ShuffleWatcher : Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters

Faraz Ahmad[†*]

Srimat T. Chakradhar[‡]

Anand Raghunathan[†]

T. N. Vijaykumar[†]

[†] PURDUE UNIVERSITY

[‡] NEC Laboratories America

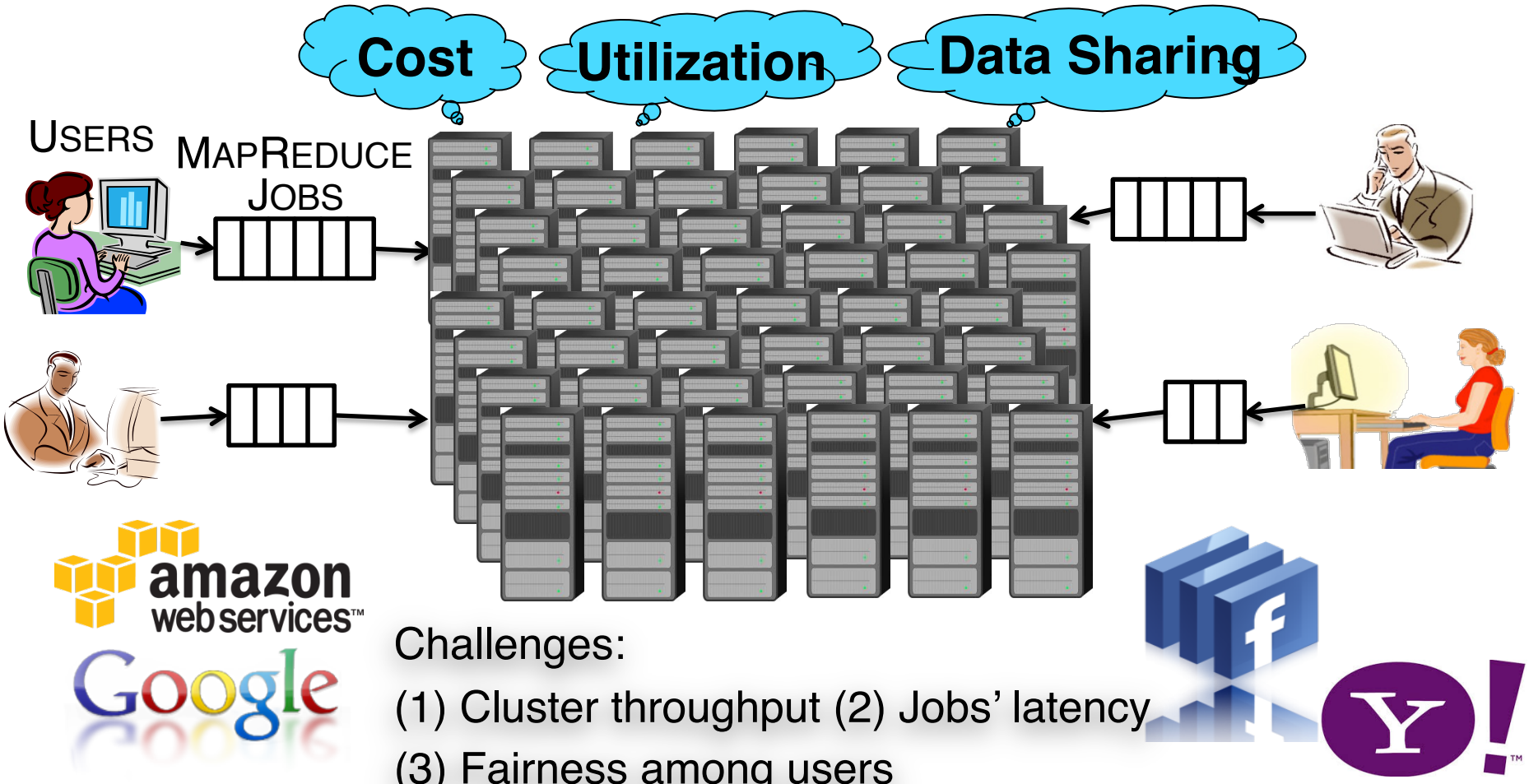
Relentless passion for innovation

[*] TERADATA ASTER

usenix ASSOCIATION

ATC 2014 Philadelphia, PA

Multi-tenancy in MapReduce Clusters



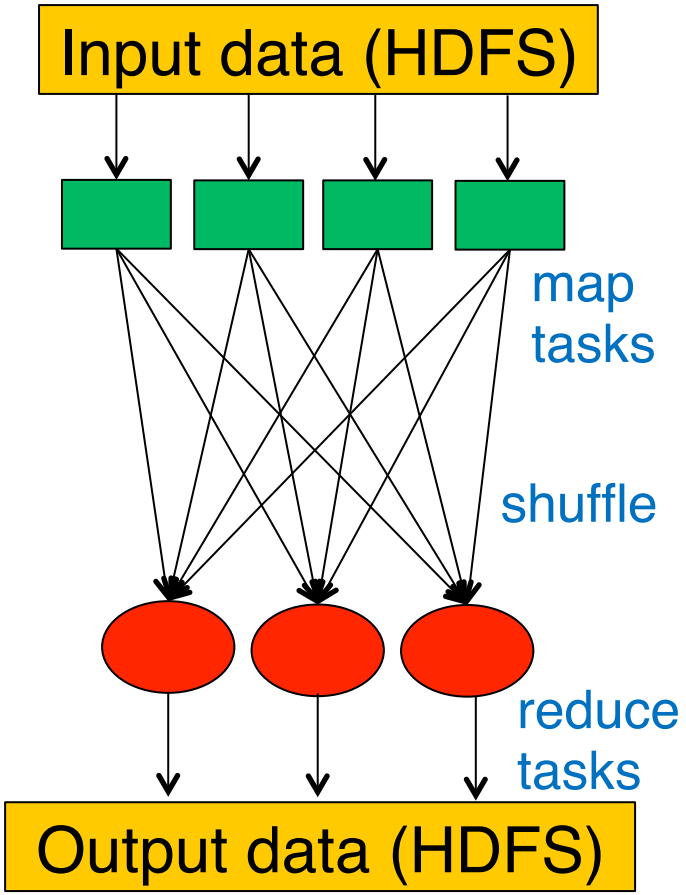
Our contribution: We achieve high throughput and low latency while maintaining fairness among users

Significance of Shuffle in Multi-tenant Clusters

- Shuffle
  - All-Map-to-All-Reduce communication
- Impact of Shuffle-heavy jobs
  - 60% jobs in Yahoo, 20% jobs in Facebook are Shuffle-heavy
  - Shuffle-heavy jobs run much longer than Shuffle-light

  → high network traffic volume
- Impact of Multi-tenancy
  - Multiple concurrent jobs' shuffle

  → high network bisection pressure



Net impact : Low cluster throughput / high jobs' latency

Related Work: Multitenant Scheduling

- Targeting fairness
  - FIFO, Capacity, Fair (Hadoop)
  - Dominant Resource Fairness [NSDI '11]
- Targeting throughput
  - Delay Scheduler [EuroSys '10], Quincy [SOSP '09]
  - Optimizes remote Map traffic

Traffic type	Job Type	Traffic volume (% of total)
Remote Map Traffic	Shuffle-heavy	5%
Remote Map Traffic	Shuffle-light	5%
Shuffle	Shuffle-heavy	78%
Shuffle	Shuffle-light	12%

Our contribution : We improve throughput by focusing on Shuffle while maintaining fairness among users

ShuffleWatcher: Contributions

- Achieves high throughput and low job latency by shaping and reducing Shuffle traffic
- Leverages factors unique to multi-tenancy
- Exploits trade-off : intra-job concurrency vs. Shuffle locality
- Employs three mechanisms :
  - Network-Aware Shuffle Scheduling (NASS) (traffic shaping)
  - Shuffle-Aware Map Placement (SAMP) (traffic reduction)
  - Shuffle-Aware Reduce Placement (SARP) (traffic reduction)
- Keeps the underlying fairness policy intact

ShuffleWatcher achieves 46% higher throughput, 32% lower job latency, 48% reduced traffic compared to Delay Scheduler

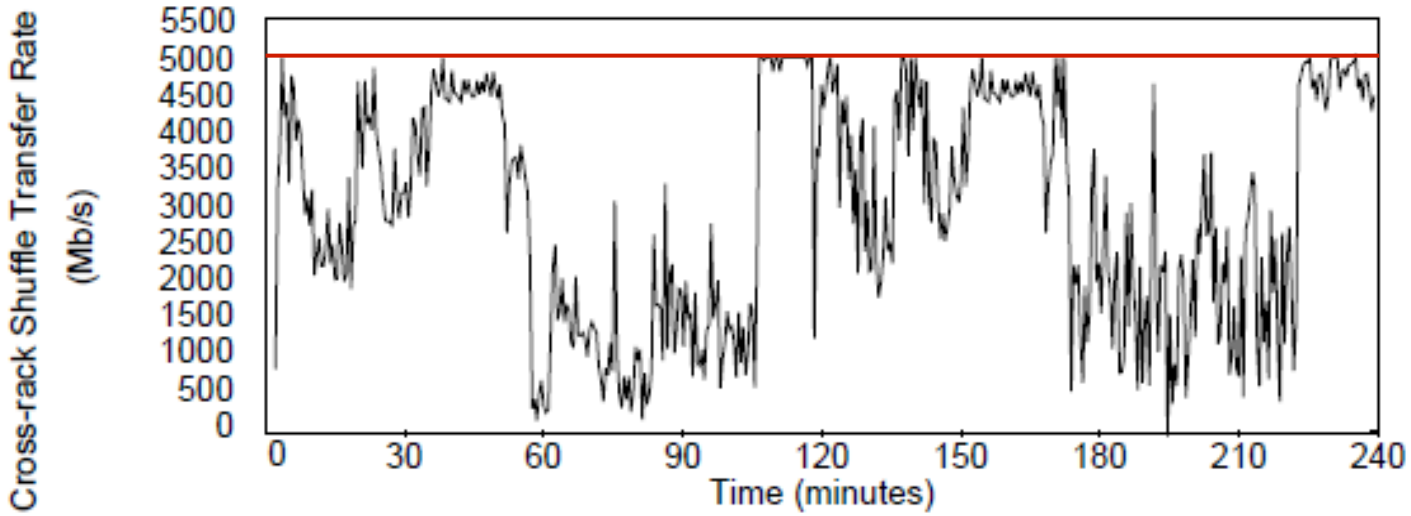
Outline

- Introduction
- ShuffleWatcher mechanisms
  - Network-aware Shuffle Scheduling (NASS)
  - Shuffle-aware Reduce Placement (SARP)
  - Shuffle-aware Map Placement (SAMP)
- Experimental Evaluation
- Conclusion

Network-Aware Shuffle Scheduling (NASS)

Observation #1 :

- Network not saturated all the time
  - 40% jobs in Yahoo, 80% jobs in Facebook are Shuffle-light



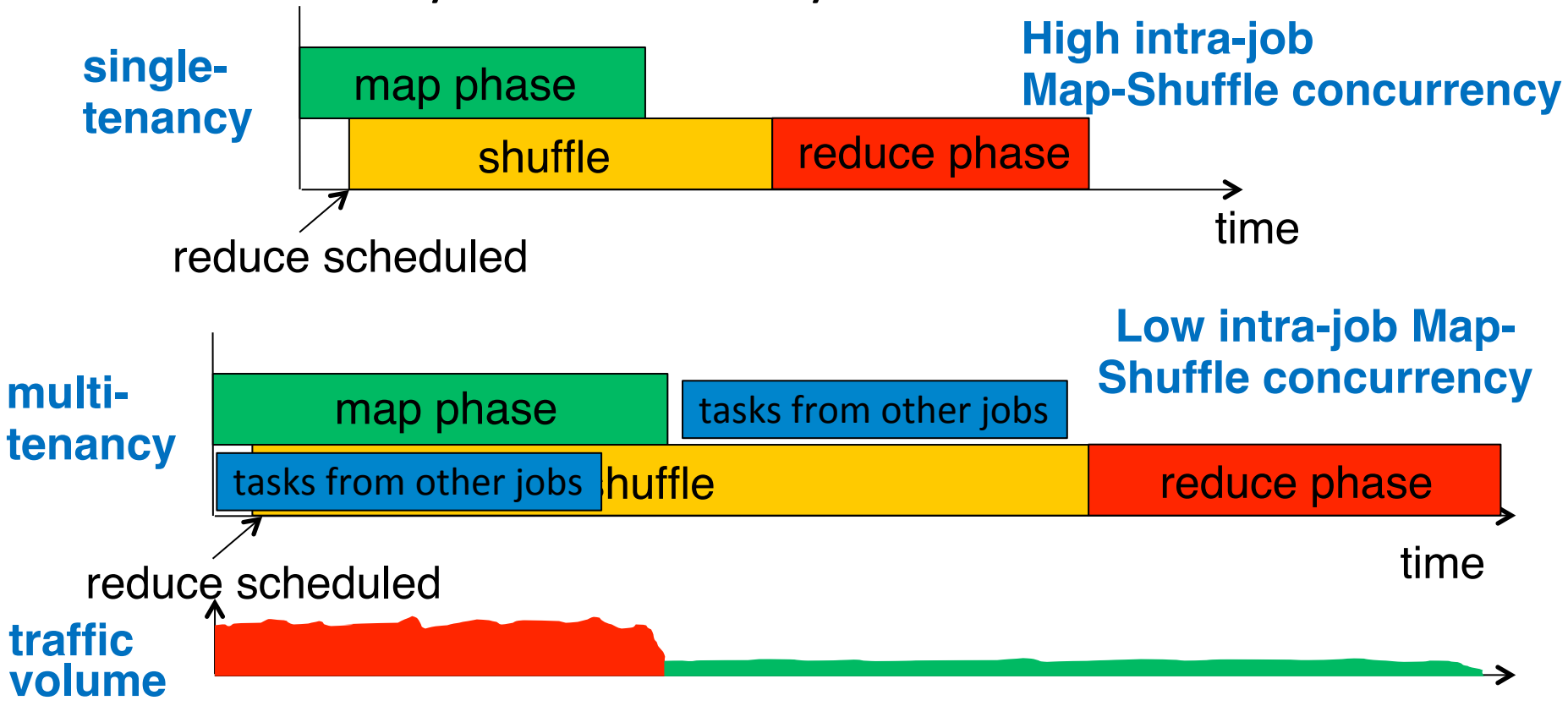
Shuffle profile in 100-node Amazon EC2 cluster (Fair Scheduler)

  - Provides an opportunity to shape traffic.
- Simple delaying a job – not an option
  - *fairness* compromised

Network-Aware Shuffle Scheduling (NASS) (contd.)

Observation #2 :

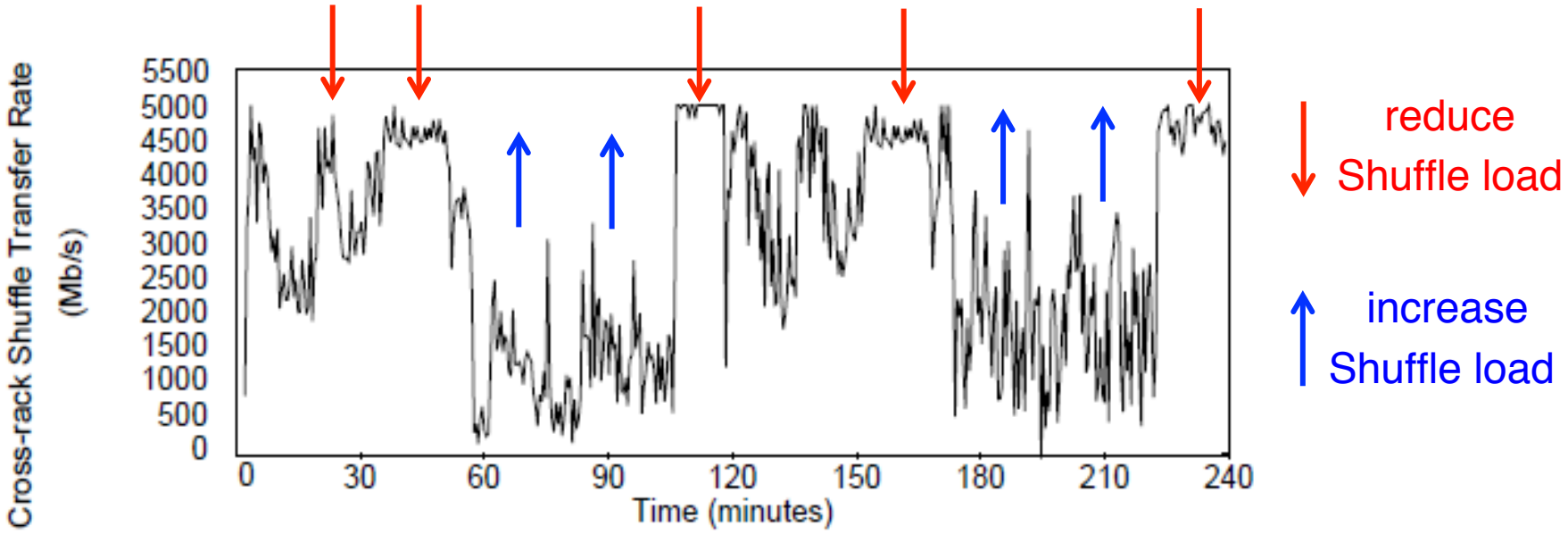
- Multi-tenancy offers flexibility in Shuffle schedule



Shuffle may be delayed in multi-tenancy, if needed, without loss of communication-computation overlap

Network-Aware Shuffle Scheduling (NASS) (contd.)

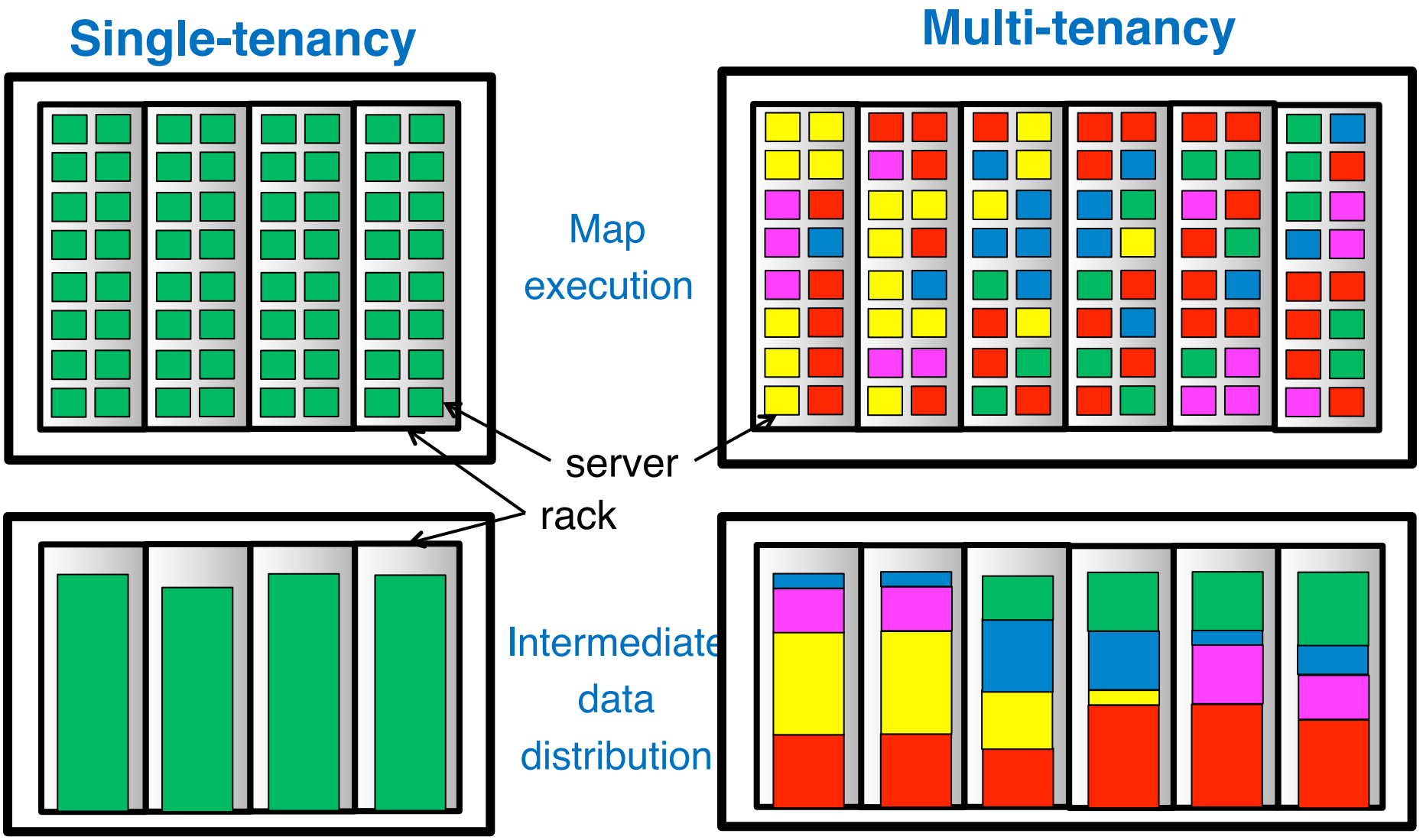
- Contribution:
  - Delay a job's shuffle at high network loads to shape traffic



  - At high load, schedule other tasks of the same user that do not stress network.

NASS achieves **high throughput** while maintaining **fairness**

Shuffle-Aware Reduce Placement (SARP)

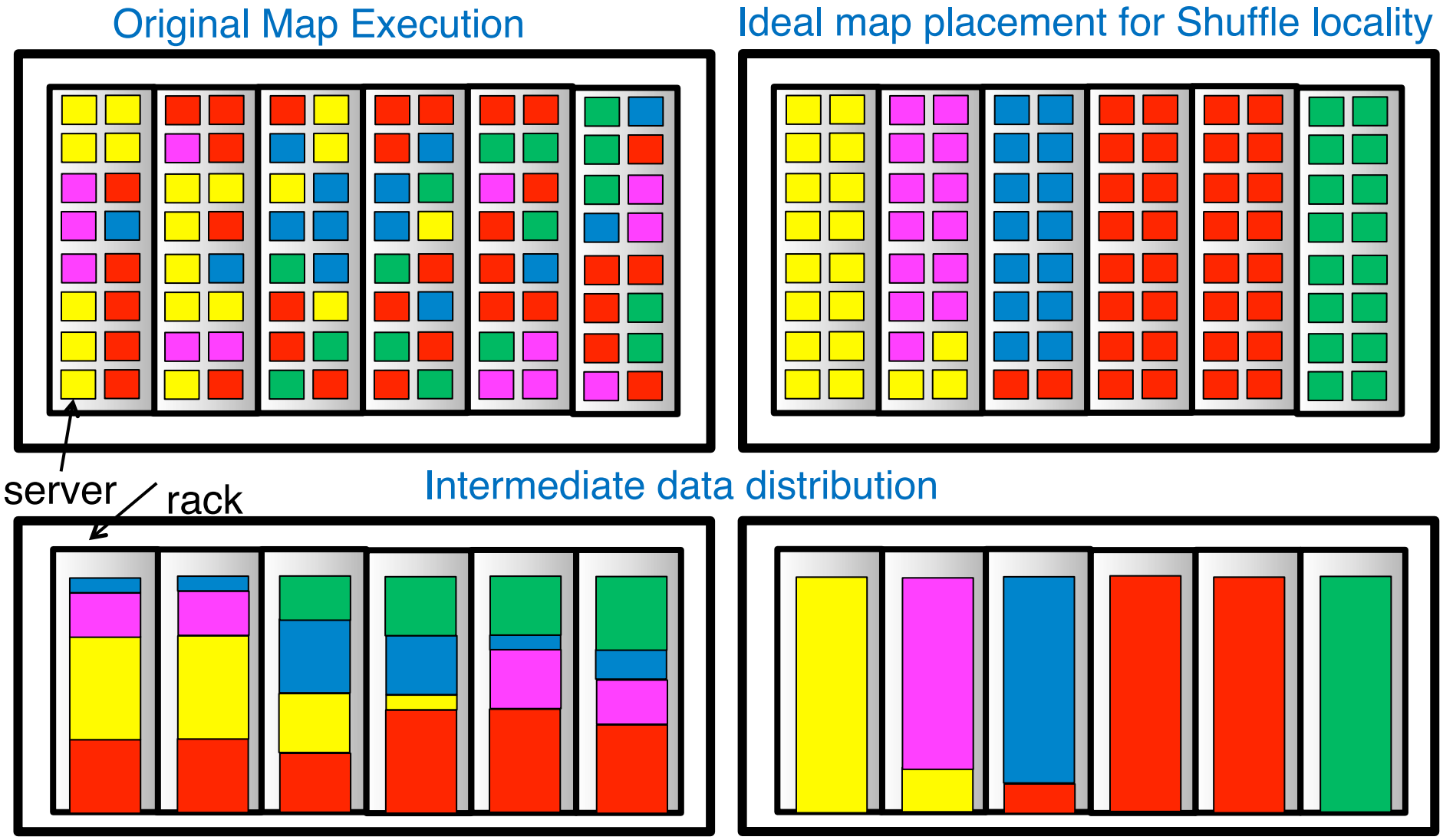


Shuffle-Aware Reduce Placement (SARP) (contd.)

- Observation:
  - A job's intermediate data is likely to be *skewed*
    - Multiple jobs run concurrently
  - Exploit NASS's delayed Shuffle to improve *locality*
    - Intermediate data locations are known

- Contribution:
  - Assign Reduce tasks to racks with more intermediate data
    - Improves Shuffle *locality* → lowers cross-rack Shuffle traffic

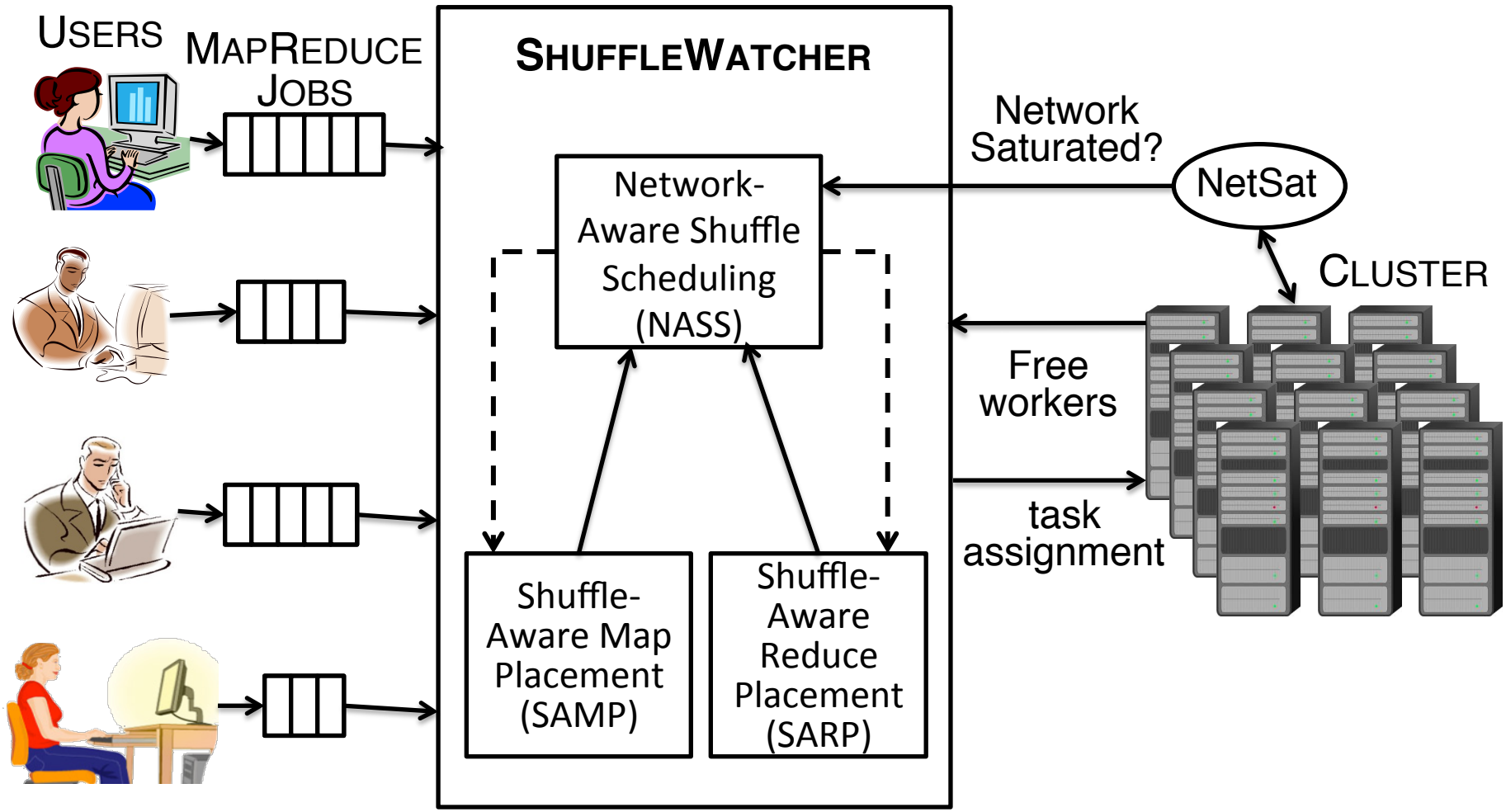
Shuffle-Aware Map Placement (SAMP)



Shuffle-Aware Map Placement (SAMP) (contd.)

- Observation:
  - Input data redundancy provides flexibility in Map scheduling
- Contribution:
  - Localize Map tasks to fewer racks to improve Shuffle locality
    - Provides further opportunities for SARP to localize Shuffle
  - Lowers the sum (Remote Map Traffic + cross-rack Shuffle)
    - May incur some remote map traffic for larger savings in Shuffle

ShuffleWatcher – Overall Picture



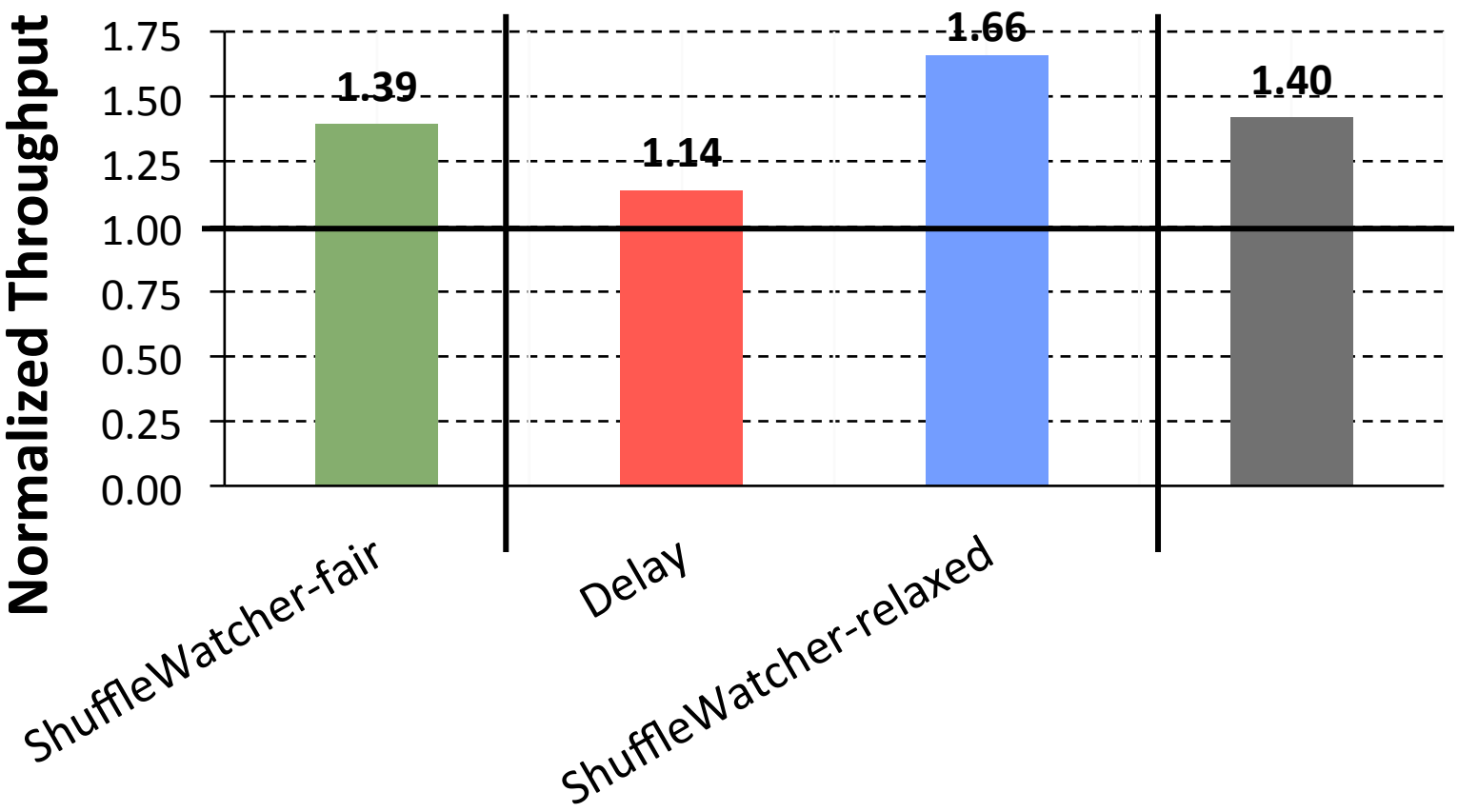
Outline

- Introduction
- ShuffleWatcher mechanisms
  - Network-aware Shuffle Scheduling (NASS)
  - Shuffle-aware Reduce Placement (SARP)
  - Shuffle-aware Map Placement (SAMP)
- Experimental Evaluation
- Conclusion

Experimental Methodology

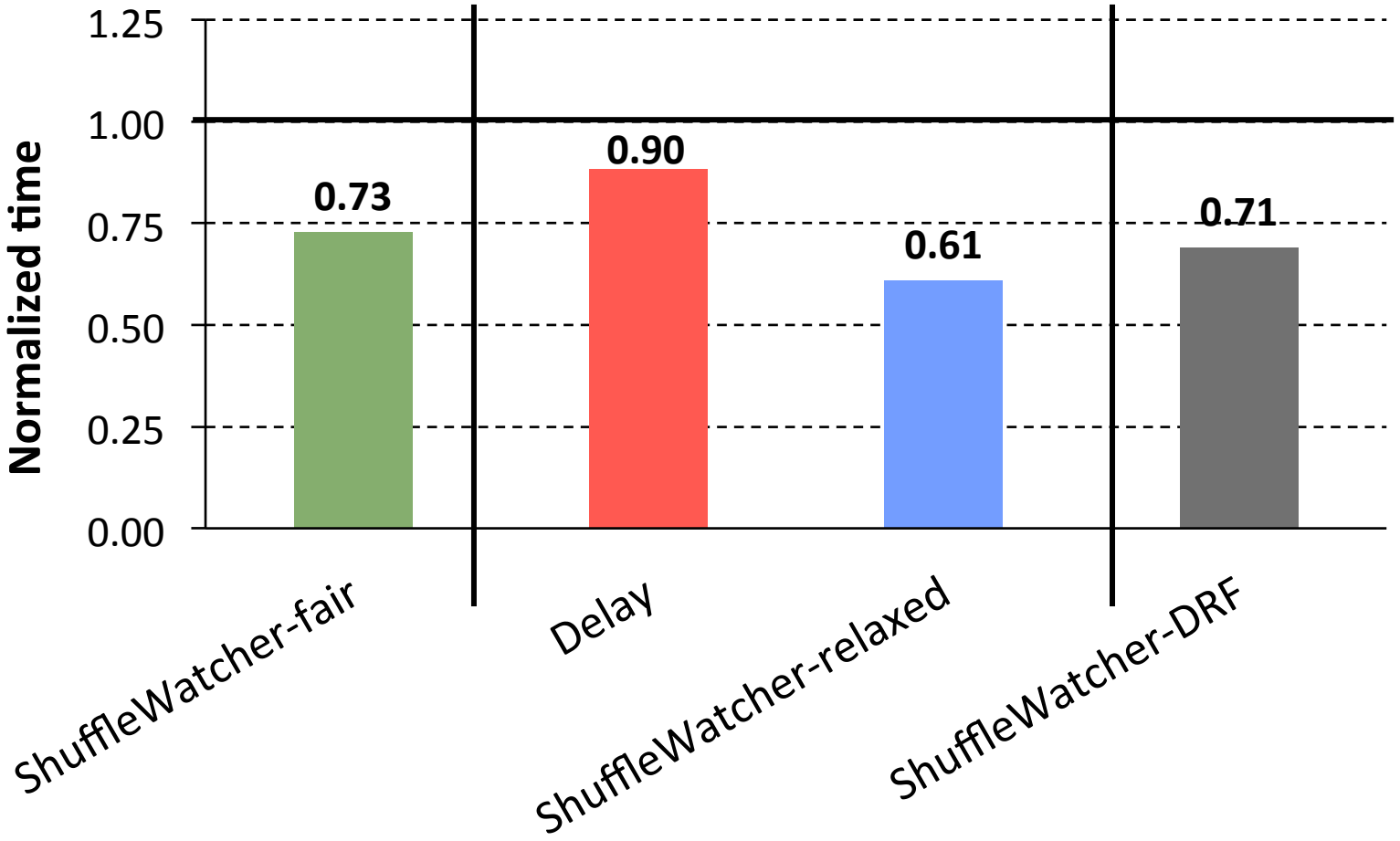
- Baseline:
  - Fair Scheduler [Hadoop implementation]
  - DRF Scheduler [developed in-house as per NSDI '11]
  - Delay Scheduler [Eurosys '10, Hadoop implementation]
- Platform:
  - 100-node Amazon EC2 cluster
    - 4 virtual cores per node, 15 GB memory
    - 10 racks of 10 nodes each
    - 50 Mbps cross-rack per-node bisection bandwidth
- Workloads:
  - Job submission : exponential distribution (mean arrival rate : 40 s)
  - 30 users, jobs from 12 MapReductions, run for 4 hours
    - 40% shuffle-heavy, 20% shuffle-medium, 40% shuffle-light
  - Job sizes : mix of small, medium, large sizes vary from 100 MB-1 TB

Throughput Comparison



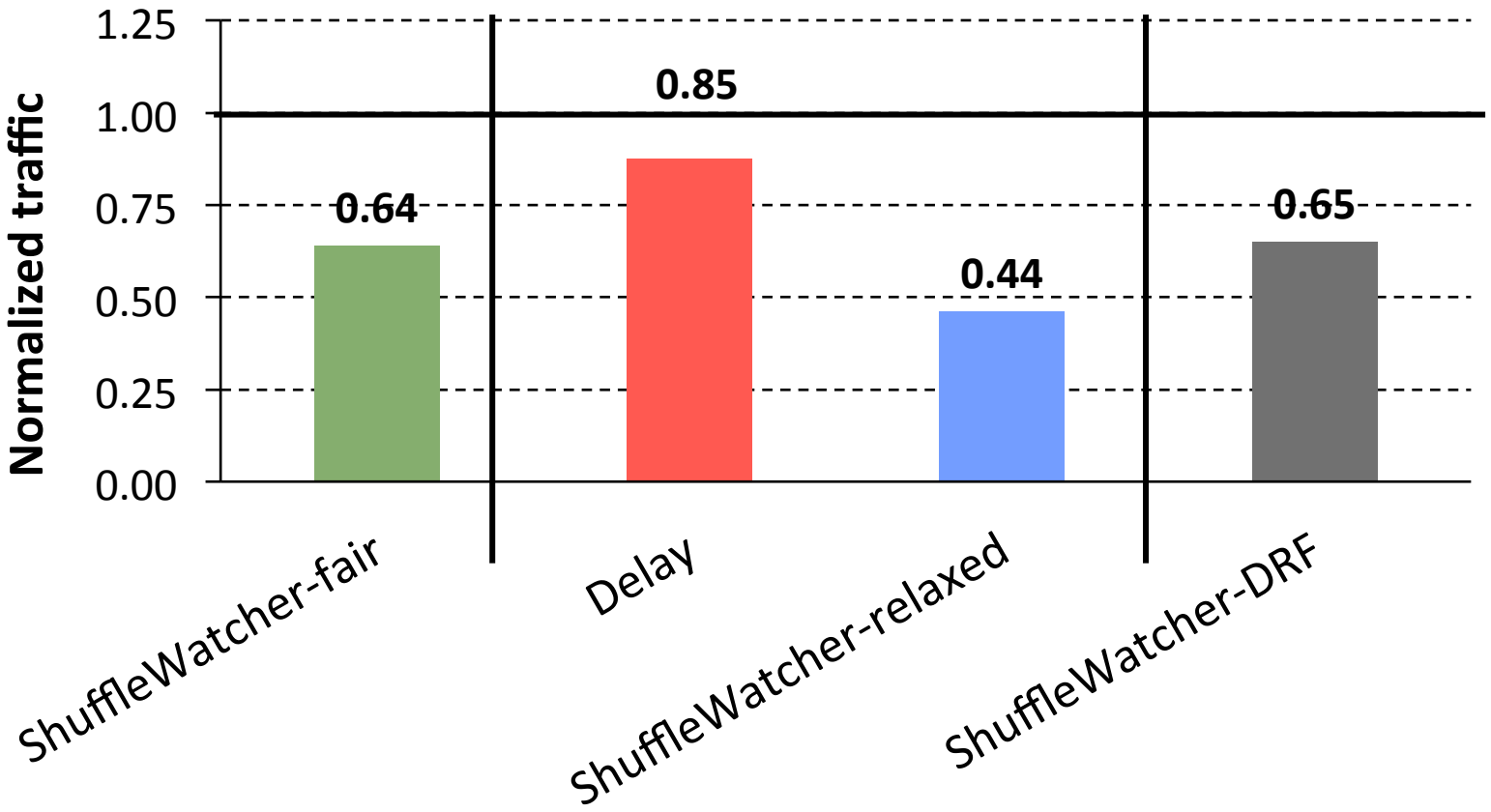
ShuffleWatcher is independent of fairness policy and improves throughput (39%, 46%, 40%) over multiple schemes

Jobs Latency (turn-around time) Comparison



ShuffleWatcher improves latency (27%, 32%, 29%) over multiple schemes

Cross-rack Traffic Comparison



ShuffleWatcher achieves significant traffic reduction (36%, 48%, 35%) over multiple schemes

Conclusion

- Shuffle significantly impacts multi-tenancy performance
- ShuffleWatcher achieves high throughput by shaping and reducing Shuffle traffic while maintaining fairness
  - Leverages factors unique to multi-tenancy
  - Trades-off intra-job concurrency for Shuffle locality
- Employs three mechanisms :
  - Network-Aware Shuffle Scheduling (NASS) for traffic shaping
  - Shuffle-Aware Map Placement (SAMP) and Shuffle-Aware Reduce Placement (SARP) for traffic reduction
- ShuffleWatcher achieves 46% higher throughput, 32% lower job latency, 48% reduced traffic compared to Delay Scheduler

Thanks

Back up slides

NASS Algorithm

called when a worker requests for a task

```
1.  Select user based on fairness policy
2.  if (NetworkSaturated) {
3.     find a task from queue in the following order:
4.        Map tasks preferred by SAMP
5.        Local Map task of any job
6.        Reduce task of any Shuffle-light job
7.        Reduce tasks preferred by SARP
8.        Reduce task of head-of-queue job
9.        Any available task of user
10. }
11. else {
12.    find a task from queue in the following order:
13.       Reduce tasks preferred by SARP
14.       Reduce task of any Shuffle-heavy job
15.       Reduce task of head-of-queue job
16.       Map tasks preferred by SAMP
17.       Local Map task of any job
18.       Any available task of user
19. }
```

SARP Algorithm

Invoked when a job J schedules its Reduce tasks

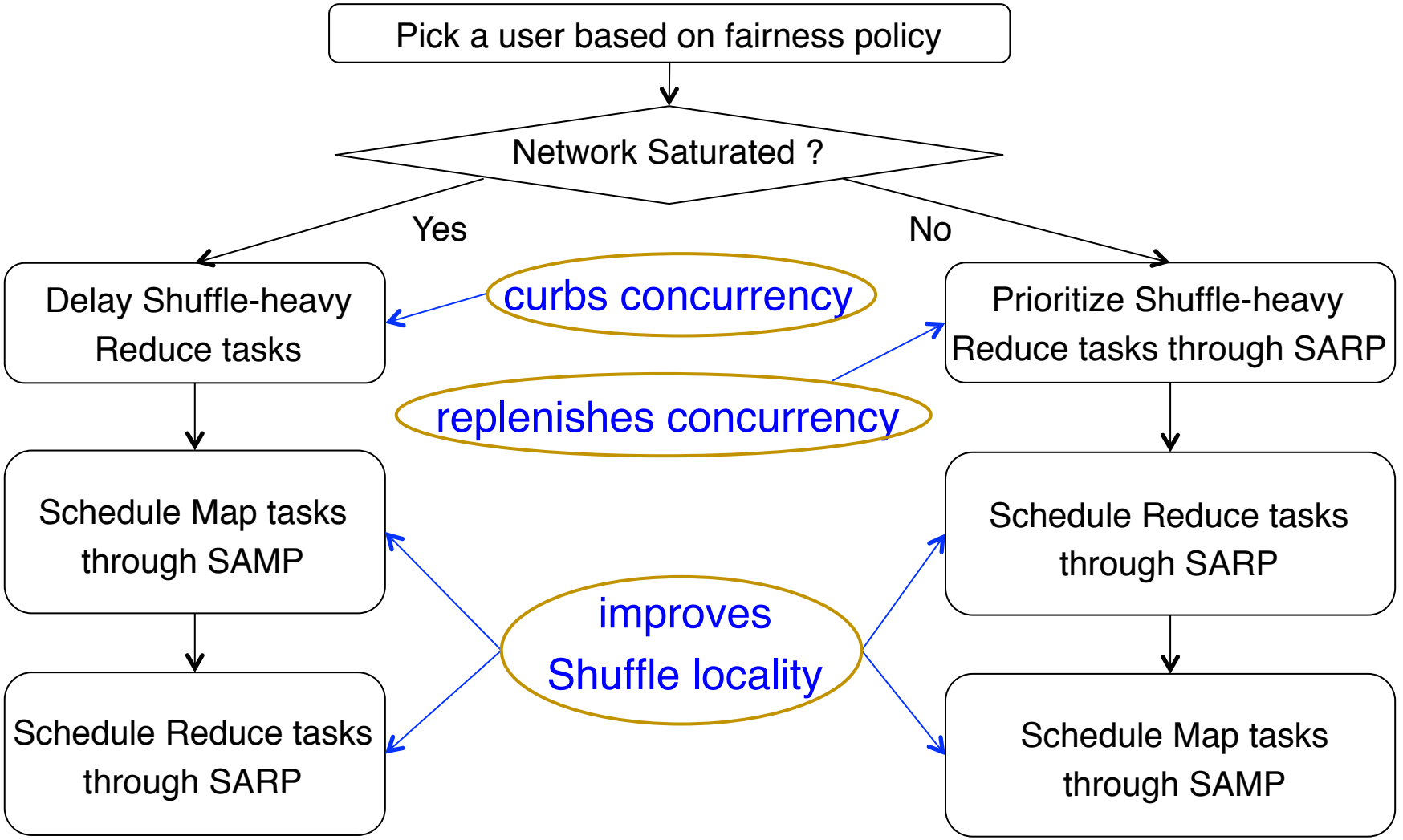
```
1.if (fraction of Maps completed for J > MapCompletionThreshold) {
2.    for each rack R {
3.        PreferredReducesPerRack[R] = NumReduces *
              (Intermediate data of job J on rack R / total Intermediate
              data of job J)
      }
   } else {
4.    for each rack R
5.        PreferredReducePerRack[R] = TentativeReducePerRack[R]}
   }
```

SAMP Algorithm

called when a new job J is submitted

1. Sort racks in decreasing order of input data for J followed by increasing order of number of preferred jobs
2. TmpMapRacks = {}
3. minCrossRackTraffic = ∞
4. do {
5. remove first rack R in sorted list and add to TmpMapRacks
6. compute TmpReducesPerRack //assumes SARP
7. estimate minCrossRackTraffic =
   remote Map traffic + cross-rack Shuffle
8. } while (minCrossRackTraffic decreases)
9. PreferredMapRacks ← TmpMapRacks
10. TentativeReducesPerRack ← TmpReducesPerRack

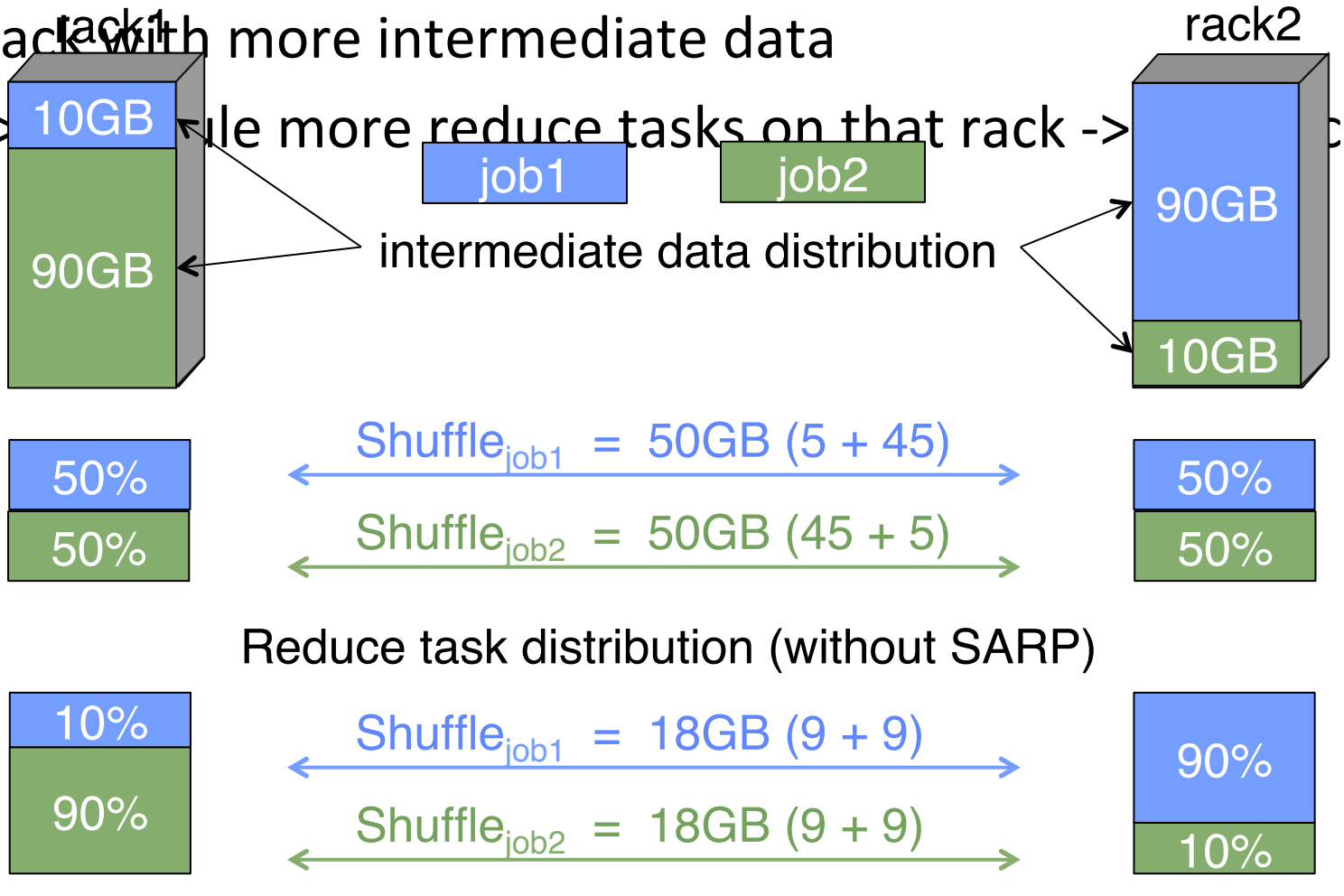
Network-aware Shuffle Scheduling (NASS)



Better critical resource (network) utilization -> higher throughput

Shuffle-aware Reduce Placement (SARP)

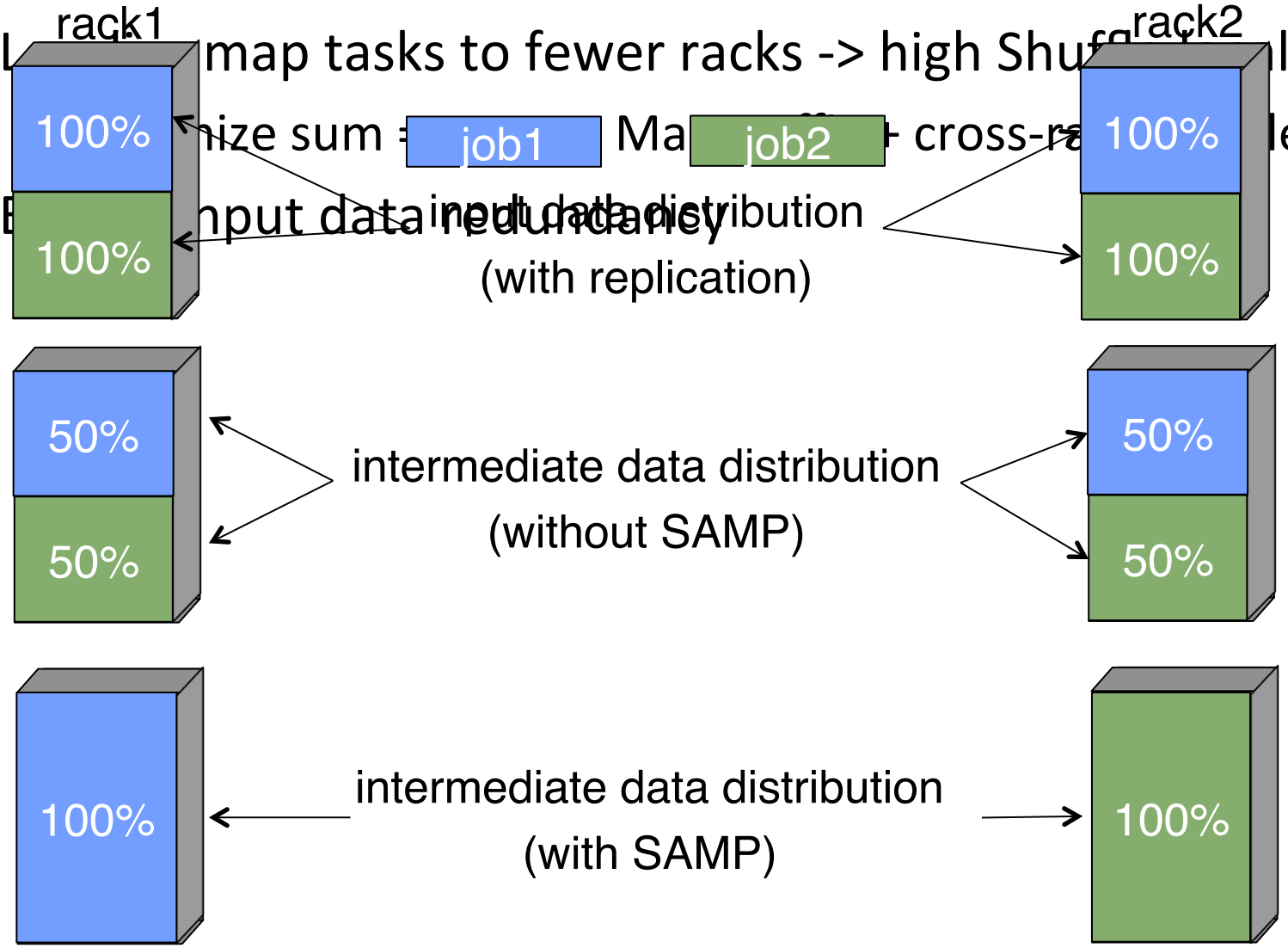
- Rack with more intermediate data
  -> ule more reduce tasks on that rack -> cality



Cross-rack traffic reduction : (100 – 36)/100 = 64%

Shuffle-aware Map Placement (SAMP)

- L map tasks to fewer racks -> high Shuf lity
  ize sum = Ma + cross-ra le
- E nput data redundancy



Cross-rack traffic reduction (with SARP) : (100 – 0)/100 = 100%

Workload characteristics

- Exponential distribution
  - job size, job type picked at random at submission time (consistent with Facebook, Yahoo traces [Mascots'11]).
- Mean job arrival rate
  - Highest throughput for base case (delay scheduler)
  - 40 seconds

Job Category	Benchmarks (Frequency)
Shuffle-heavy	Terasort(5%), Ranked-inv-index(10%), Self-join(10%), Word-sequence-count(10%), Adjacency-list(5%)
Shuffle-medium	Inverted-index(10%), Term-vector(10%)
Shuffle-light	Grep(15%), Wordcount(10%), Classification(5%), Histo-movies(5%), Histo-ratings(5%)

Input Job Sizes	%jobs	Input Job Sizes	%jobs
<100MB	20%	100-200GB	10%
100MB-1GB	19%	200-500GB	7%
1-20GB	21%	500GB-1TB	8%
20-100GB	10%	> 1TB	5%

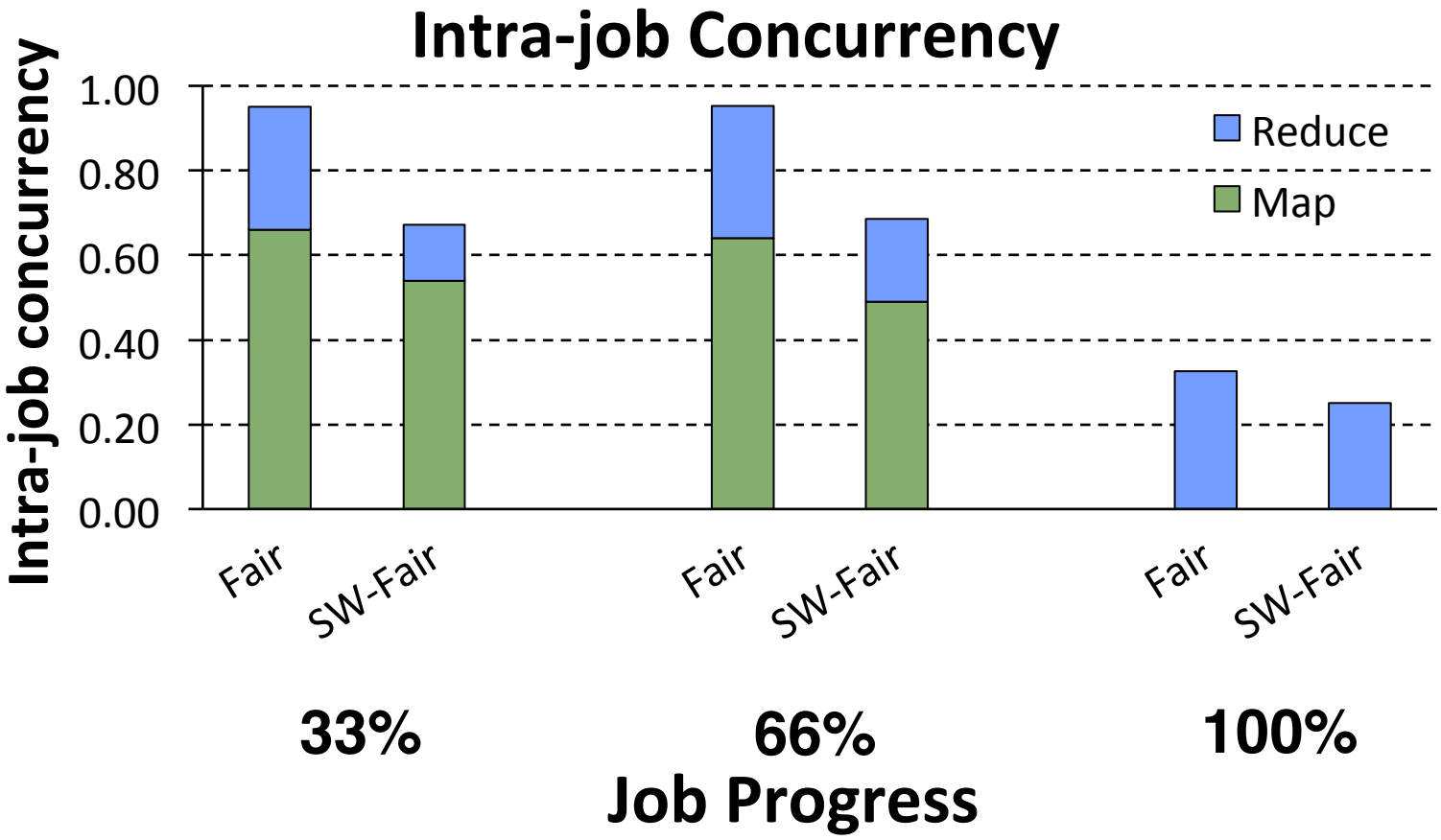
Benchmarks

Job Category	Benchmarks (Frequency)
Shuffle-heavy	Terasort(5%), Ranked-inv-index(10%), Self-join(10%), Word-sequence-count(10%), Adjacency-list(5%)
Shuffle-medium	Inverted-index(10%), Term-vector(10%)
Shuffle-light	Grep(15%), Wordcount(10%), Classification(5%), Histo-movies(5%), Histo-ratings(5%)

Job Sizes Distribution

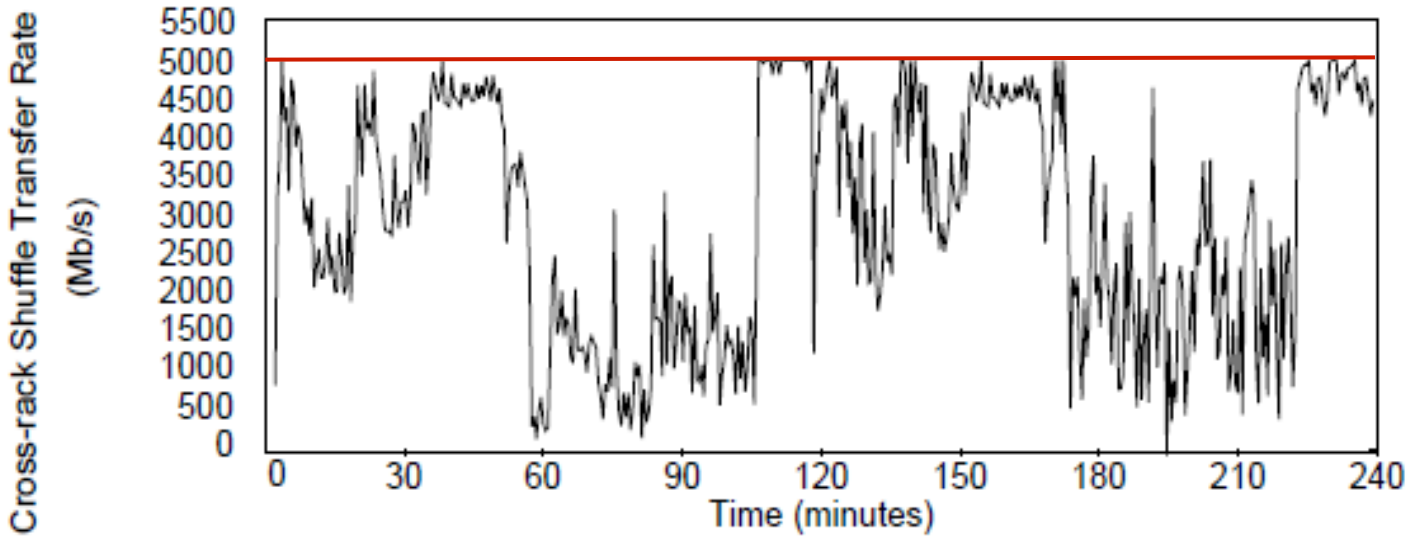
Input Job Sizes	%jobs	Input Job Sizes	%jobs
<100MB	20%	100-200GB	10%
100MB-1GB	19%	200-500GB	7%
1-20GB	21%	500GB-1TB	8%
20-100GB	10%	> 1TB	5%

Performance Comparison (contd.)

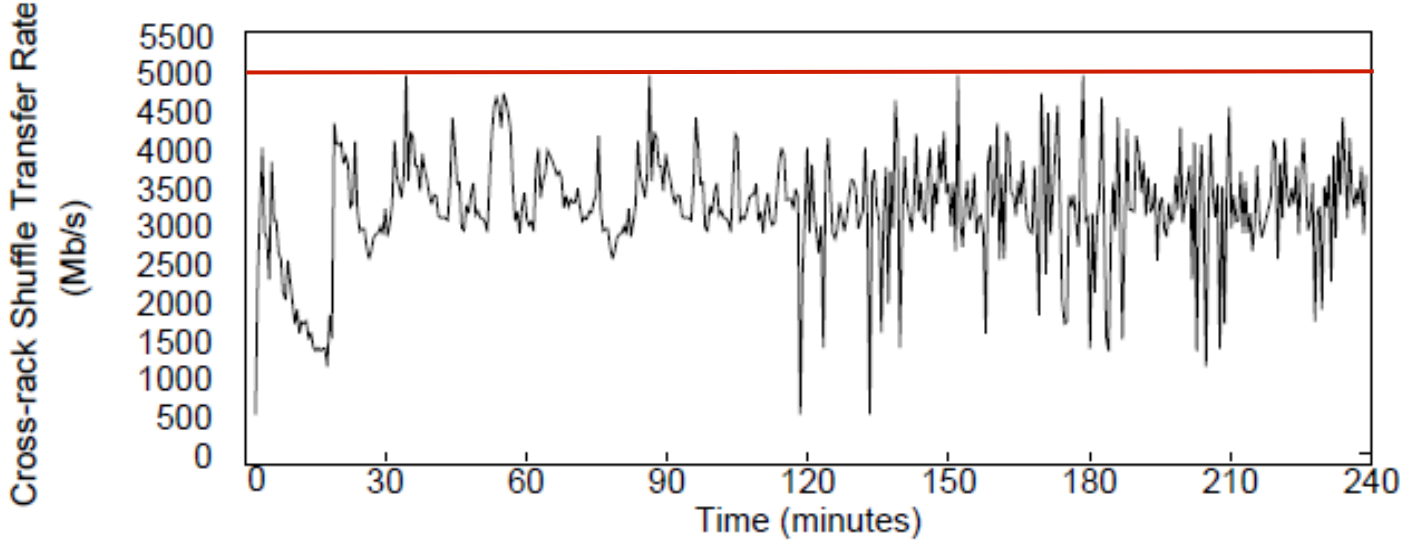


ShuffleWatcher achieves lower intra-job concurrency to improve Shuffle locality

ShuffleWatcher impact on Traffic Shaping

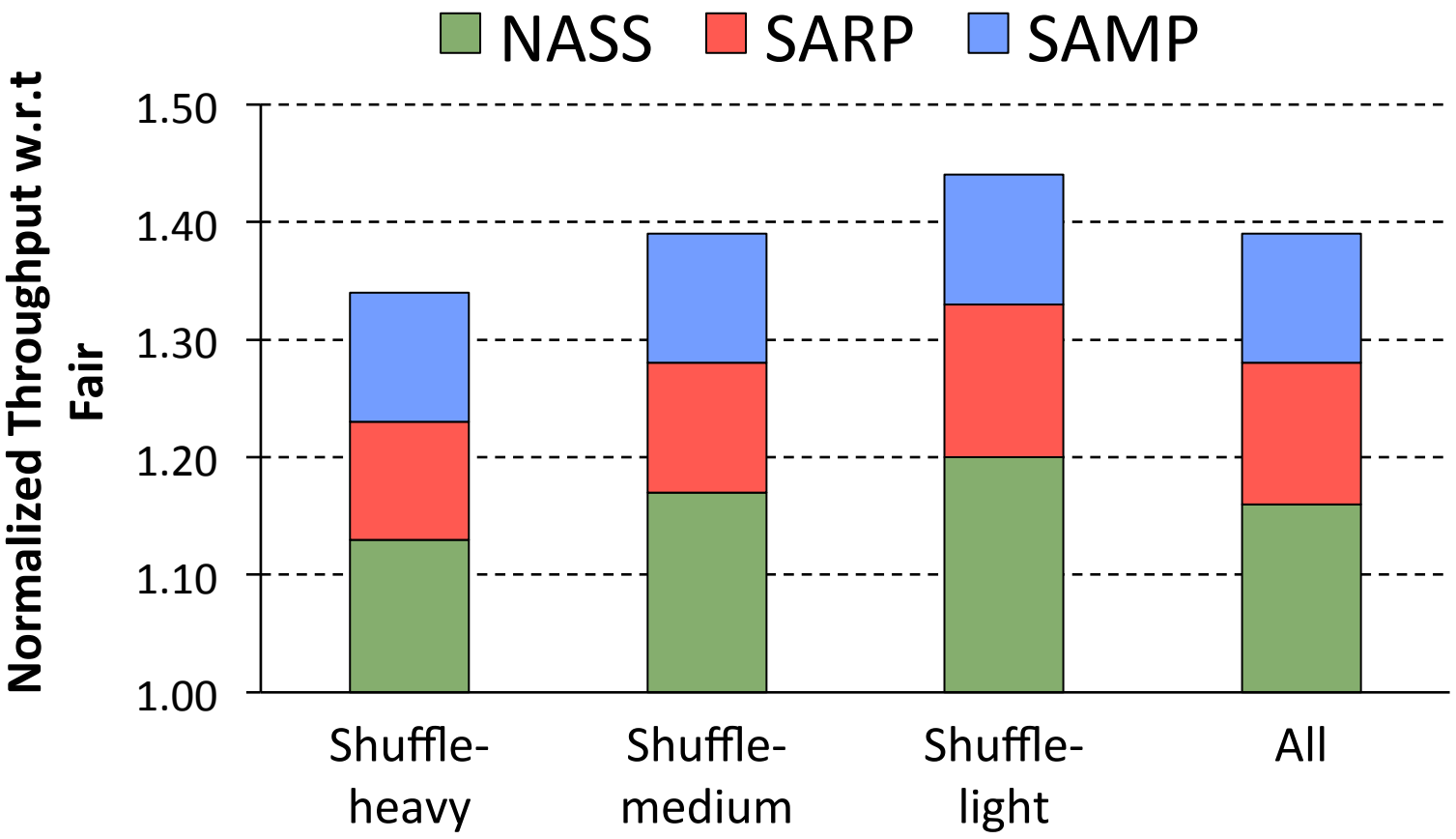


without ShuffleWatcher



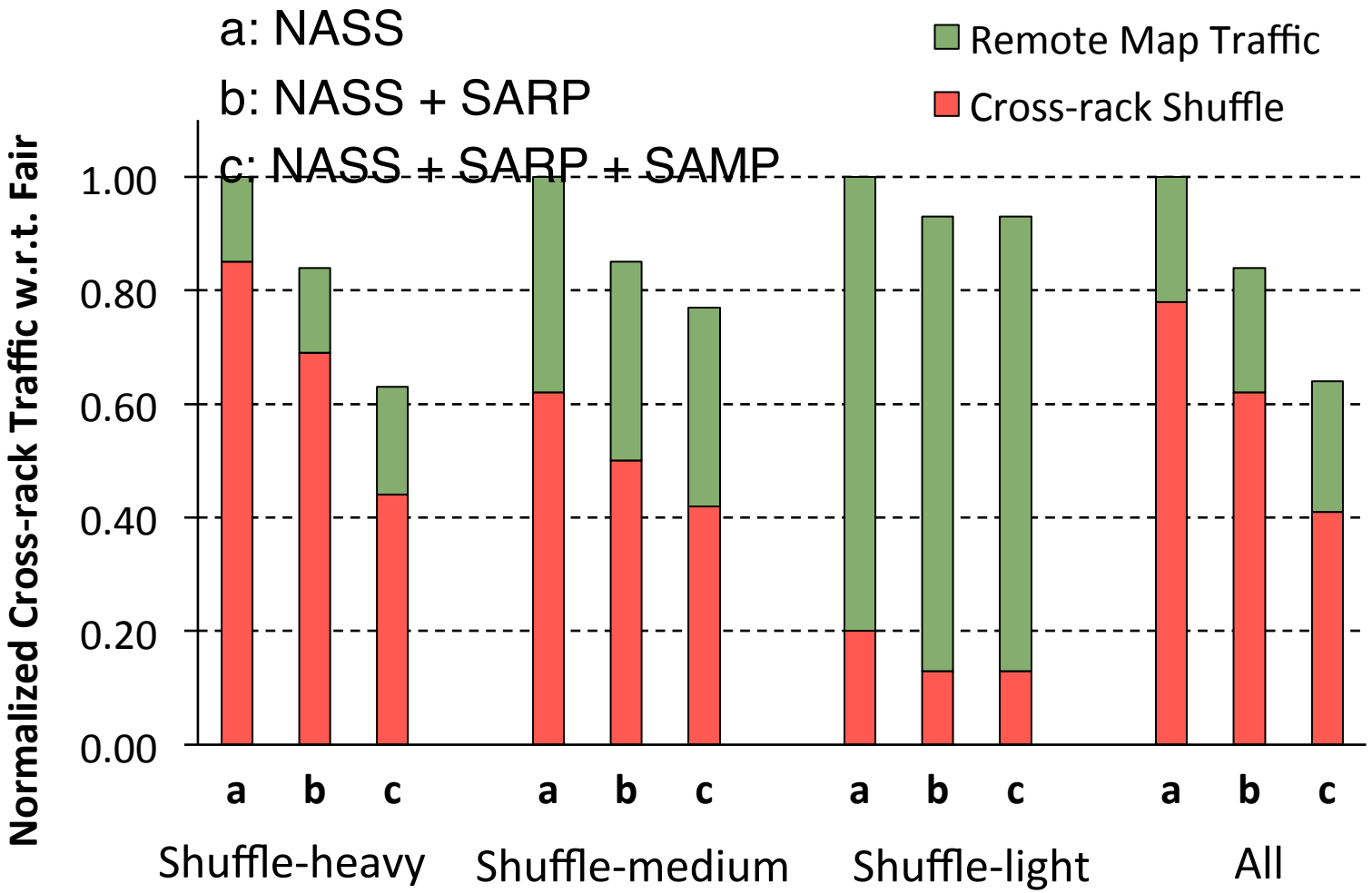
with ShuffleWatcher

Throughput Impact of NASS, SARP and SAMP



All mechanisms contribute significantly to improve throughput

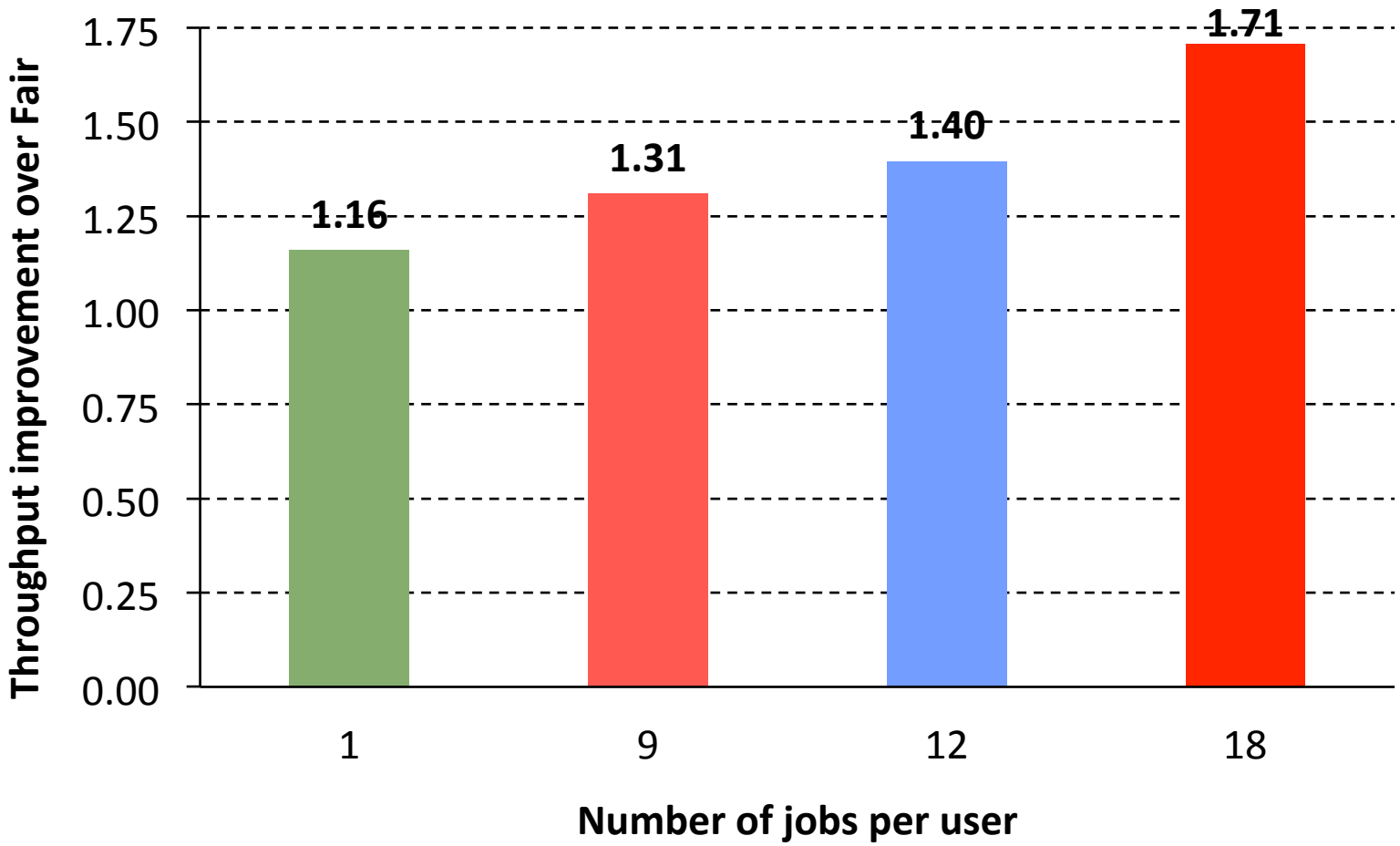
Traffic Reduction achieved by SARP and SAMP



SARP alone reduces overall network traffic by 15%

SARP+SAMP reduce overall network traffic by 36%

Impact of variation in number of jobs per user



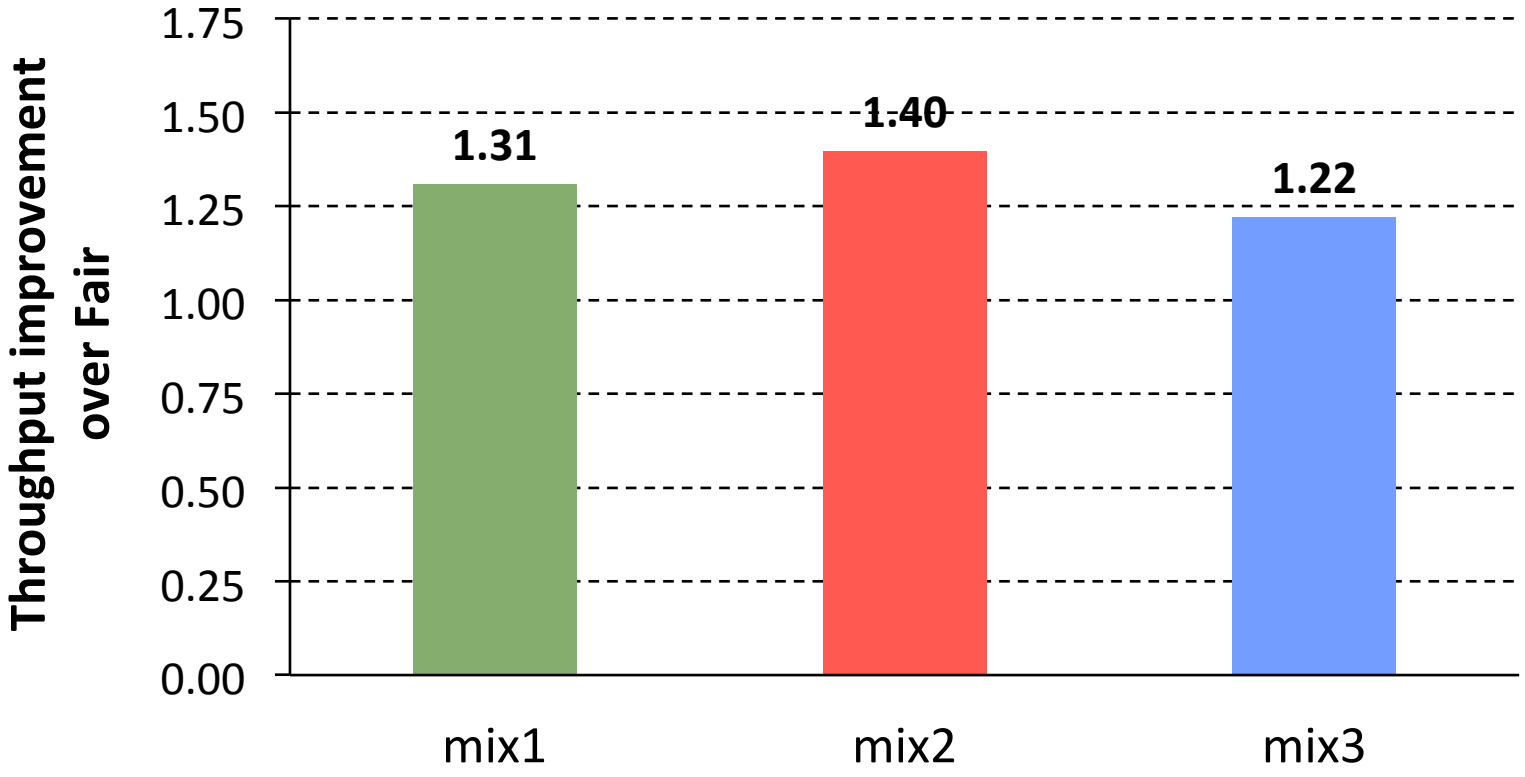
ShuffleWatcher scales well with number of jobs per user and achieves higher improvement with higher number of jobs

Sensitivity to job mix

mix1: 20%,20%,60% (Shuffle-heavy, Shuffle-medium, Shuffle-light)

mix2: 40%,20%,40% (default)

mix3: 60%, 20%,20%



ShuffleWatcher is effective over a wide range of job mixes

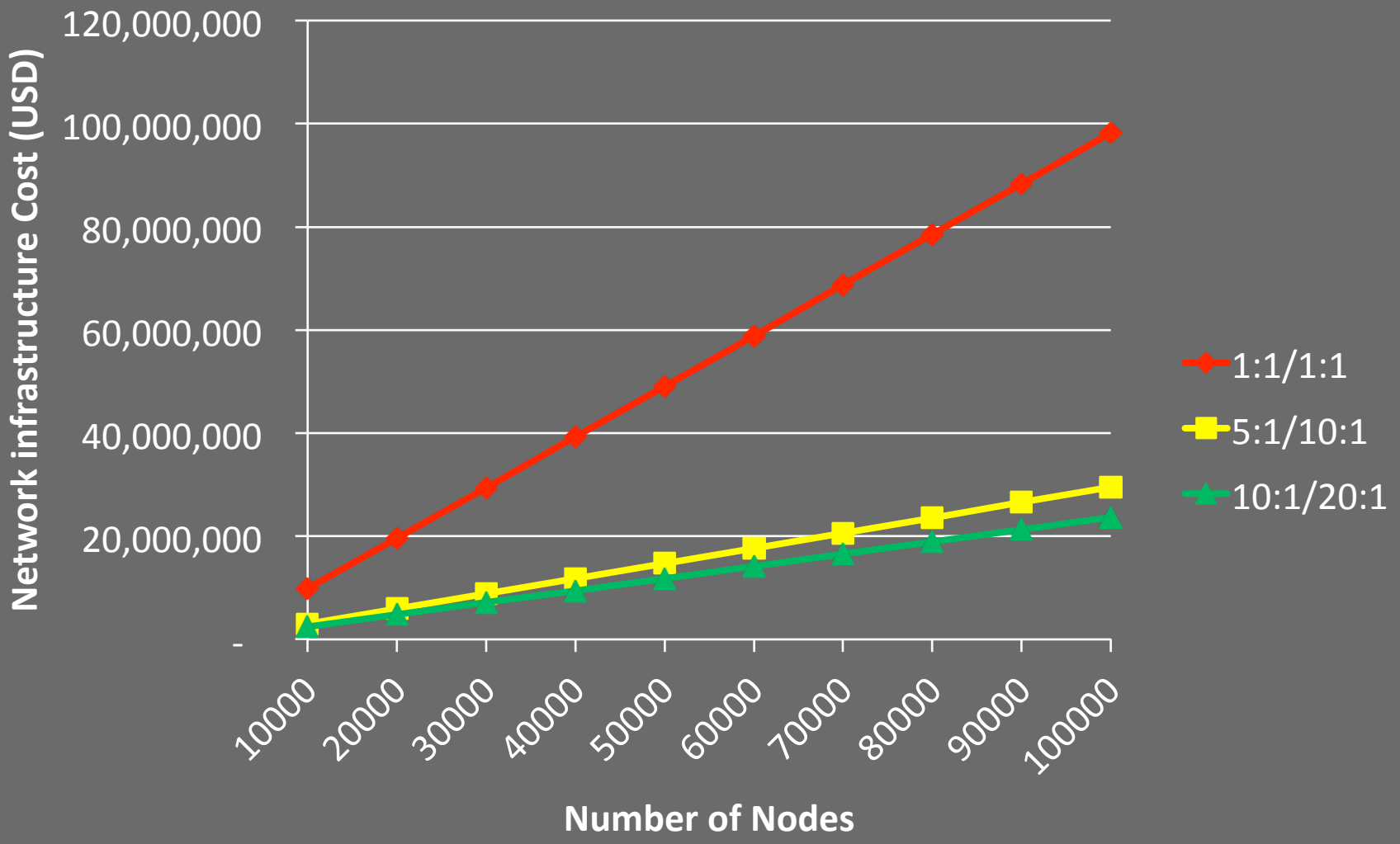
Simpler algorithms to lower Shuffle won't work

- Need to consider the following aspects of scheduling:
  - jobs' latency, fairness, cluster utilization
- Scheduling Reduce tasks based on Intermediate data
  - static decision
  - loses opportunity to overlap with own Map
  - potential increase in latency
  - fixed assignment to racks -> reduced flexibility
- Scheduling Reduce tasks based on Input data distribution
  - cannot exploit additional skew in intermediate data
  - fixed assignment to racks -> reduced flexibility
- Traffic shaping ignored:
  - 40% improvements due to traffic shaping

Bisection Bandwidth

- Bisection bandwidth is a global resource that is hard to scale up with the cluster's compute and storage resources (CPU, memory, disk).
  - On the contrary, network switch and link bandwidth scale with hardware technology.
- Even with recent advances in data center networks, large clusters are typically provisioned for *per-node* bisection bandwidth that is 5-20 times lower than the within-rack bandwidth.
  - Full Bisection bandwidth is an expensive solution.
  - Bisection not saturated all the time.
  - Same cost can be spent to buy additional compute resources.

Full bisection vs. High over-subscription



NetSat daemon

- Determines whether network is saturated or not.
- Cross-rack link bandwidths are fixed to maintain higher oversubscription ratio (5:1) in our case.
  - E.g., 500Mbps if 10 nodes per rack (50 Mbps per node).
  - Nodes can share cross-rack link bandwidth
- Algorithm (determines whether link is saturated):
  i. Receive data from all running reduce tasks for all jobs periodically.
  ii. Receive data from NameNode for remote map and output write traffic
  iii. Update data transfers among racks (link usages).
  iv. Compare to cross-rack link bandwidth limit
  v. Return true if link usage (for n intervals) > NWSaturationThreshold (0.8 * SaturationLimit)
- Shapes traffic more conservatively.
  - Rate limiters per rack.
- Specific topological information or precise congestion monitoring mechanisms can be overlaid
- Can be modified to account for traffic from other applications (such as interactive workloads, MPI jobs) in the cluster.

ShuffleWatcher: where network is not critical

- The concurrency – locality trade-off will likely remain in favor of achieving high concurrency.
- Traffic shaping not needed.
- Traffic reduction still helpful
  - Localized shuffle still faster than dispersed shuffle
  - Remote map tasks is still slower than local map task
  - SARP will follow SAMP if Shuffle not delayed.
- ShuffleWatcher still better than base case.
  - SAMP and SARP remain active.
  - No additional overheads of ShuffleWatcher.

ShuffleWatcher implementation changes

- New multi-tenant scheduler in a separate /contrib folder.
  - ~500-1000 lines of scheduler code.
- Data structures needed:
  - *JobProfiling* (Shuffle-heavy/medium/light) : updated at job completion
    - ShuffleInputRatio (0.3-0.8 ->Shuffle-medium)
  - *NetSat daemon* (for NASS): updated periodically
  - *currentShuffleJobs* (for NASS): list of jobs in shuffle phase
  - *IntermediateDataLoc* (for SARP): updated as map tasks complete
  - *PreferredReducePerRack* (for SARP):updated before scheduling Reduce tasks or when time-window expires for the job.
  - *PreferredMapRacks* (for SAMP):updated at job submission
  - *TentativeReducePerRack* (for SAMP):updated at job submission
  - *FairnessModule* : based on fairness policy to be used.
- Shuffle data transfer information from Reduce tasks
  - At every heartbeat (regular intervals)

DRF Scheduler

- Observation: Each job has different resource requirement
  - Fixed slots waste resources
  - Prepare fine granularity slots for nodes and allocate as many as are needed by the task.
  - Task resource requirement is provided by the user.
- Max-min fairness for multiple resources
  - Equalize the dominant shares of users.
  - Works on CPU, Memory resources.
- Not straight-forward to apply to network resource.
  - Depends upon task locations and data locations which is dynamic.
  - Worst case requirements do not utilize network efficiently

Network Topology (ShuffleWatcher)

- EC2 does not provide precise topological information.
  - Same observed node-to-node bandwidth for all nodes in our cluster.
  - We group 10 nodes per rack and limit cross-rack bandwidth to 500Mbps.
- Uses the hadoop-default notions of local, rack-local and cross-rack traffic.
  - Cross-rack is remote traffic which needs to be minimized.
  - We limit per-node cross-rack bandwidth to 50Mbps which is the expected limit per-node in a large cluster.
  - Mechanisms applicable to a broader range of bisection bandwidths.
- Schemes do not depend upon precise topological information.
  - Generally applicable to all clusters.
  - Topological information can be overlaid if available.