Cloud-Based Data Processing

Cluster-level Scheduling and Resource Management

Jana Giceva

Performance

- Performance is measured in terms of **throughput, response time**, and **availability.**

- In the cloud, performance targets are tuned for the requirements of each workload:
  - **Latency** or response time of specific requests
  - **Throughput**: the number of requests performed per second

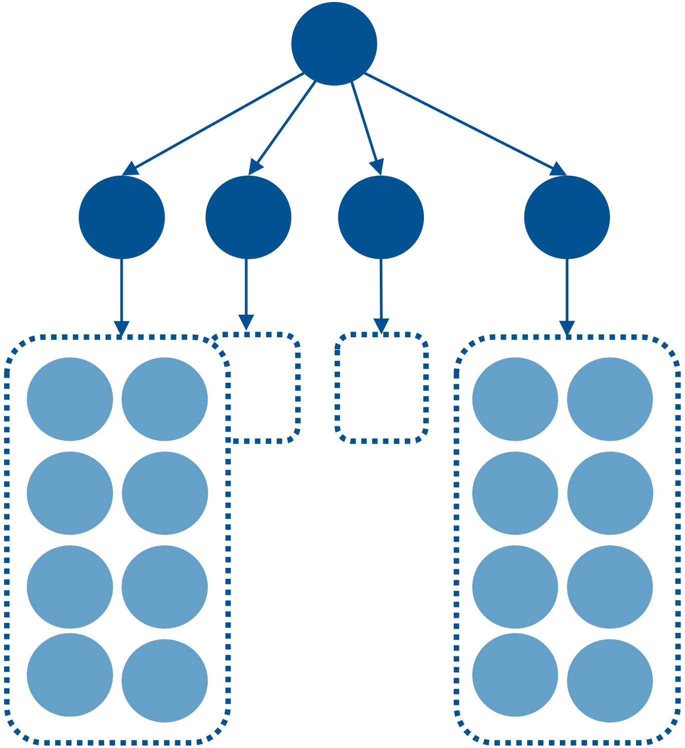
- Example SLO
  “Client requests will have a response within 500ms at P90, at loads up to 25 K requests / second”

Challenges in a distributed system

- A common technique to reduce latency at scale is to parallelize across many machines.

- A single transaction involves **multiple components** of a system.
  - Big **fan-out** and latencies add up
    - **Component-level variability amplified by scale**
  - Hard to get a holistic end-to-end view of a single operation

- Resource consumption for each operation is distributed across multiple nodes.



Tail at scale

- Recall the tail-at-scale?
  - 99th percentile for **a random request** to finish is 10ms
  - 99th percentile for **all requests** to finish is 140ms.

- Waiting for the slowest 5% of the request to finish is responsible for half of the total 99th percentile latency.

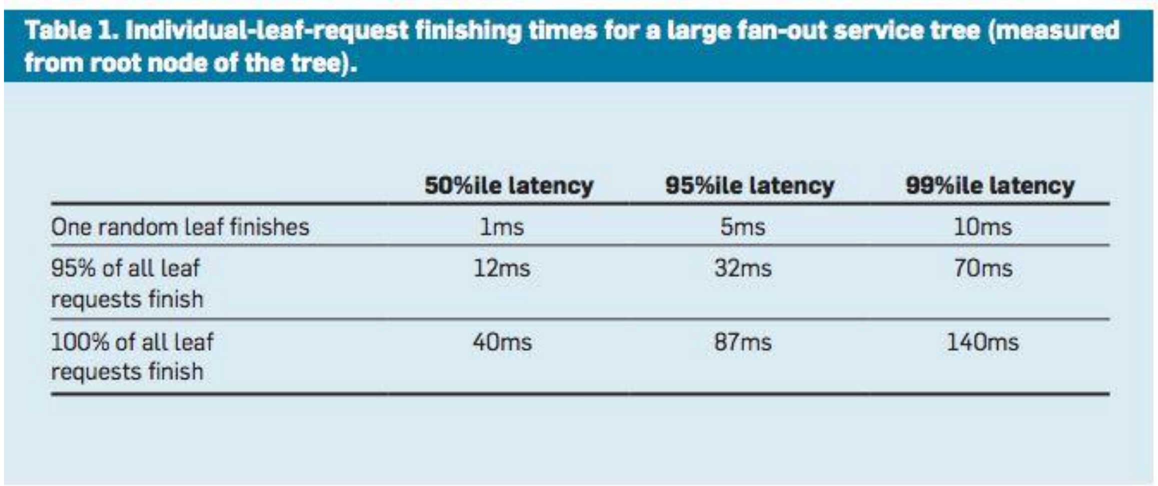


Table 1. Individual-leaf-request finishing times for a large fan-out service tree (measured from root node of the tree).

	50%ile latency	95%ile latency	99%ile latency
One random leaf finishes	1ms	5ms	10ms
95% of all leaf requests finish	12ms	32ms	70ms
100% of all leaf requests finish	40ms	87ms	140ms

Why does variability exist?

- Variability can arise from many reasons:
  - **Shared hardware resources:**

    that may lead to contention (CPU cores, caches, memory-, network bandwidth, etc.)
  - **Background tasks and daemons**:

    although using a limited set of resources on average, when scheduled can generate a short disruption
  - **Global resource sharing**:

    network switches or shared file systems can become a contented hot-spot
  - **Queuing**:

    multiple layers of queuing at intermediate servers and network switches amplify the variability
  - **Maintenance:**

    background activities (e.g., recovery, log compaction, garbage collection, etc.)
  - **etc.**

    including hardware trends like power limits, energy management, etc.

General best practices (client's perspective)

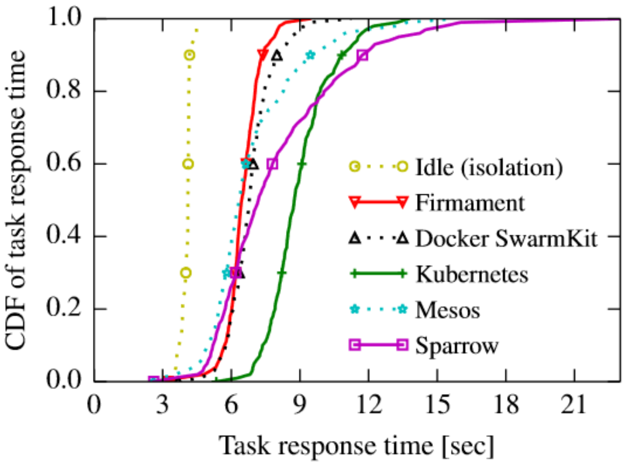
- Performance tuning by taking a **systematic approach**:
  - Enable **telemetry** to collect metrics and instrument your code.
    - Use correlated **tracing** so that you can view **all the steps in a transaction**.
  - **Monitor the 90/95/99th percentiles**, not just average. The average can mask outliers.
    - The **sampling rate** also **matters**. If it is too low, it can hide spikes or outliers that indicate problems.
  - Look for opportunities to **parallelize** (**replication** and **partitioning** are your friends).
  - Watch out for **skews** and **hot-spots –** balance the service distribution and load (e.g., **repartitioning**).
  - Issue the same request to multiple replicas (after a short delay) and use the result(s) that arrive first

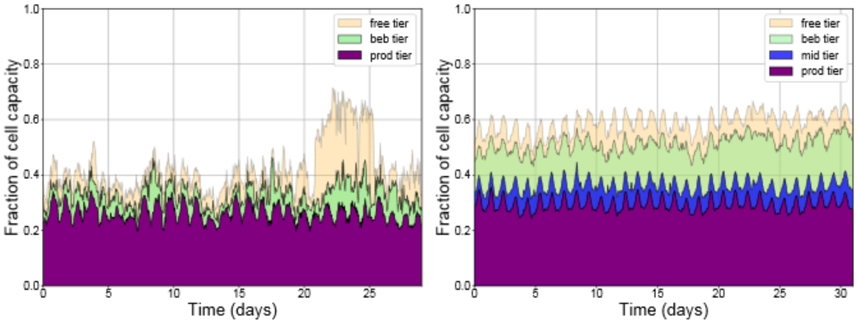
Optimization goals

- A **client** is interested in low **latency** for its own transaction(s) and **high availability**
  - Tail latency, or **percentiles 90/95/99th**

- An **application** is interested in good **tail latency** at a given **throughput** and **high availability**
  - Tail latency by itself is not as important, needs a high (guaranteed) throughput as well

- A **cloud vendor**:
  - Needs to meet the SLAs to the client applications
  - But also **maximize** the utilization of its fleet and **consolidate** many **workloads/applications.**





(a) 2011 CPU usage.

(b) 2019 CPU usage, averaged across all 8 cells.

Applications with different WL requirements

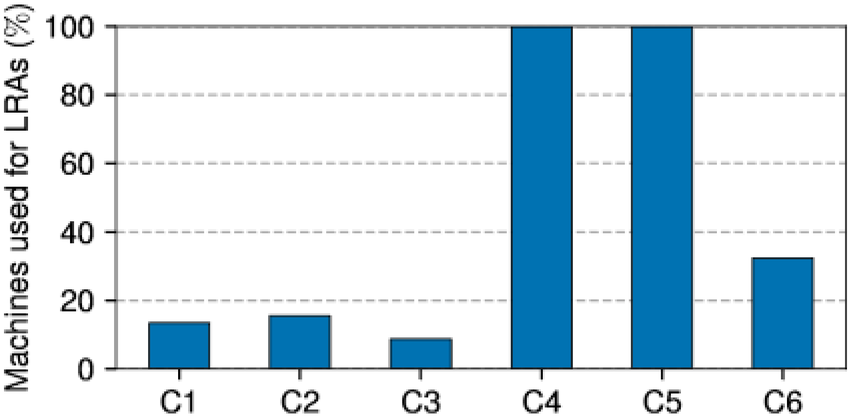
- **Batch –** Production workloads that execute tasks (in containers) that run for seconds to minutes.
  - e.g., MapReduce, Hadoop, Scope, Tex, etc.
  - Perform off-line computation, not sensitive to machine failures, no strict placement requirements except optional data locality to avoid data movement.

- **Long-running jobs**
  - **Streaming systems**: process data streams in near-real time via data-flows of long-running operators that are deployed using containers (e.g., Storm, Flink, Kafka etc.)
  - **Interactive data-intensive applications**: employ long-standing workers (executors), to avoid container start-up costs and to process data that resides in memory with low latency (Spark, Impala, etc.)
  - **Latency-sensitive applications:** serve requests using long-standing containers to achieve low end-to-end latency (e.g., Hbase, ZooKeeper, Memcached, etc.)
  - **ML frameworks** use long-running executors to efficiently perform iterative computations (e.g., TensorFlow, Spark MLLibs, etc.)

Workload diversity

- Publicly available cluster **traces** confirm the diversity of workloads in terms of job runtimes and resource demand.

  – *Google*: 80% of the jobs use containers in the cluster that have durations of less than 12 min, while the longest last more than a month.



  – *Alibaba*: a significant imbalance in terms of container resource usage between long-running on-line services and off-line batch analytics.

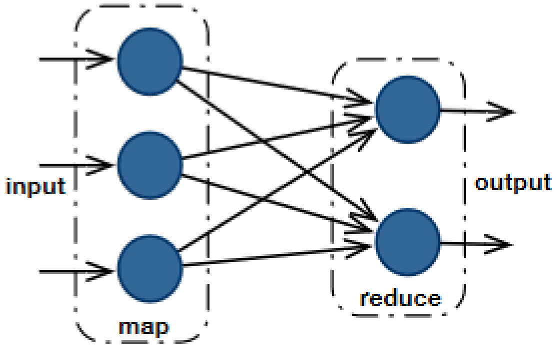
  – *Microsoft*: at least 10% of each cluster’s machines are used for long-running applications (12+ hours), while in two clusters the machines are used exclusively by them.

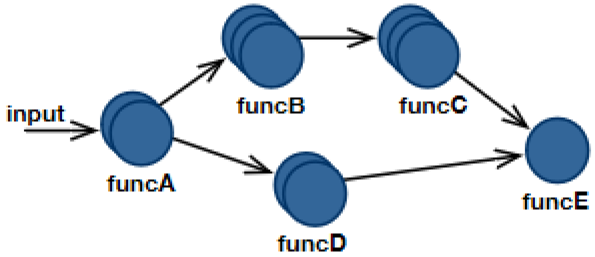
Workload scheduling within a framework

Representing application’s workload

Workloads in data processing frameworks are represented as generic dataflow graphs

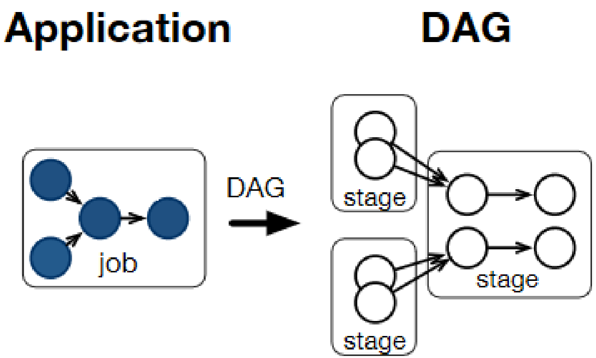
- **Nodes** perform **computation**
- **Edges** represent **data flow** across the nodes.





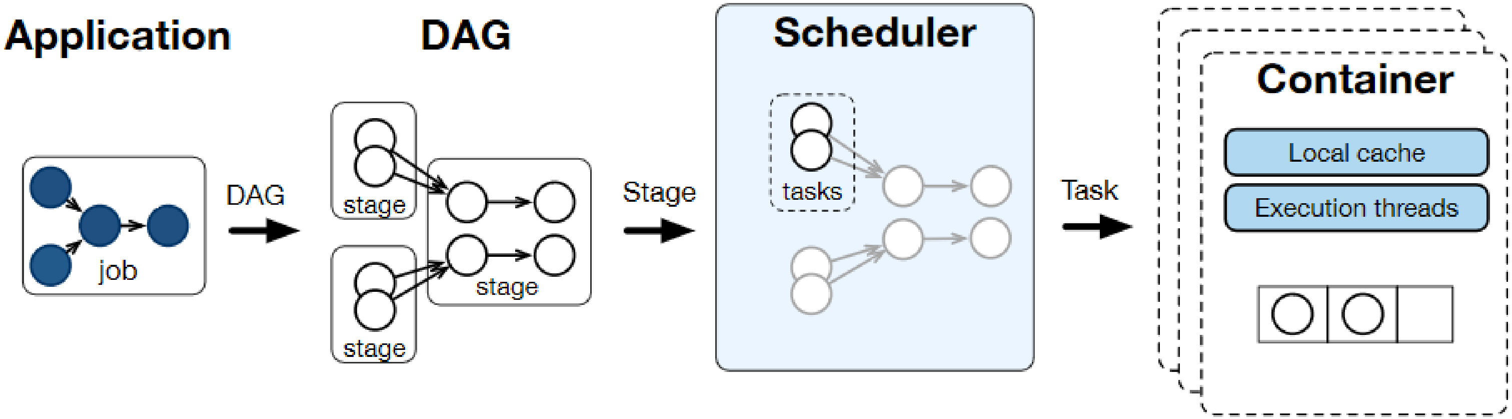
- Dataflow computation is performed in parallel tasks that usually run as part of **long running containers** in the cluster.

Execution steps in a dataflow engine I



- An **application** consists of **several jobs**. Each **job** is represented as a **DAG of operators**.
  - Operators in the DAG are grouped into **stages** (also called *operator chaining* or *pipelining*).
  - Each stage is a collection of tasks, where each **task** performs **computation** over a different **partition of data.**
  - At each stage boundary, data is written to a local cache (memory, or disk) and then transferred over the network to tasks in the downstream stages.

Execution steps in a dataflow engine II

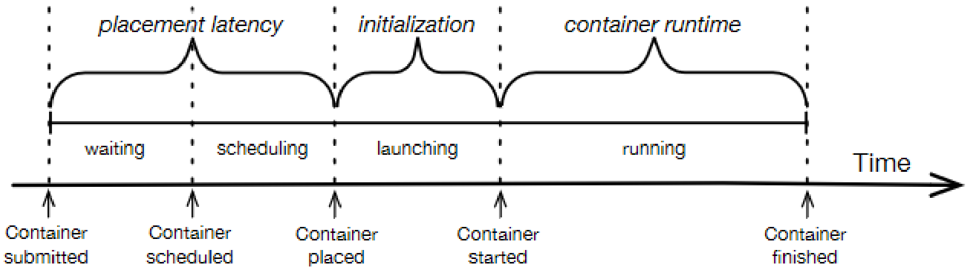


- A **scheduler** is responsible for assigning tasks to containers in the cluster.
- For execution, a **container** receives a computation **task** and a **reference** to its **input data**.

- The scheduler's decisions are **data-driven:**
  - Which operator in the DAG graph is ready for execution (has satisfied dependencies).
  - Spawn parallel tasks on particular machines in the cluster that have the binary by respecting the resource needs and data locality.

Execution model

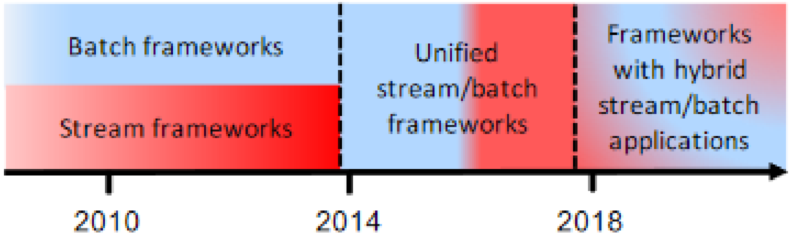
- Depending on the **execution model** of the dataflow system, tasks are executed on a container in
  - **continuous operator model –** low RT by immediately processing a stream of data
  - **bulk-synchronous model –** higher throughput with optimized execution within each batch of data.



  - **executor model**
    - Tasks are dispatched to executors (long-running containers).
    - Executors are deployed on machines in the cluster (worker nodes) and typically run for the entire lifetime of an application (avoiding repeated container scheduling latencies and initialization costs).

Evolution of the execution model

- **Early dataflow systems followed an execution model targeting a particular use-case.**
  - Batch vs. stream processing
  - (e.g., Hadoop, Spark, Presto, Storm, Flink)

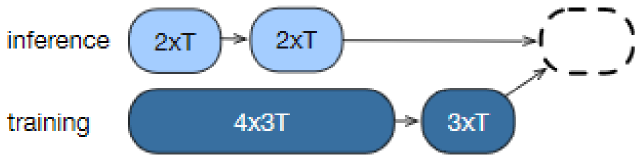

Batch frameworks
Stream frameworks
Unified stream/batch frameworks
Frameworks with hybrid stream/batch applications
2010
2014
2018

- **Modern dataflow platforms evolved to support both stream and batch applications**
  (e.g., Spark's structured streaming, Flink's Table API or Apache Beam, etc.)

- **Today hybrid applications have both latency-sensitive stream jobs with latency-tolerant batch jobs.**
  - e.g., real-time service to detect malicious behavior
    - with a batch-based ML-training model job going over historical data
    - stream job doing inference over the trained model to detect malicious behavior on real-time data.

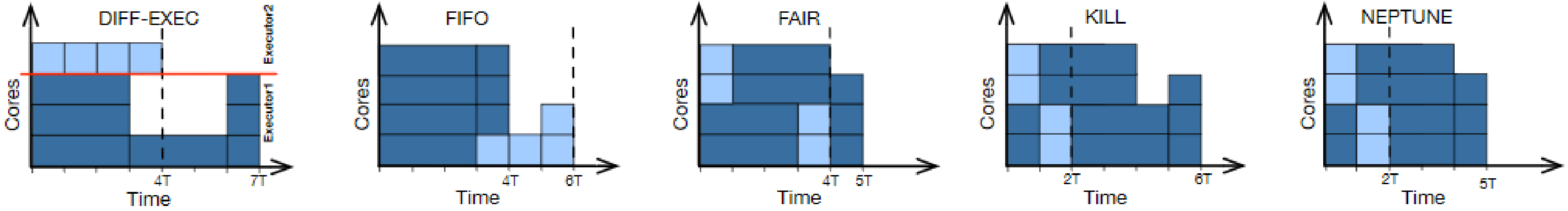
Scheduling hybrid dataflow applications

- Problem when scheduling a hybrid stream/batch application.
  - Real-time inference job
  - Historical data training job

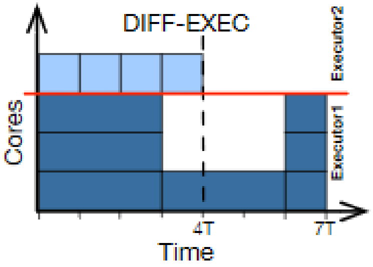


  - The training tasks require up to 3x more time and 2x more resources than the latency-sensitive tasks.

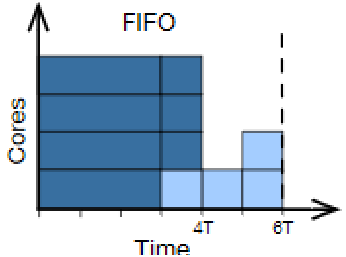
- The scheduling **policy** can significantly affect total **job execution time = throughput**, efficient **usage of resources**, and meet the tasks’ **latency requirements**.



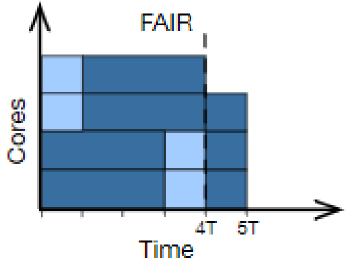
Scheduling hybrid dataflow applications



- **Basic approach:** each jobs uses a dedicated set of resources
  - Separate engines for stream and batch processing
  - Used in existing production environments with *general purpose resource managers* that have *dedicated queues* for latency-critical jobs.

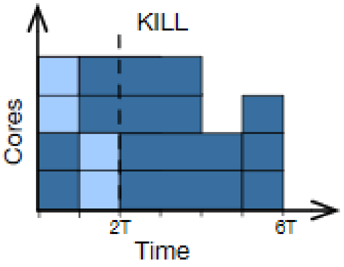


- **FIFO:** unified stream/batch engines use this policy on shared executor worker machines (based on job’s priorities and submission jobs).
  - each job gets priority on **all** resources as long as its stages have tasks to launch.

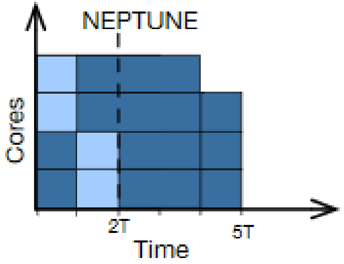


- **FAIR:** runs tasks on shared executors in a round-robin fashion. All jobs receive an equal share of resources.
  - Achieves better overall utilization and reduces the stream response time.

Scheduling hybrid dataflow applications



- **KILL:** to avoid the queuing delays for the latency-sensitive stream, one policy is to do a non-work conserving preemption by killing tasks of the batch job.
  - Side-effects may be bad for overall system performance
  - The batch job needs to start from scratch → more redundant work.



- **SUSPEND:** Ideally, the framework should suspend the batch jobs in favor of higher-priority stream tasks and resume them when resources are available.
  - Both minimizes the queuing latency and the wasted work.

Cluster-level scheduling

Cluster scheduling

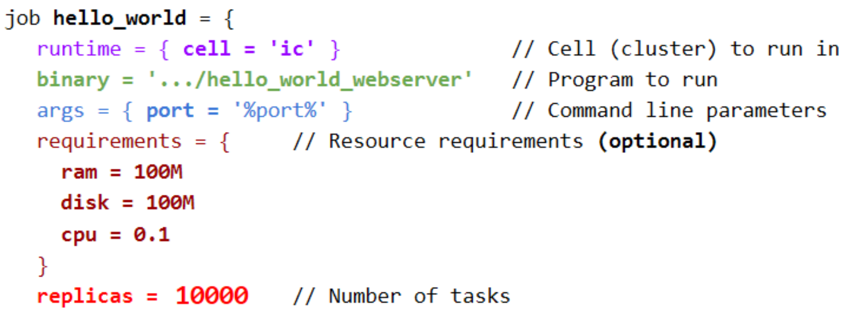
- **Cluster managers** enable resource sharing across users, frameworks and applications
  - Package hardware resources (e.g., CPU, disk, memory) in containers, allocated on demand

- **Optimization goals:**
  - machines are efficiently shared,
  - container and thus application performance is not hindered, and
  - the majority of application- and cluster-level requirements are satisfied.

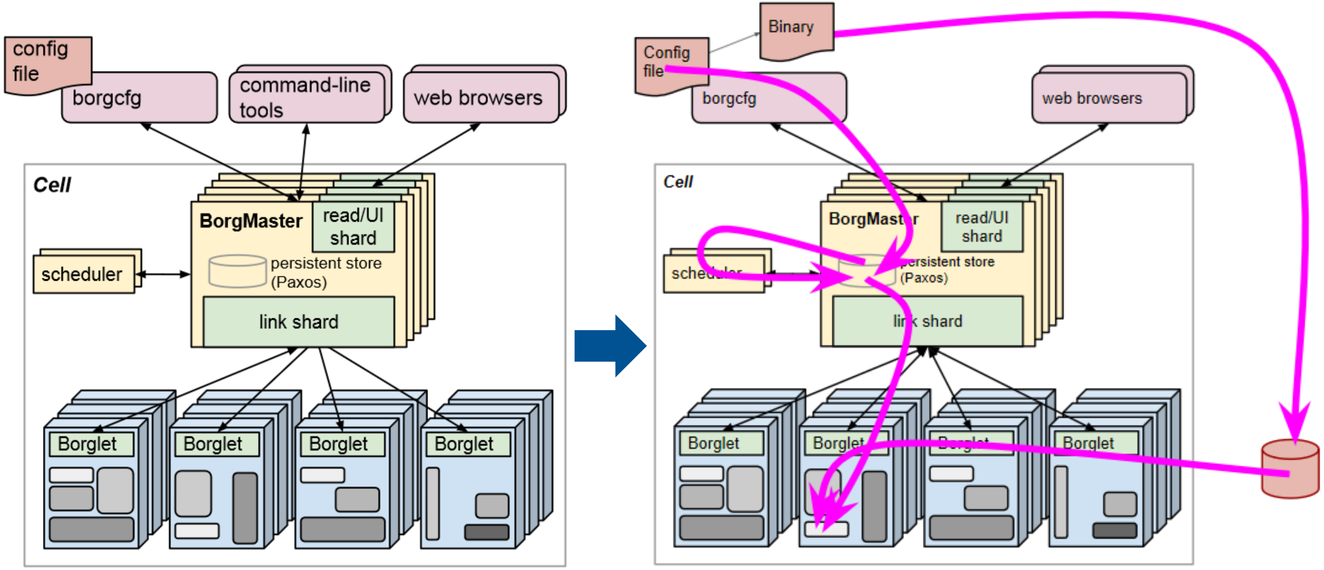
- But, **meeting these goals is hard:**
  - cluster workloads grow richer,
  - workload requirements become more diverse, and
  - clusters grow in size.

Behind the curtains

- The cluster scheduler receives a job.
- These requests identify:
  – The executable binary, the resources needed, the number of replicas, the job priority, etc.



```
job hello_world = {
  runtime = { cell = 'ic' }                 // Cell (cluster) to run in
  binary = '.../hello_world_webserver'      // Program to run
  args = { port = '%port%' }                // Command line parameters
  requirements = {      // Resource requirements (optional)
    ram = 100M
    disk = 100M
    cpu = 0.1
  }
  replicas = 10000    // Number of tasks
```

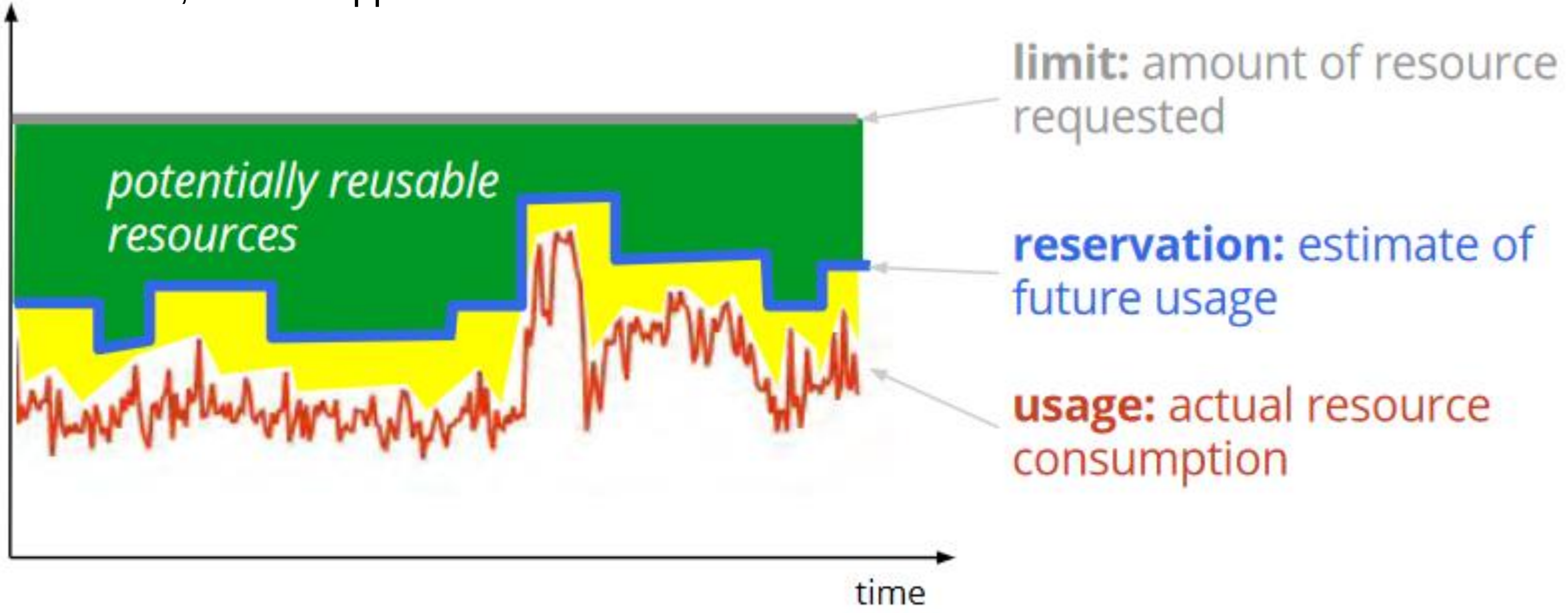


- Each replica is a task.
- The role of the scheduler is to place each task onto a suitable machine.

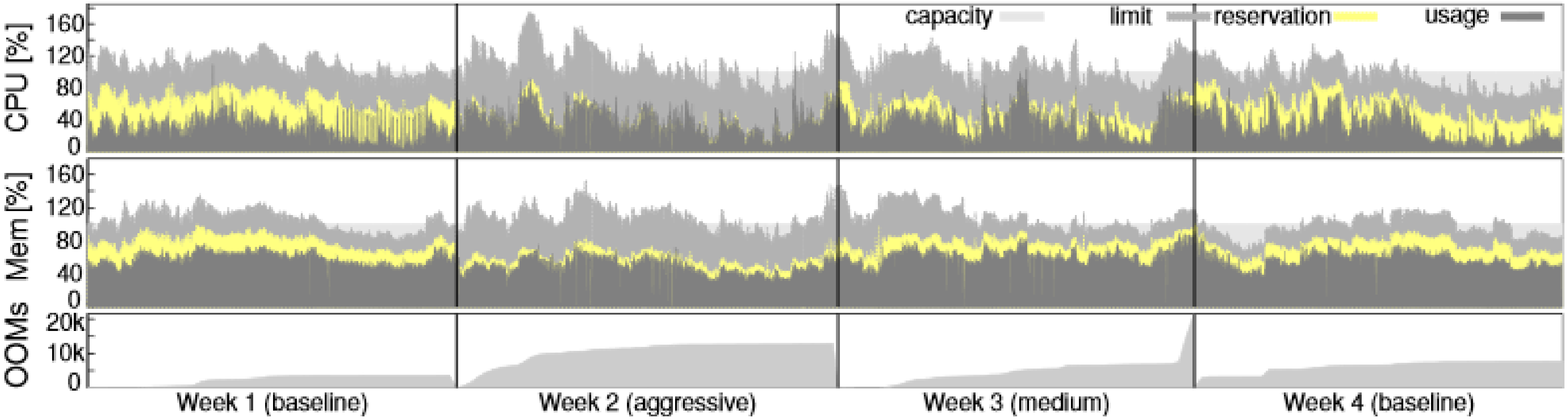
But how efficient is that?

- **Users** tend to be conservative and often **over-estimate the resources they’d need.**
  - Better than an expensive job to be terminated because it exceeded the resource limit.

- Yet, that **results** in a lot of **waste** of hardware resources!
  - Hence, look for opportunities for **resource reclamation.**



Benefits of resource reclamation



Closely matching the resource requirements, but leads to more out-of-memory events!

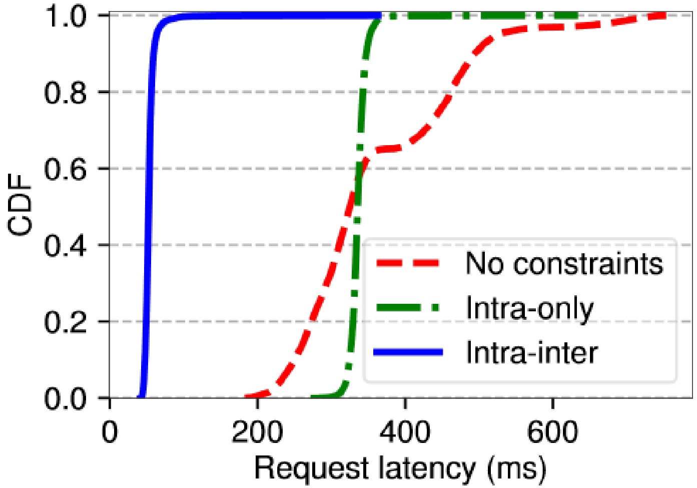
Sweet-spot between efficient resource usage and OOMs job terminations.

Sensitivity of job/task placement I

- Cluster with 275 node
  - 8 CPU cores, 128 GB memory, 3 TB of HDD storage, connected on a 10Gbps betwork.
  - Various LRAs are deployed such as HBase, TensorFlow and Storm

- **Constraints:**
  - **Affinity**: collocate the containers of a long running application on the same node or group of nodes as other containers

    e.g., to **reduce network traffic between communicating containers** within the same or across different applications

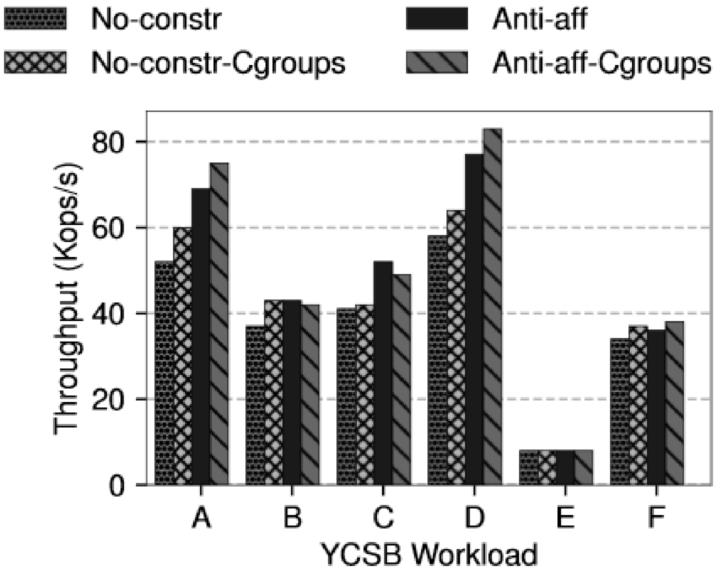


Sensitivity of job/task placement II

- Cluster with 275 node
  - 8 CPU cores, 128 GB memory, 3 TB of HDD storage, connected on a 10Gbps betwork.
  - Various LRAs are deployed such as HBase, TensorFlow and Storm

- **Constraints:**
  - **Anti-affinity**: may be desirable to place the containers on different machines through intra- and inter-application anti-affinity.
  - E.g. to **minimize resource interference.**

.

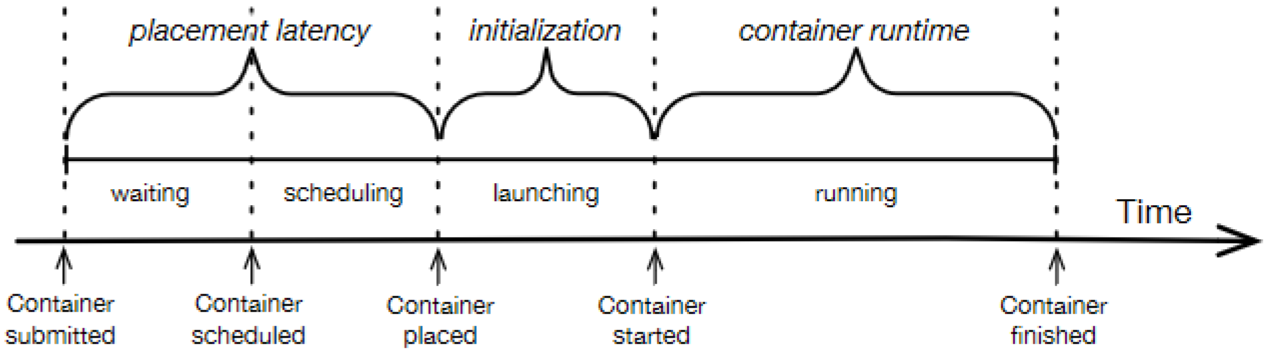


Policy quality/placement time trade-offs

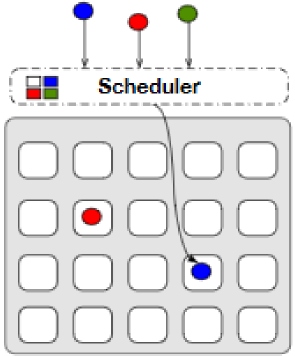
- Higher quality container placement using sophisticated scheduling in general is essential for achieving higher cluster utilization, more predictable application performance and increased application resilience.

- Yet, high placement latency (in the order of seconds) is unacceptable for particular types of workloads such as batch jobs with shorter container runtimes.

- Trade-off between low scheduling latency and high quality container placement

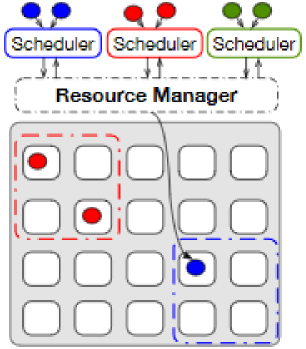


Cluster scheduling architectures



Centralised scheduler

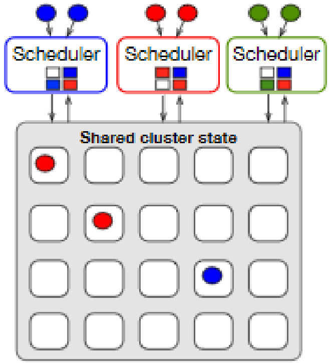
- **Centralized**
  Schedulers use the same scheduling logic to choose placements for all and maintain up-to-date information centrally.
  - Pros: high quality placement by implementing sophisticated scheduling algorithms.
  - Cons: (potentially) long placement latency
  - e.g., Borg, Bistro, Quincy, Firmament, Quasar, etc.



Two-level scheduler

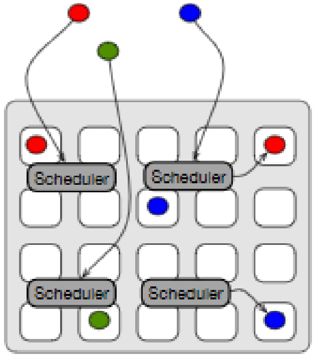
- **Two-level**
  A simpler resource manager that allocates containers, and a number of application specific schedulers, aware of their independent needs.
  - Resources are managed centrally, but scheduling is delegated to frameworks that deal with workloads requirements: batch, streaming, ML, etc.
  - e.g. Apache Mesos (for Apache Spark).

Cluster scheduling architectures



Shared-state scheduler

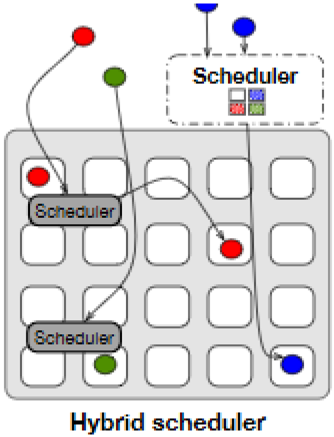
- **Shared-state**
  Similar to 2-level scheduling, but allow multiple application schedulers with a different internal scheduling logic to have a complete view of the cluster state.
  - e.g., (Google’s) Omega supports multiple schedulers with each dealing with a fraction of the total cluster workload, Kubernetes



Distributed scheduler

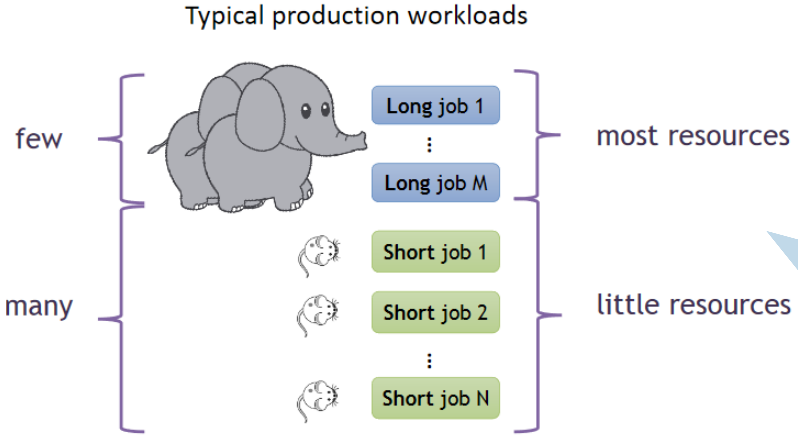
- **Distributed**
  Schedulers that achieve extreme scalability and low-latency allocations.
  - Use worker nodes that pull tasks directly from distributed schedulers or maintain a queue of tasks locally to minimize the period when they remain idle.
  - e.g. Sparrow and Microsoft’s Apollo.

Cluster scheduling architectures



Hybrid scheduler

- **Hybrid**
  combine the high quality placement of centralized schedulers and the low scheduling latency of distributed schedulers.
  – e.g., Microsoft's Mercury, Yaq, academia Hawk



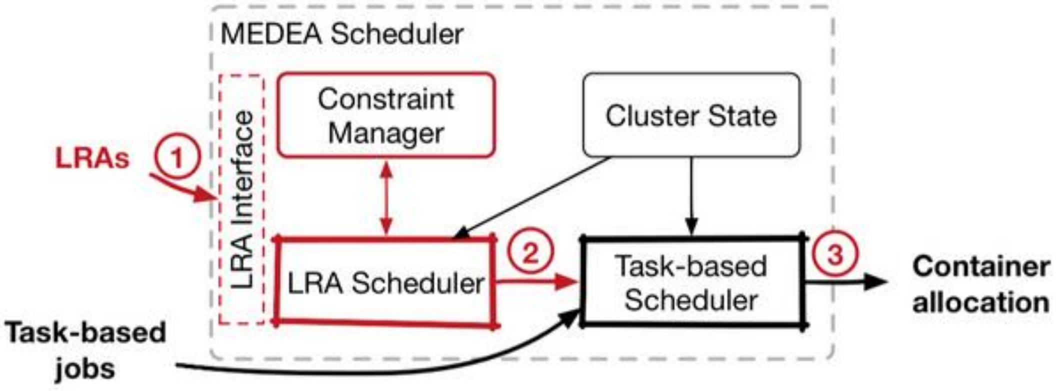
Google's traces suggest: 99% of the resources are taken by 1% of the jobs.

Yet, the scheduler needs to ensure not to damage the "mice" jobs.

Hybrid schedulers with affinities

- Task-based jobs go directly to the task-scheduler, which does the resource allocation.

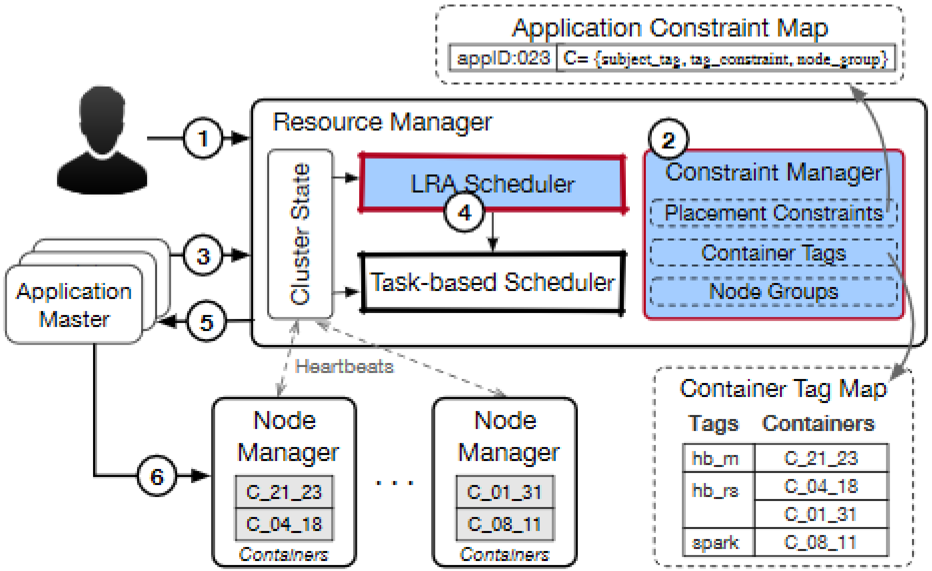
- Long-running applications (LRAs) go to the LRA scheduler that makes placement decisions, later passed to the task-scheduler that does allocations.



- To manage placement constraints (both from the application owners and cluster operators), a new central component keeps all the data (global view of all active constraints).

- The LRA placement is then an optimization problem under a set of constraints, expressed as an integer linear programming (ILP) problem.

Implementing it in practice



Tags	Containers
hb_m	C_21_23
hb_rs	C_04_18
	C_01_31
spark	C_08_11

YARN – centralized architecture, where the *Resource Manager* (RM) allocates resources on *Node Managers* (NMs) for applications submitted to the cluster.

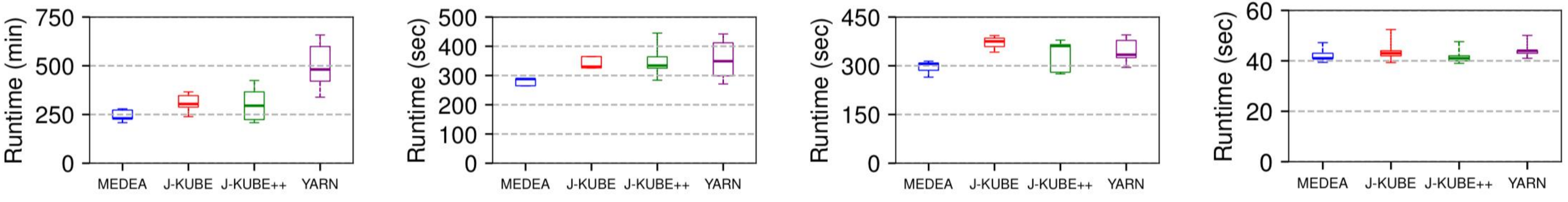
Any application-specific scheduling logic is done by the *Application Master.*

The *Node Managers* launch and manage containers and monitor resource availability.

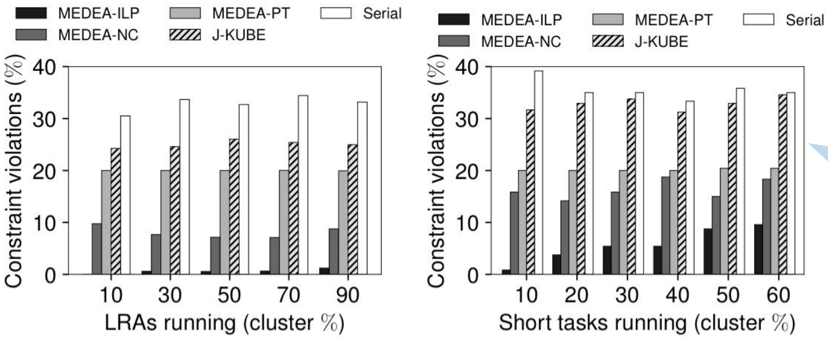
MEDEA extends YARN by adding:

- *Constraint Manager (CM)*
- *LRA scheduler*

Results



(a) TensorFlow **(b)** HBase insert **(c)** HBase workloadA **(d)** GridMix workload

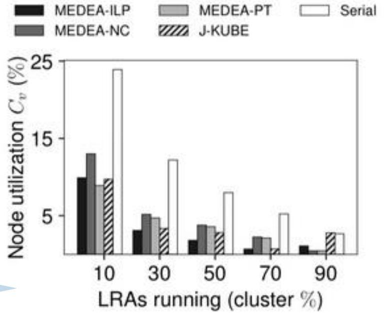


(a) Varying cluster LRA utilization **(b)** Varying task-based utilization

Shorter runtimes

Less constraints are violated.

Better cluster utilization.



(b) Coefficient of variation for node memory utilization

References

The material covered in this class is mainly based on:

- PhD thesis from Dr. Panagiotis Garefalakis
  - *Supporting Long-Running Applications in Shared Compute Clusters* (Imperial College London, 2020) (link)
- Slides by Dr. John Wilkes (Google) – *Cluster management at Google with Borg – coping with scale* (link)

Papers:

- Dean and Barros. *The tail at scale.* Communications of the ACM'13.
- Verma et al. *Large-scale cluster management at Google with Borg.* EuroSys'15.
- Delgado et al. *Hawk: Hybrid Datacenter Scheduling.* ATC'15.
- Burns et al. *Borg, Omega, and Kubernetes.* ACM Queue'16.
- Garefalakis et al. *Medea: scheduling of long running applications in shared production clusters.* EuroSys'18
- Garefalakis et al. *Neptune: scheduling suspendable tasks for unified streaming/batch applications.* SoCC'19

Further reading:

- Ousterhaut et al. *Sparrow: Distributed, Low Latency Scheduling.* SOSP'13
- Karanasos et al. *Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters.* ATC'15.
- Gog et al. *Firmament: Fast, Centralized Cluster Scheduling at Scale.* OSDI'16