Research in Operating Systems *Sparrow*

Sparrow: Distributed, Low Latency Scheduling

K. Ousterhout, P. Wendell, M. Zaharia and I. Stoica. *In Proc. of SOSP 2013*

*Slides partially based on Ousternout's presentation

(Some) useful concepts picked up from OS

- Scheduler, preemption, shortest-job first, RPC, gang-scheduling, fair-share scheduling, …

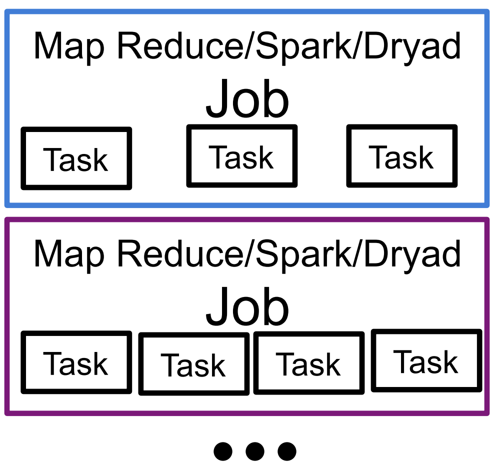
(In no particular order)

Jobs and scheduling for data analytics

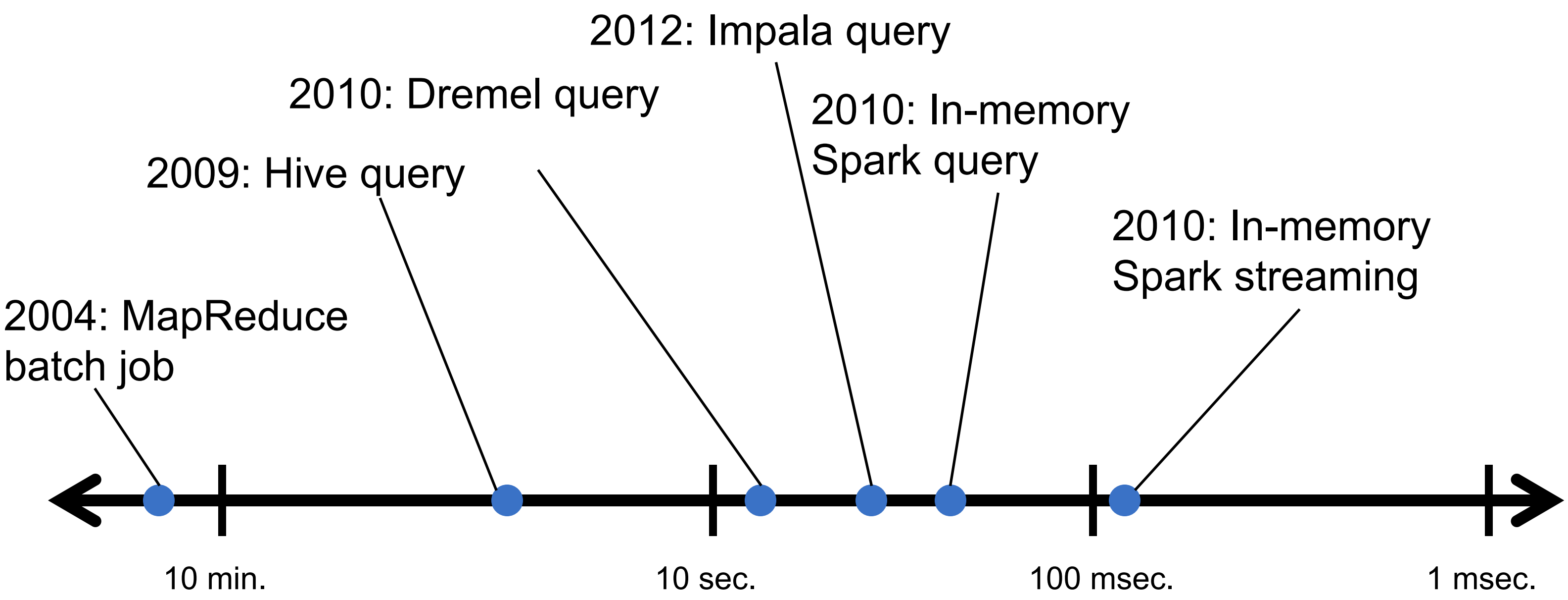
- Large data analytics clusters
- Running ever shorter and higher-fanout jobs
- What for? Finance, language translation, highly personalize search, …

Jobs and scheduling for data analytics

- Jobs composed of short tasks
- Produced from frameworks that stripe work across 1^3 machines (e.g., Dremel, Spark, …)
- Targeting task running in ~100ms



Job latencies decreasing rapidly



Scheduling challenges

- Millisecond latency
- Quality placement
- Fault tolerant
- High throughput
- *Fixing centralized schedulers is not an option*

Sparrow schedules tasks in clusters

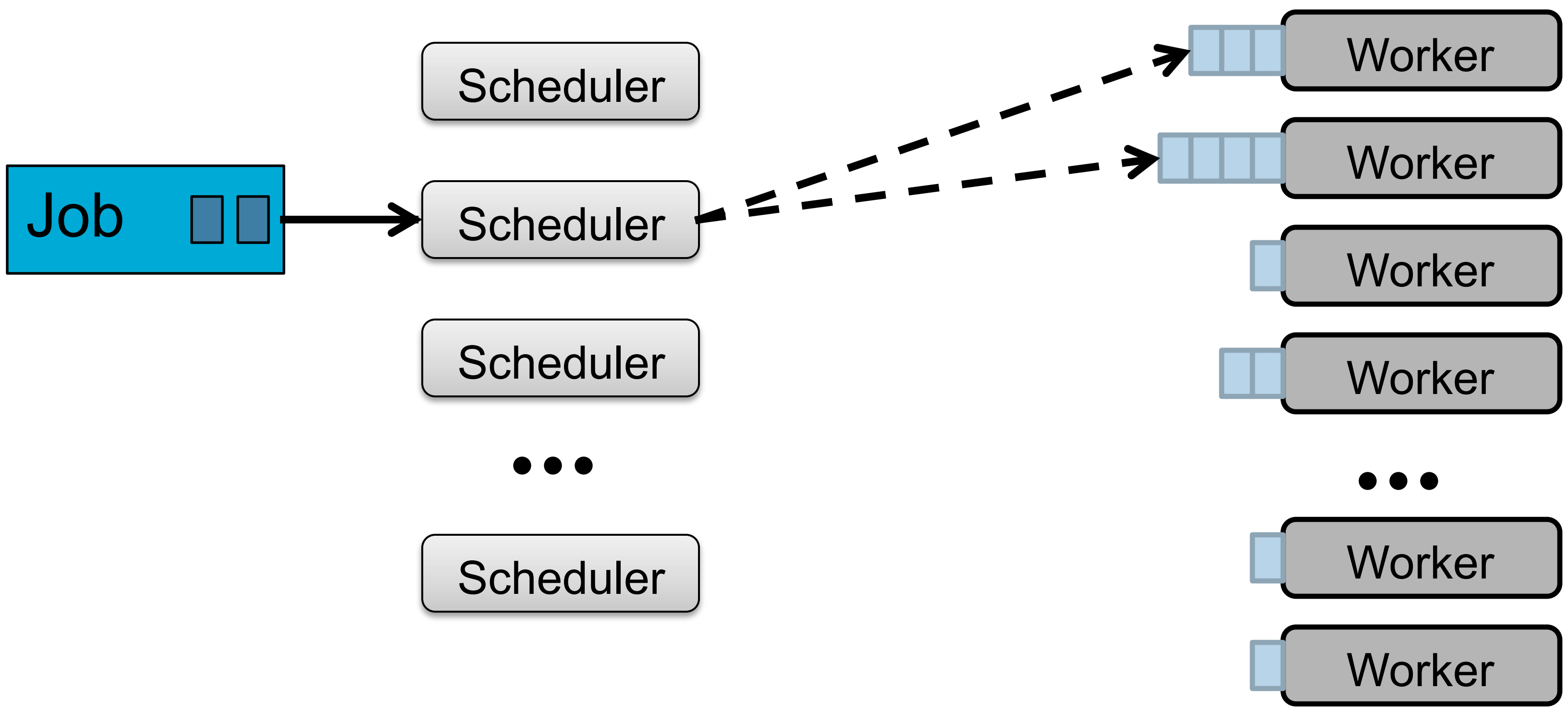
- using a decentralized, randomized approach
- support constraints and fair sharing and
- provides response times within 12% of ideal

- Adapts power-of-two-choices to parallel task scheduling, introducing three techniques
  – Batch sampling
  – Late binding
  – Policies and constrains

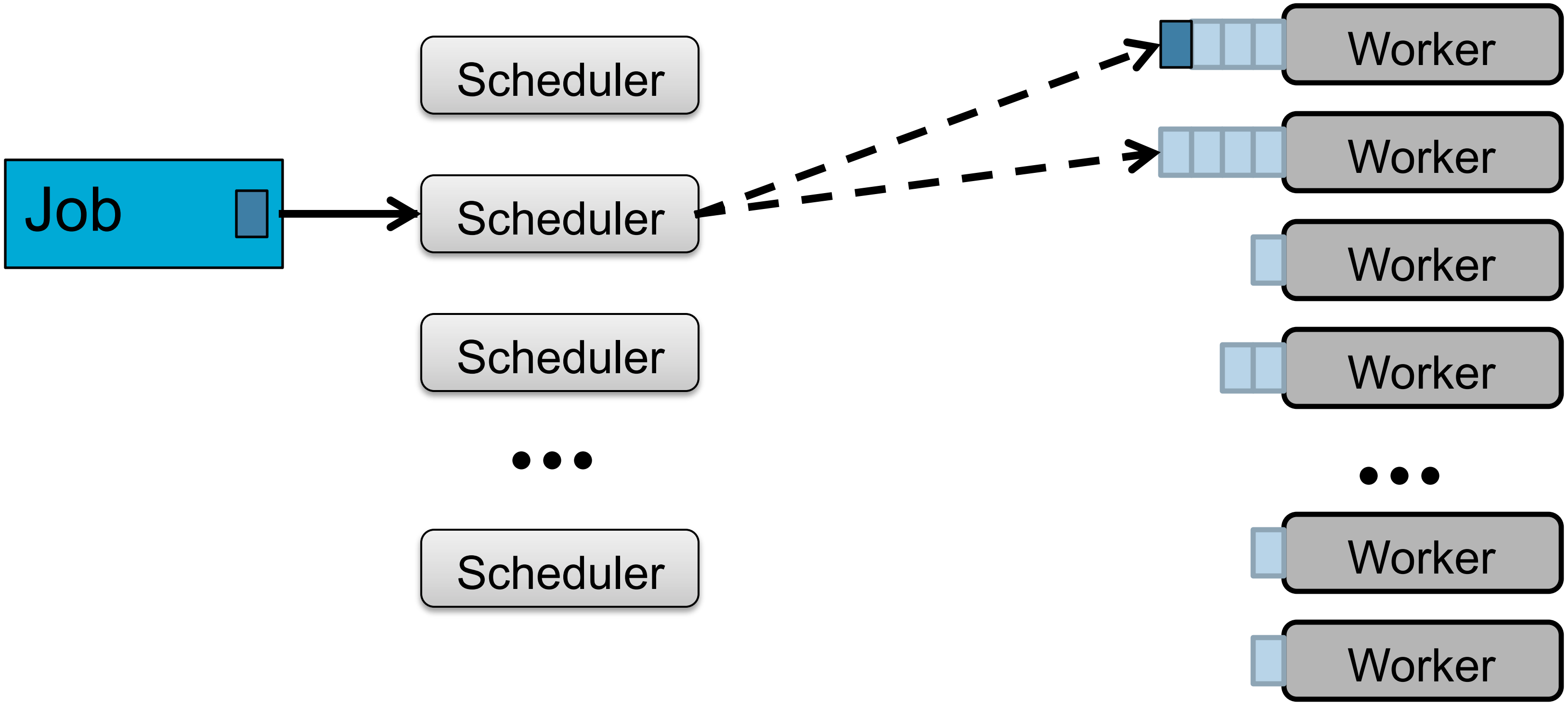
Per-task sampling

- A direct application of power of two choices
- For each task in a job, the scheduler
  – Randomly selects two workers
  – Probes each worker (a lightweight RPC) for queue length
  – Places task in shortest queue

Per-task Sampling



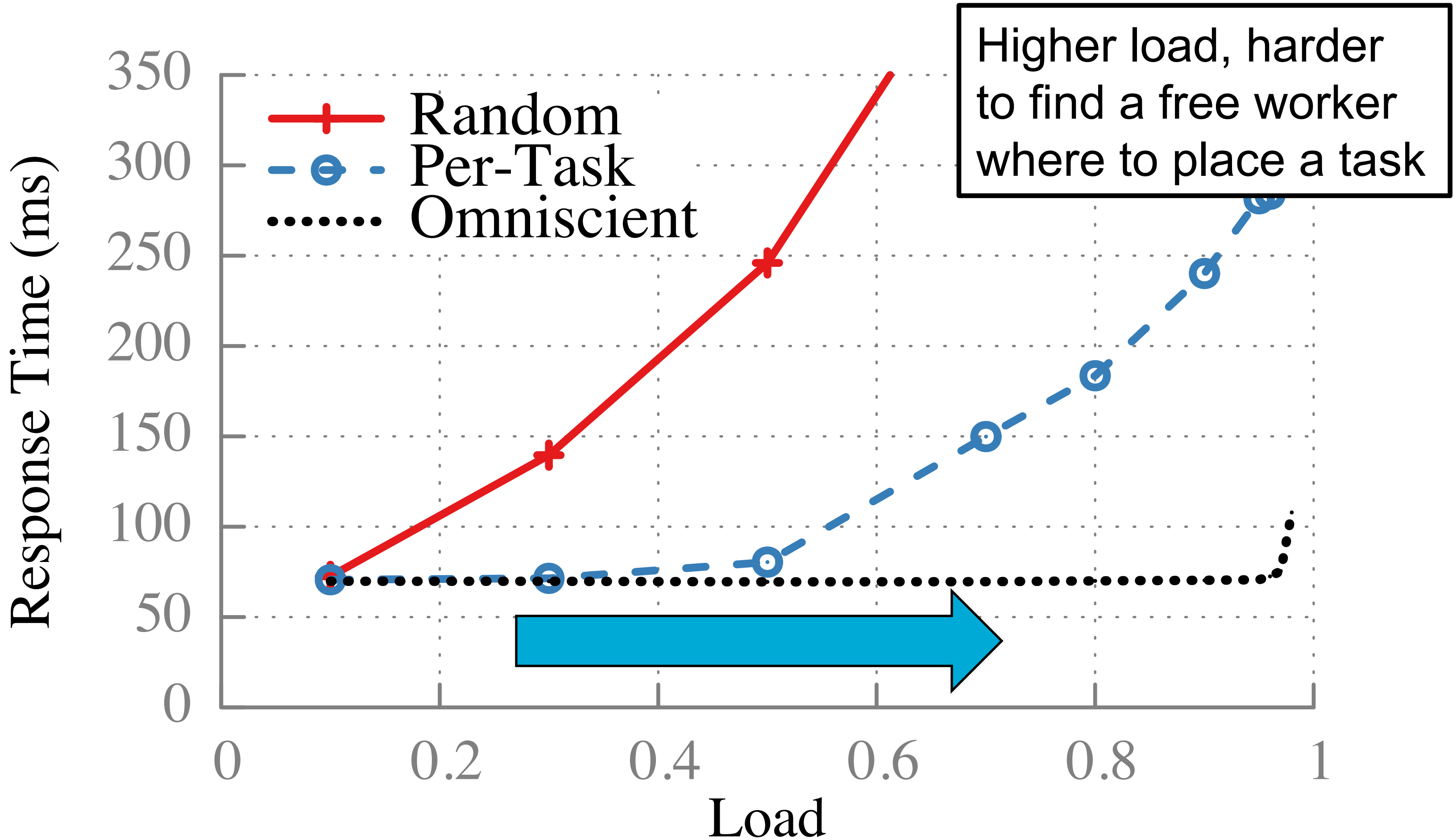
Per-task sampling



Per-task sampling

- A direct application of power of two choices
- For each task in a job, the scheduler
  - Randomly selects two workers
  - Probes each worker (a lightweight RPC) for queue length
  - Places task in shortest queue
- For comparison
  - Random
  - Omniscient – greedy, based on complete information

Better than random, >2x worst than opt

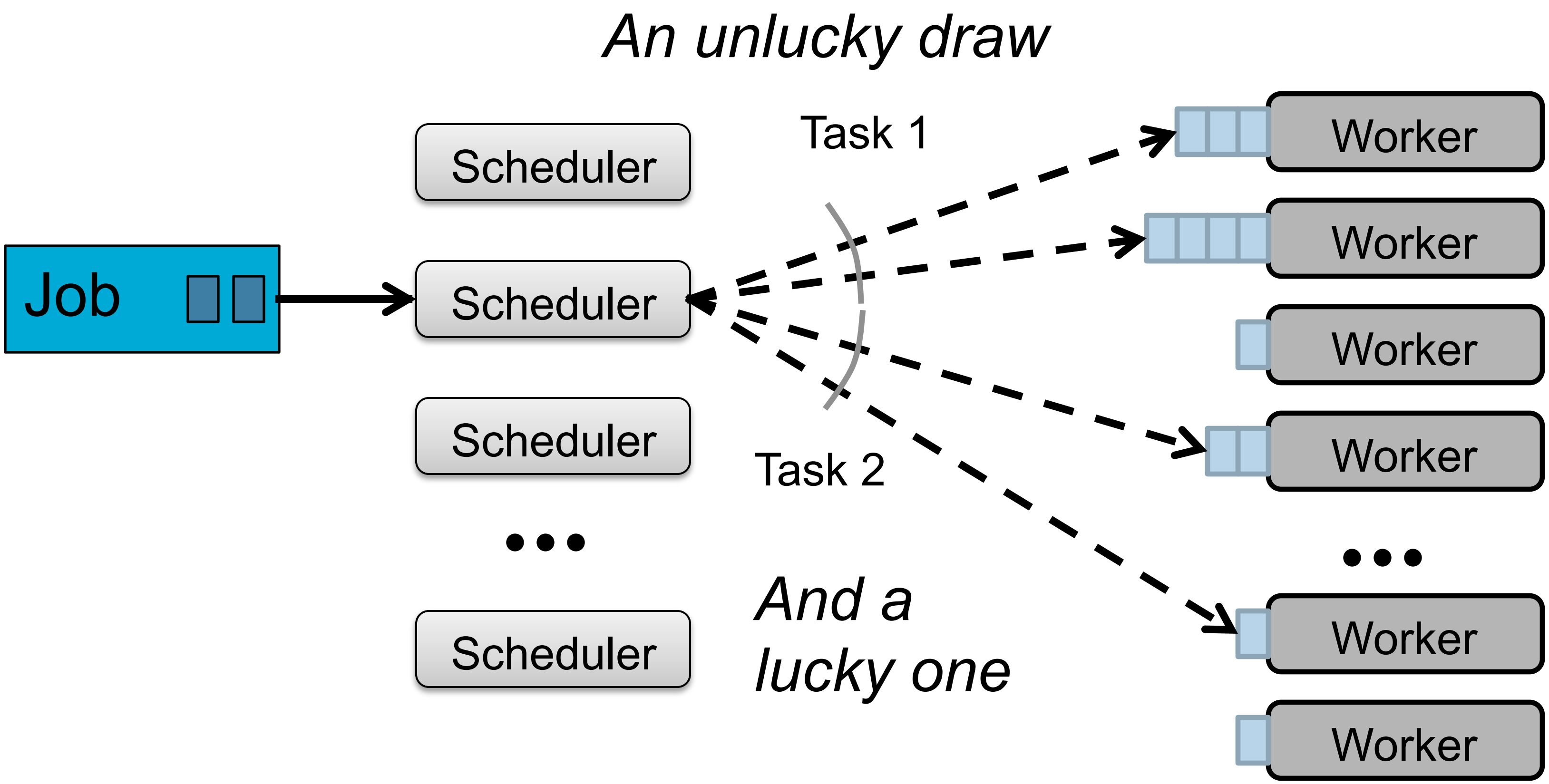


100-task jobs in 10,000-node cluster, exp. task duration

Response time grows with tasks/jobs

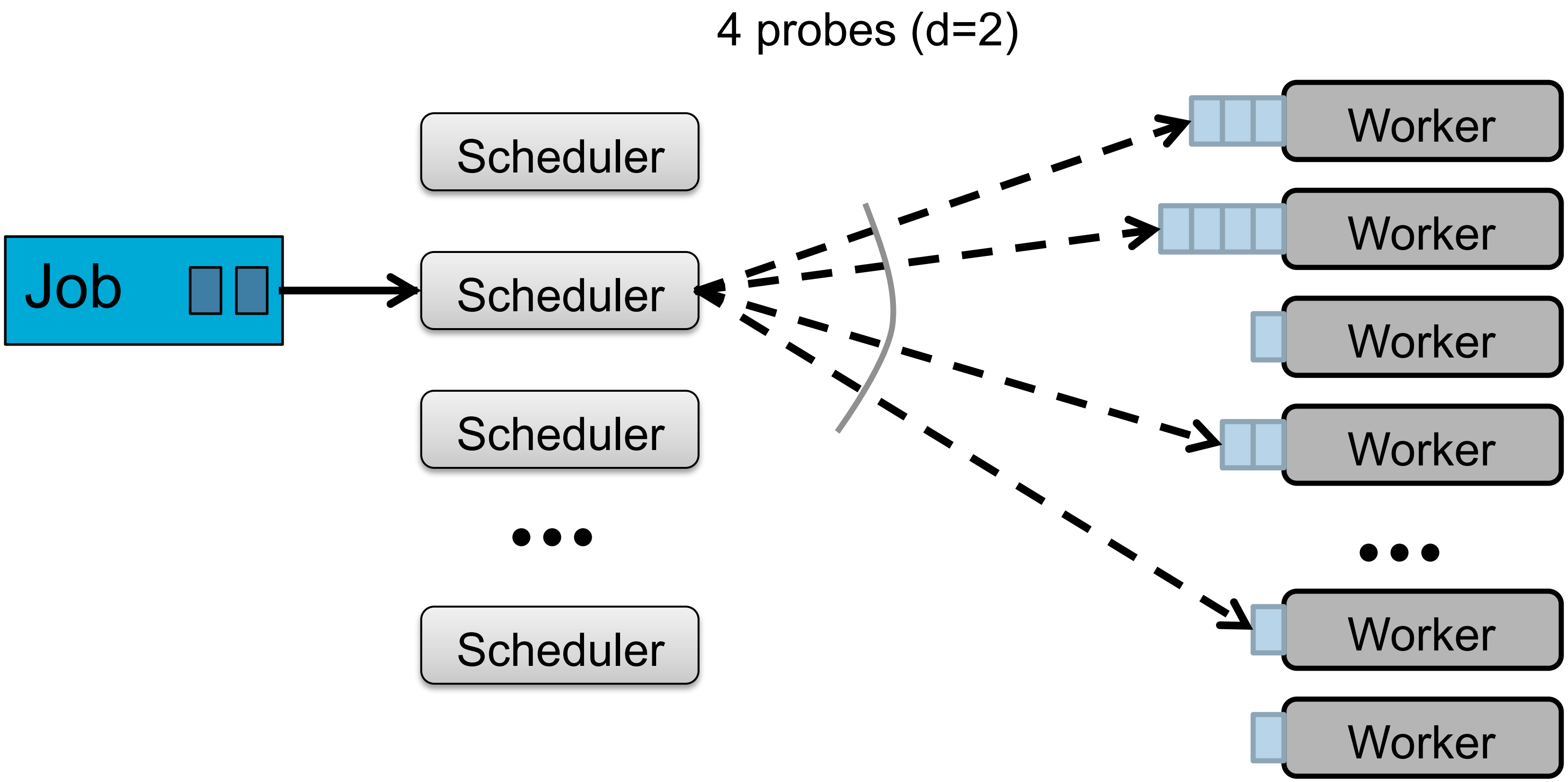
70% cluster load

Per-task sampling



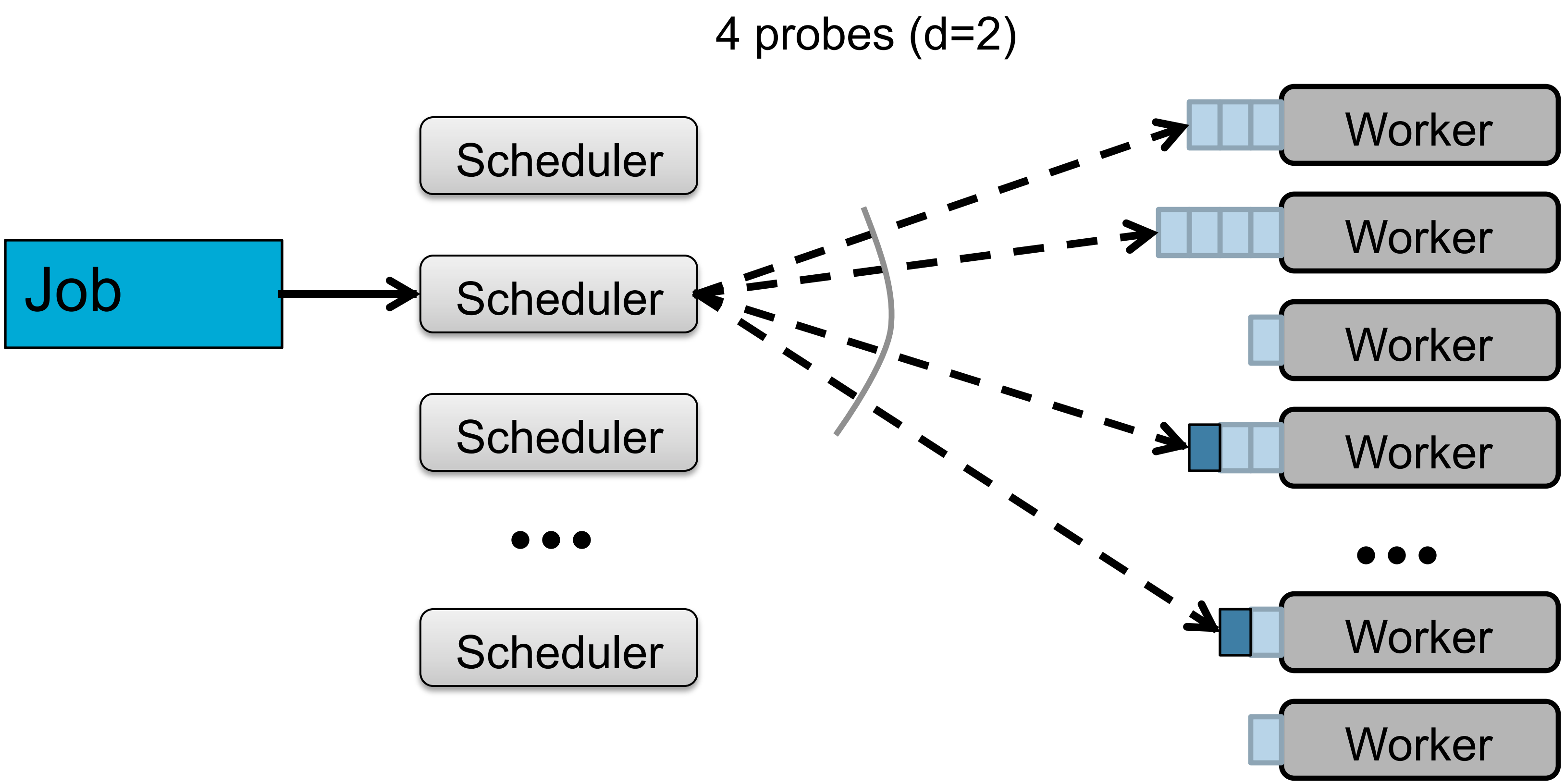
Job is done when all tasks are done ...

Batch sampling

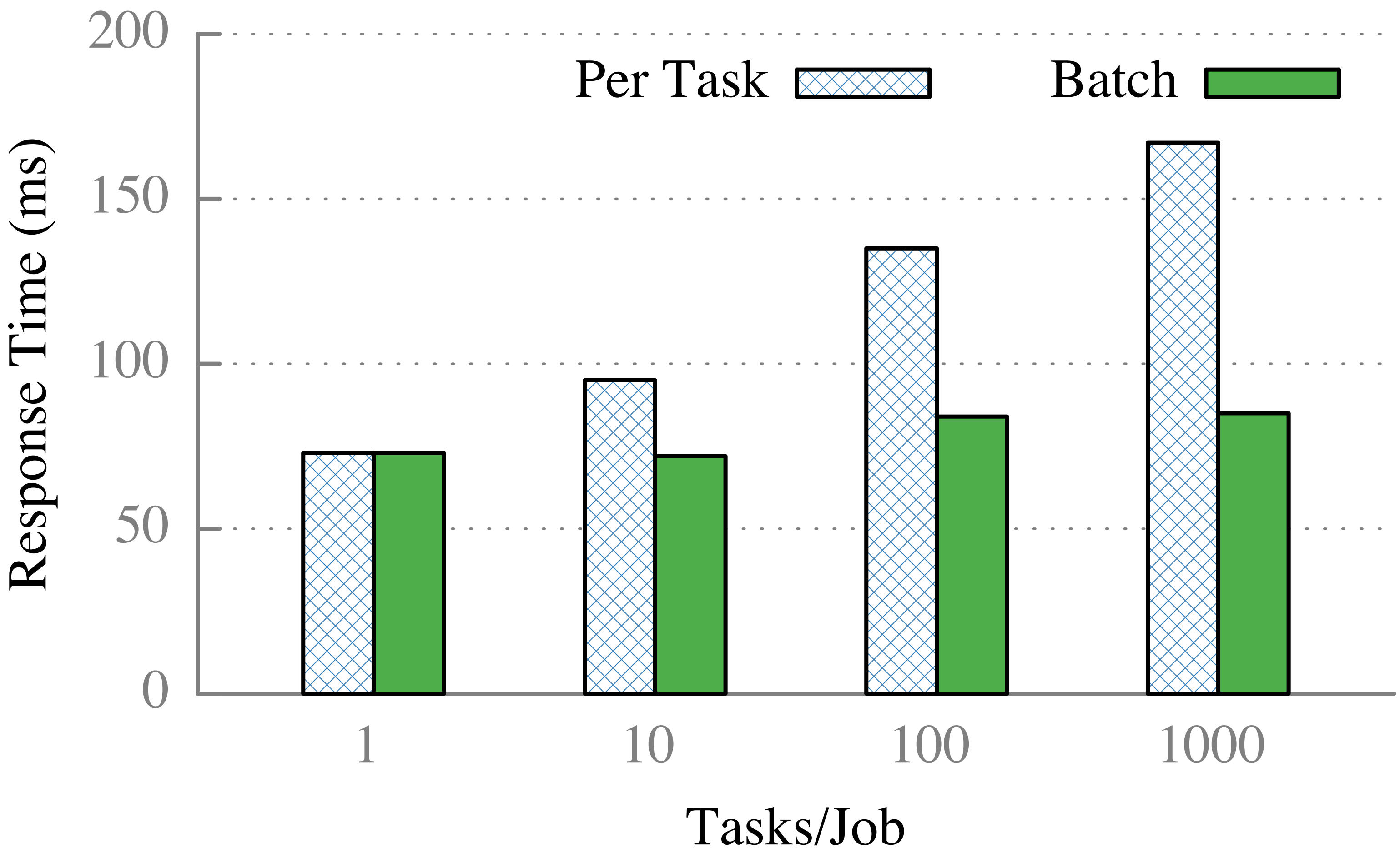


Place *m* tasks on the least loaded *d*m* workers

Batch sampling

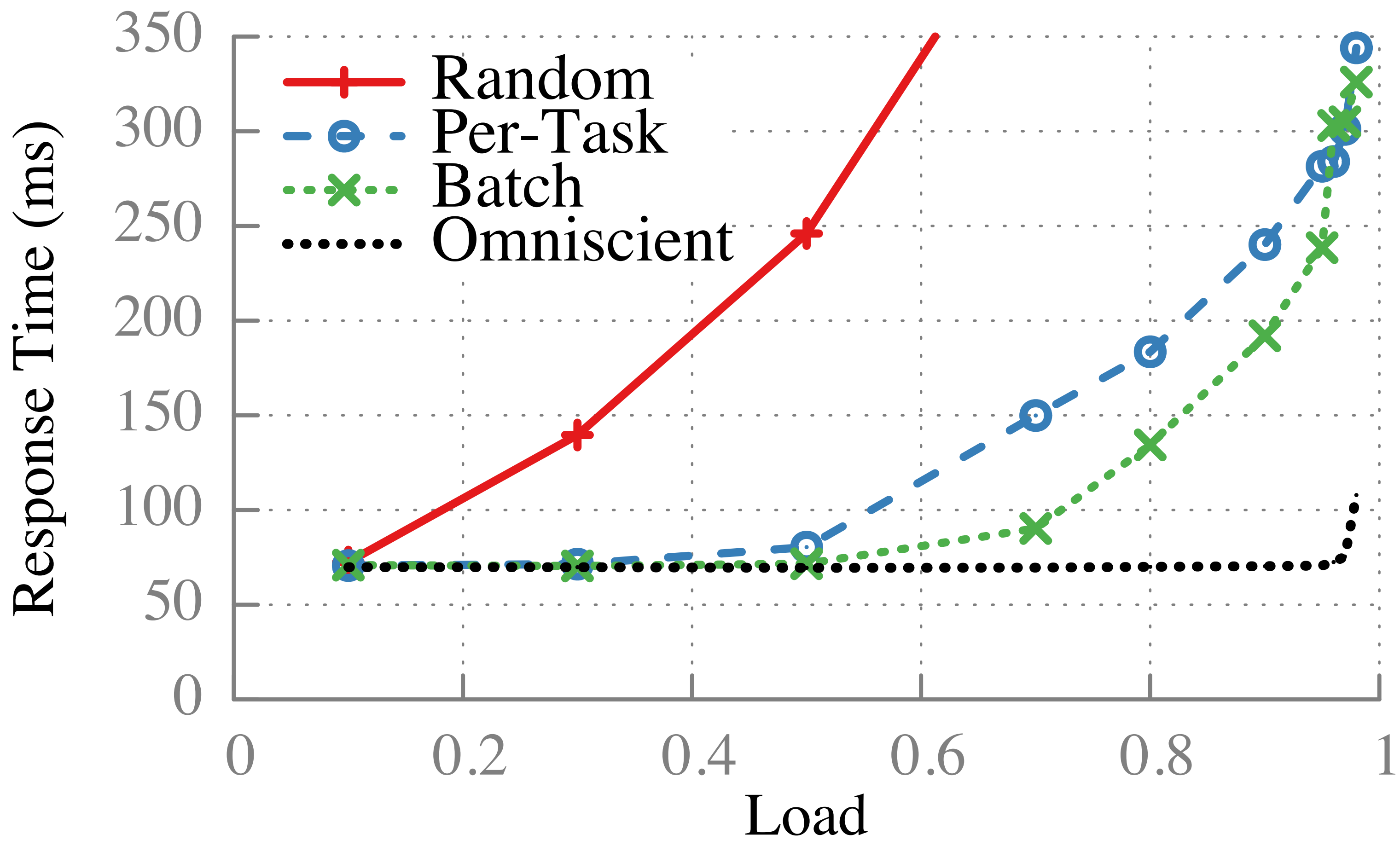


Per-task and Batch



70% cluster load

Batch sampling – better, still 1.92x opt



Late binding

- Sample-based scheduling performs poorly under load
  - Select based on queue length, coarse predictor of wait time
    - Better predictors using estimates of task durations is hard
  - Race condition – multiple schedulers picking the same worker

Late binding

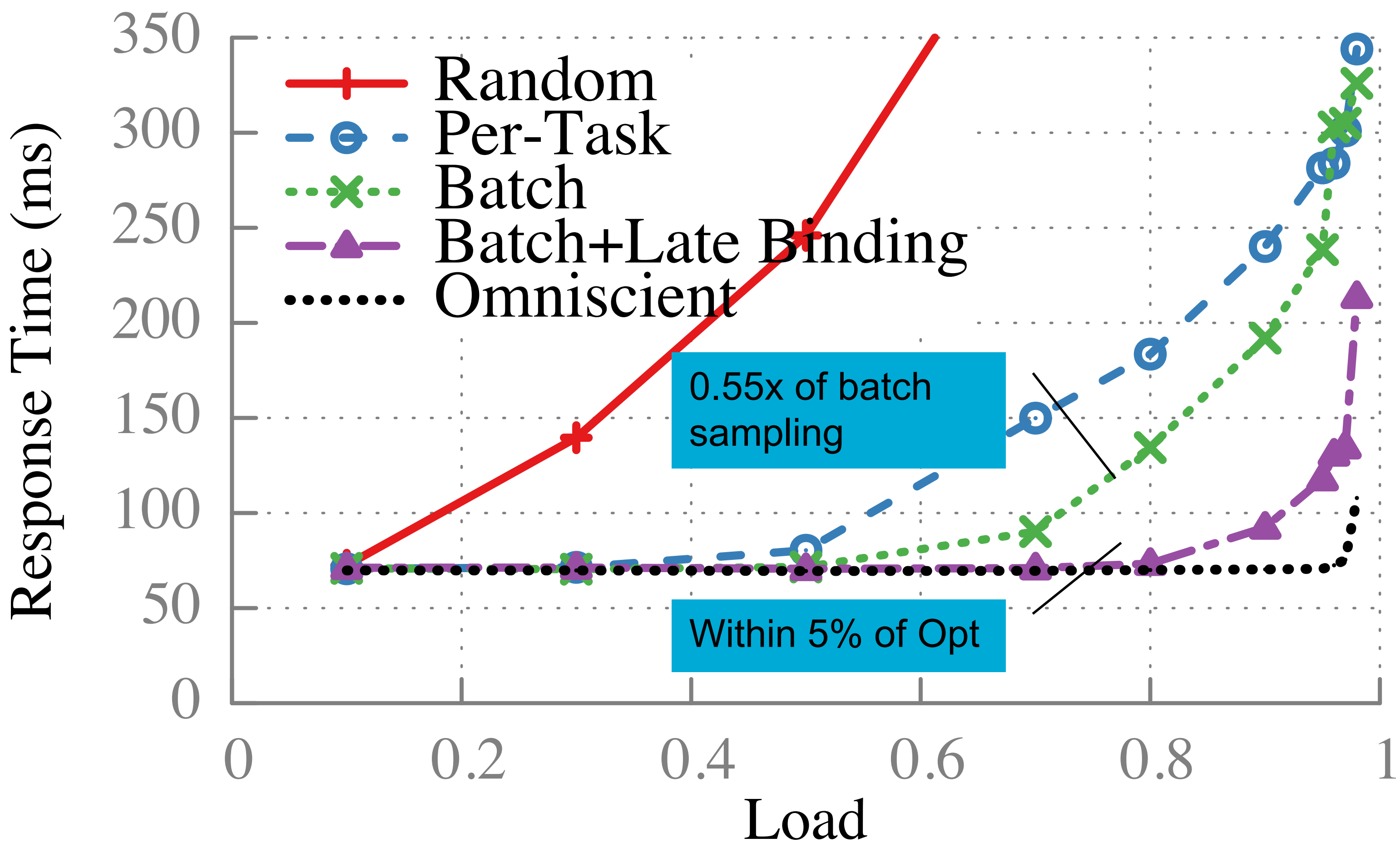
- Late binding
  - Workers put task internal work queue, hold back response
  - When task gets to the front, worker replies RPC
  - Scheduler assigns job's tasks to first *m* workers
  - … and sends no-ops to the rest (proactive)
- Cost
  - Idle while sending RPC – a 2% efficiency loss
  - Fraction of time idle while requesting tasks

$$(d * \mathrm{RTT}) / (t + d * \mathrm{RTT})$$

Mean task service time

Mean network round trip time

Late binding benefits



Handling constrains and policies

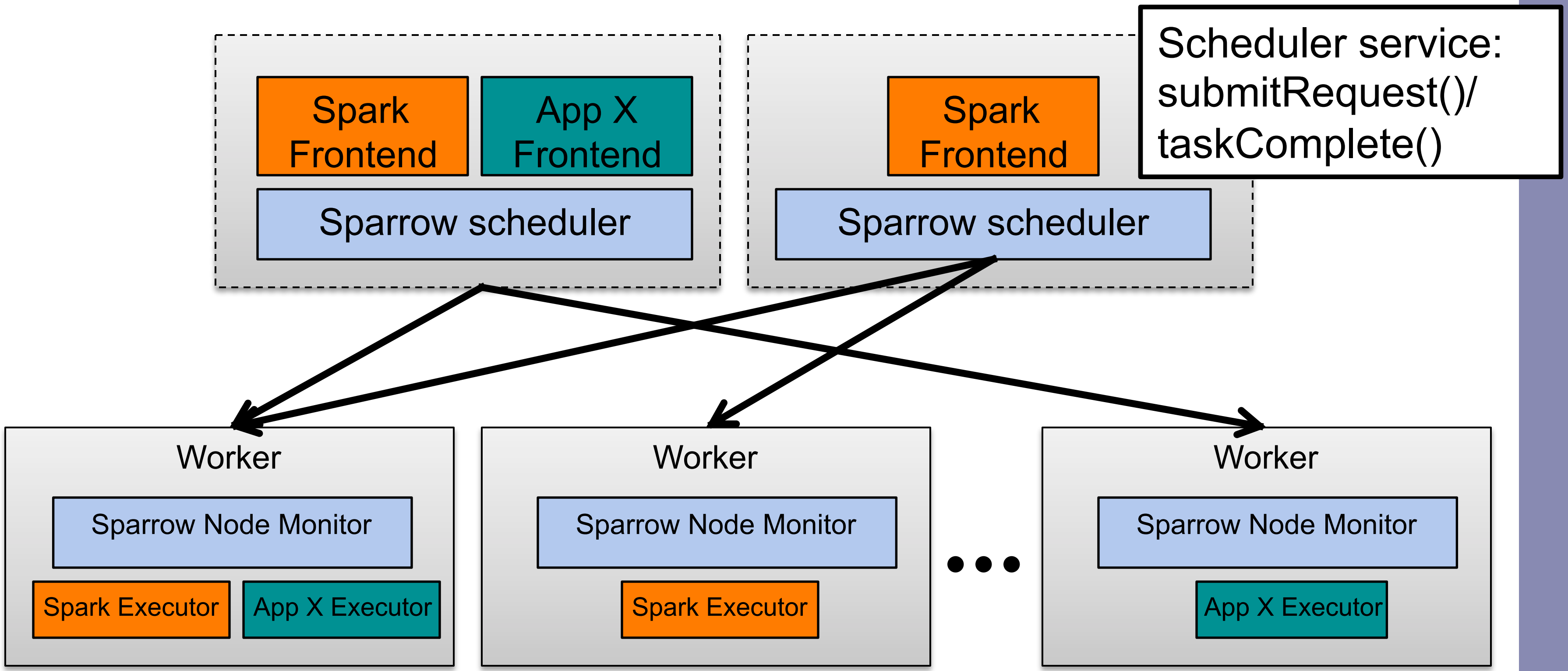
- Job constraints, e.g., *run in workers with GPU*
  - Trivially handled, pick *dm* candidate from the subset
- Per-task constraints, e.g., *run where input is*
  - Use per-task sampling, improved with
  - … sharing information across tasks when possible
  - … use late-binding

Handling constrains and policies

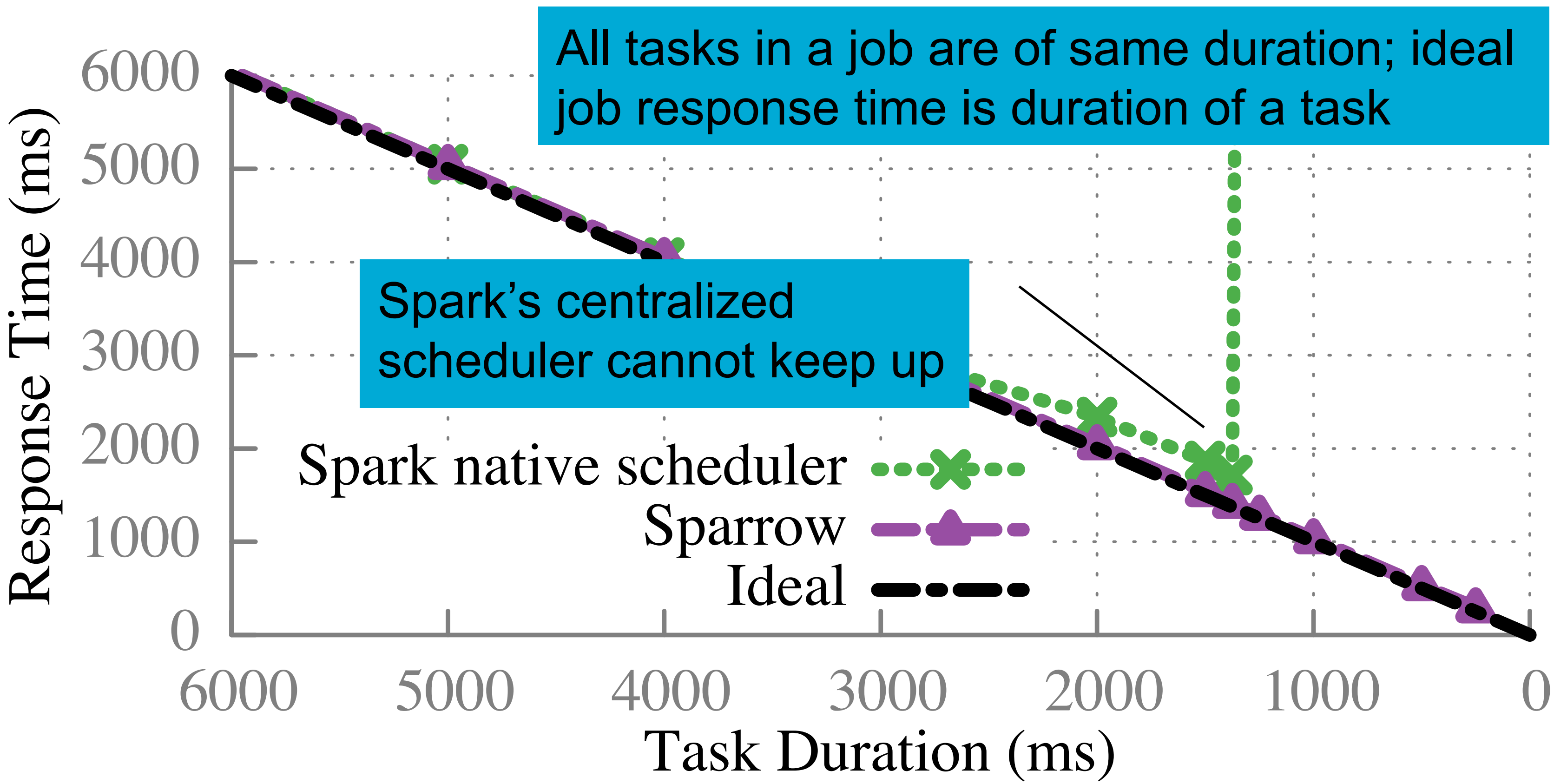
- Resource allocation policies, under load
  - Strict priorities through multiple queues on workers
    - FIFO, earliest deadline first, shortest job first, class, …
    - Trade accuracy for simplicity (no global information but low priority jobs can run first)
    - No preemption
  - Weighted fair sharing
    - One queue per user

Sparrow implementation

- Sparrow works for 1+ concurrent frameworks
  - Front end + executor (long-lived processes responsible for executing task w/o startup overhead)
  - Node monitor federates resources usage bet/ co-located frameworks



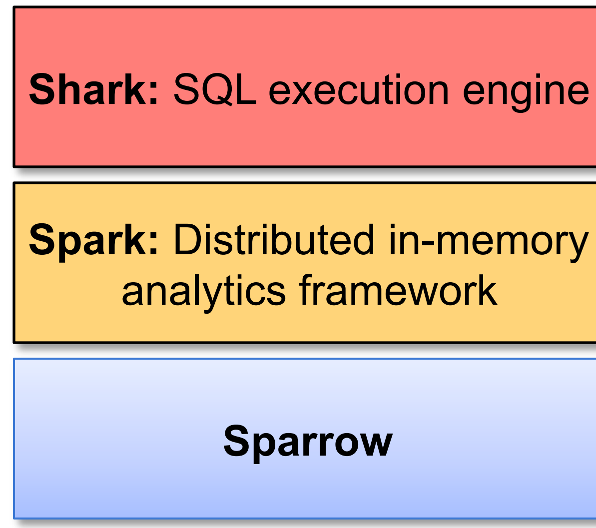
Sparrow and Spark's native scheduler



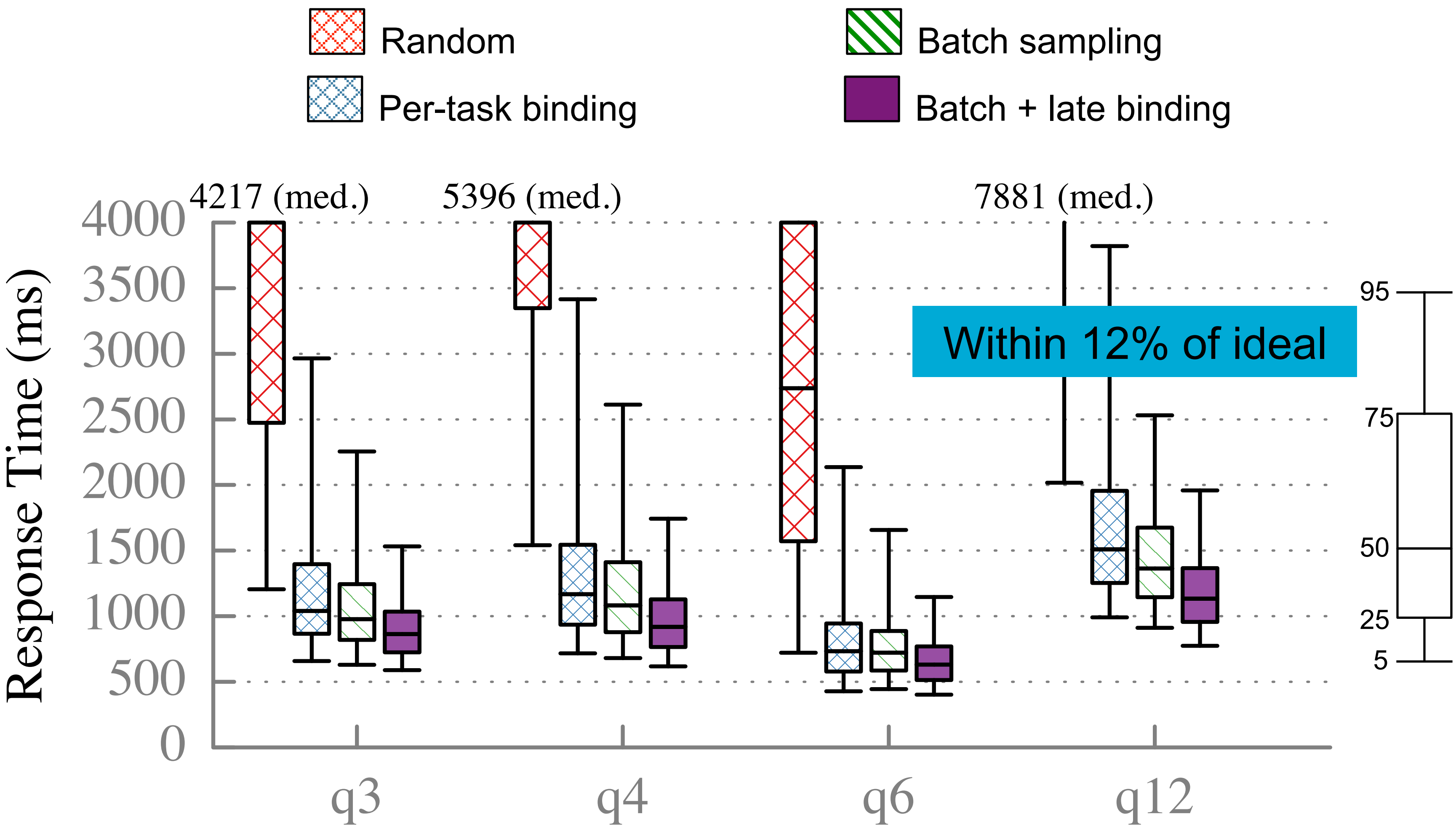
Small 100 16-core EC2, 10 tasks/job,
10 schedulers, 80% load
(synthetic workload)

TPC-H queries: background

- A common benchmark for analytics workloads; representative of ad-hoc queries on business data
- Shark queries compiled into multiple spark stages
- Each stage triggers a scheduling request using submitRequest()
- Task in first stage constrained to machines holding input data
- Stages have different number of tasks, durations and un/constrained queries



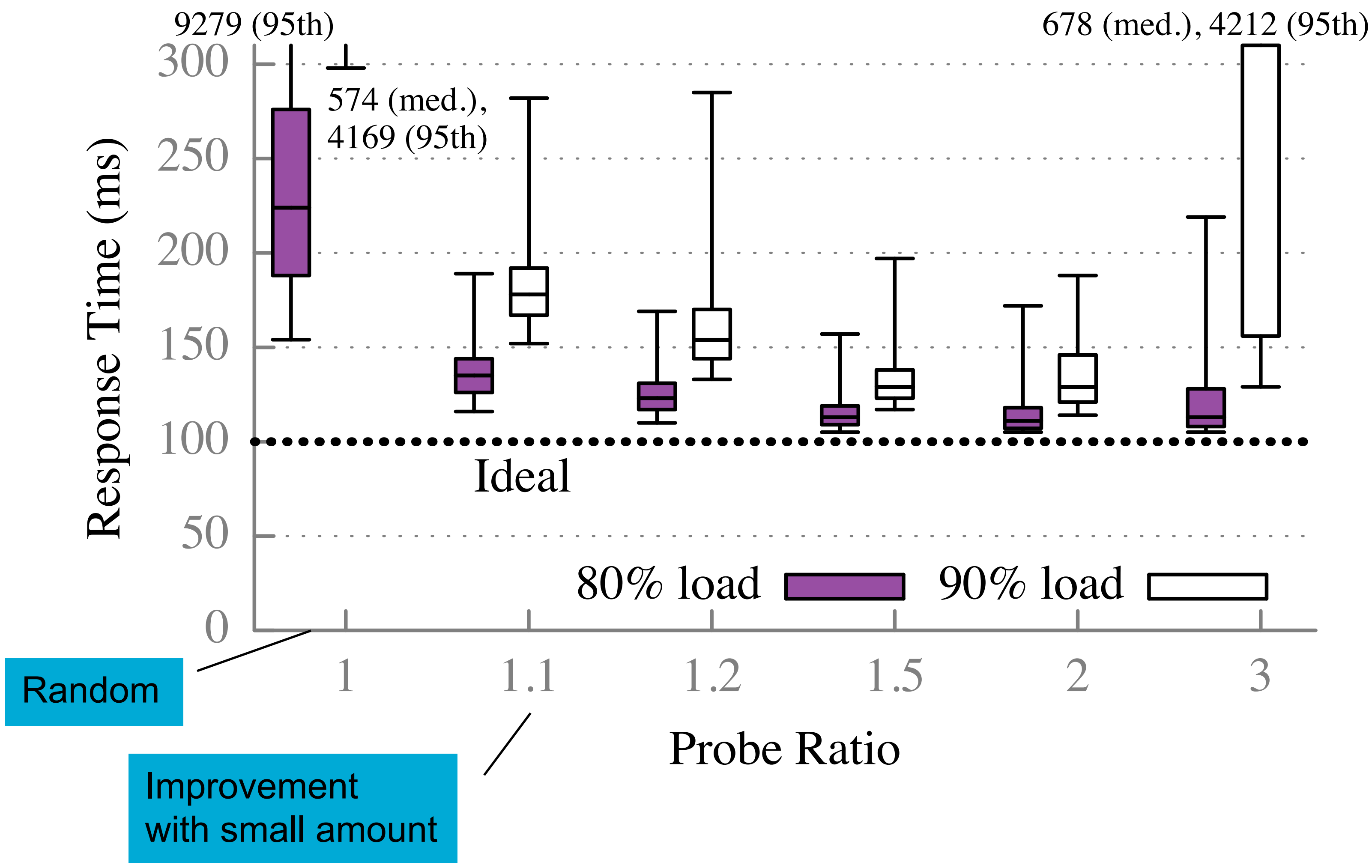
TPC-H Queries



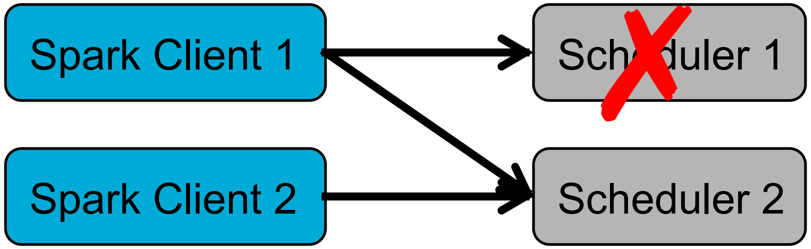
100 16-core EC2, 10 tasks/job, 10 schedulers, 80% load

Effect of probe ratio

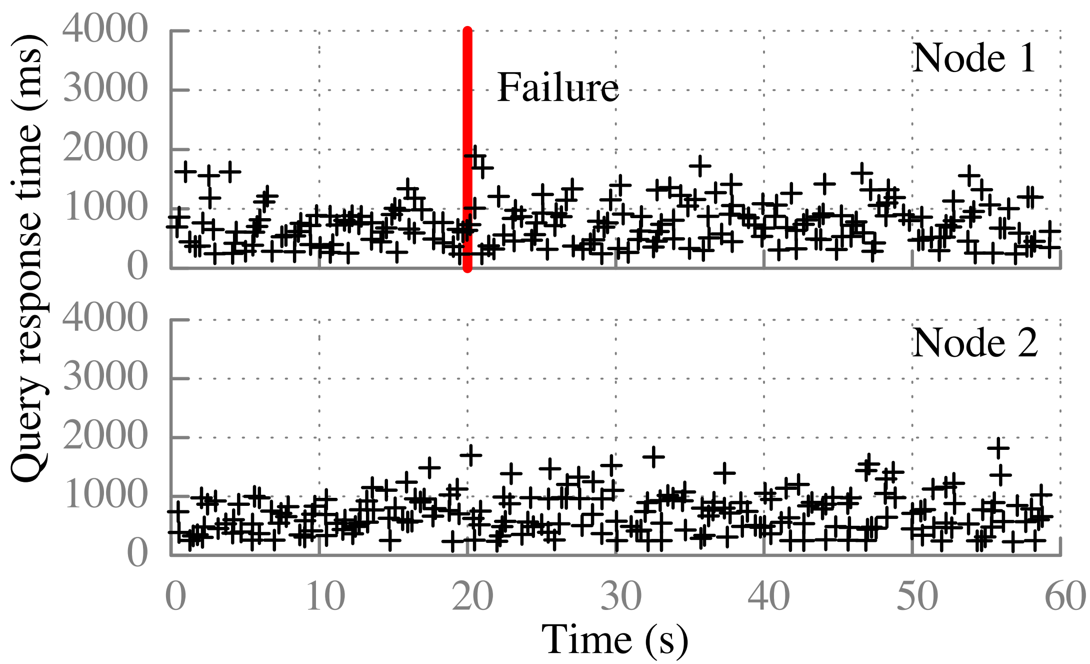
Too low, can't find lightly loaded machines
Too high, pay the cost of increased messaging



Failure impact



Detect failure 100ms
Failover 5ms
Re-launch queries 15ms



Limitations/future work

- Scheduling policies – can they do better than approximate?
- Inter-job constraints (e.g., *tasks of job A cannot run with those of B*) – hard to do w/o drastic changes
- Gang scheduling – no central point where to do it
- Query-level policies – easy to extend, FIFO
- …
- *Want to try?* http://github.com/radlab/sparrow