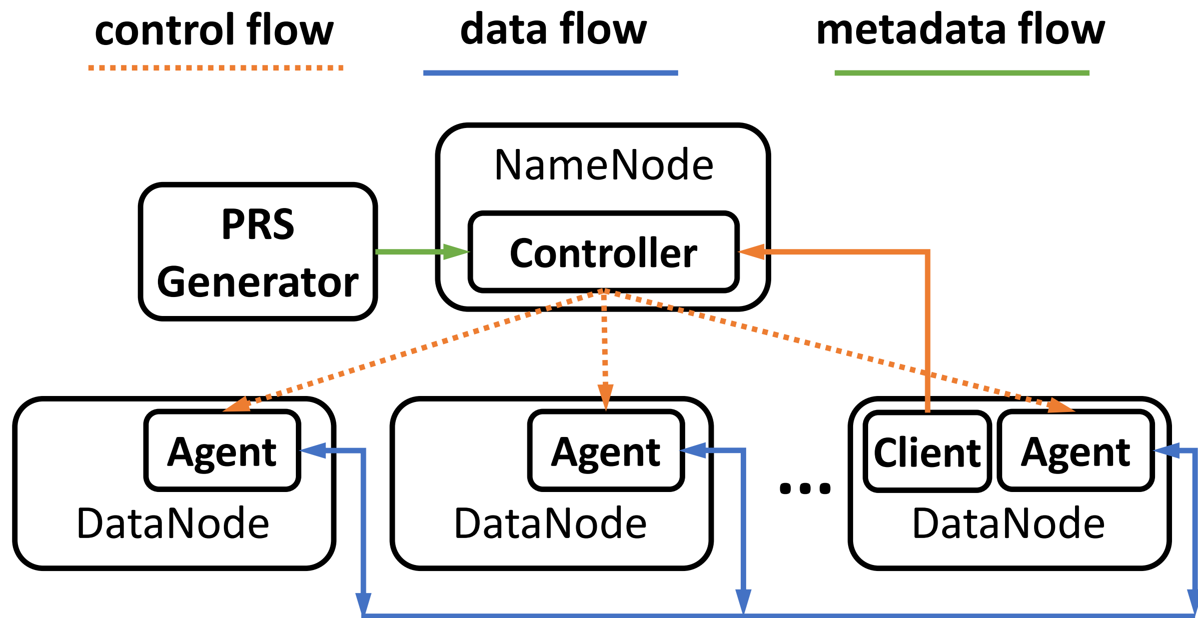




Heuristic

- Repeat steps 2 and 3 until un-searched pool is empty
- Return the point with least maximum repair load in candidate pool
  - Approximate MLP

ParaRC



- Built on Hadoop 3.3.4 HDFS
- PRS Generator
  - Pre-computes approximate MLP **offline** for all failure combinations
- Controller
  - Maintains stripe metadata
  - Parses pECDAG
  - Coordinates repair among agents
- Agent
  - Performs actual repair

Finding Approximate MLP

(maximum repair load, repair bandwidth)

(n,k,w)	**RP**	**Clay**	**Approximate MLP**	**Heuristic**	**Brute-force**
(4,2,4)	(256,512)	(384,384)	(320,448)	1.8 s	264.1 s
(12,8,64)	(256,2048)	(704,704)	(264,1224)	425.9 s	-
(14,10,256)	(256,2560)	(832,832)	(271,1609)	57.2 h	-
(16,12,256)	(256,3072)	(960,960)	(281,1774)	61.9 h	-

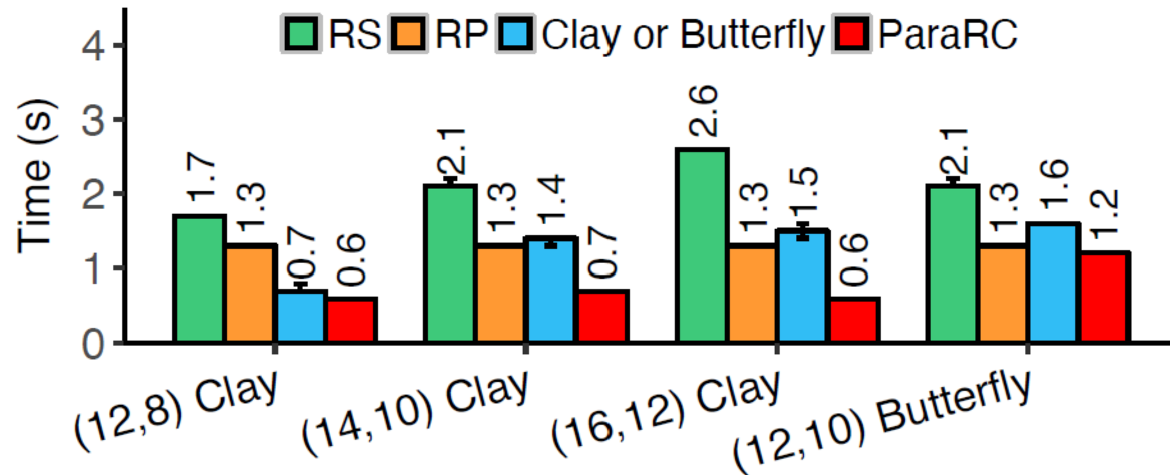
(a) Clay codes

(n,k,w)	**RP**	**Butterfly**	**Approximate MLP**	**Heuristic**	**Brute-force**
(6,4,8)	(256,1024)	(640,640)	(256,896)	0.3 s	34.2 s
(12,10,512)	(256,2560)	(1408,1408)	(297,2216)	31.64 h	-

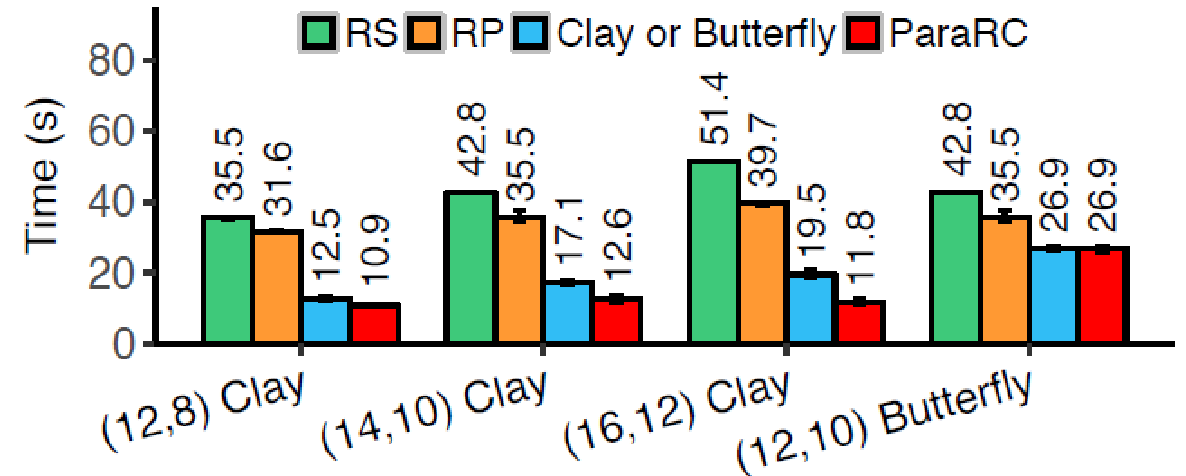
(b) Butterfly codes

- Approximate MLP balances repair bandwidth and maximum repair load, while our heuristic has feasible running time

ParaRC Performance on Alibaba Cloud



(a) Degraded reads



(b) Full-node recovery

- (16, 12) Clay code
  - ParaRC reduces **degraded read** time by 76.4%, 51.9%, and 59.3%, respectively compared with RS, RP, and Clay
  - ParaRC reduces **full-node recovery** time 76.9%, 70.2%, and 39.2%, respectively compared with RS, RP, and Clay

Conclusions

- ParaRC is a parallel repair framework for MSR codes
  - Exploits sub-packetization and balances the trade-off between repair bandwidth and maximum repair load
- Future work:
  - Theoretical analysis of trade-off
  - Co-design with the intra-stripe parallelism and inter-stripe parallelism
  - Design for small blocks and wide stripes
- Source code:
  - http://adslab.cse.cuhk.edu.hk/software/pararc