Omega: flexible, scalable schedulers for large compute clusters

A 2013 paper by Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes

Presented by Matt Levan

Outline

Abstract

Abstract

Monolithic and two-level schedulers are vulnerable to problems.

A new scheduler architecture (Omega, heir of Borg) utilizes these concepts to enable implementation extensibility and performance scalability:

- Shared-state
- Parallelism
- Optimistic concurrency control

Background

Problem

Data centers are expensive. We need to utilize their resources more efficiently!

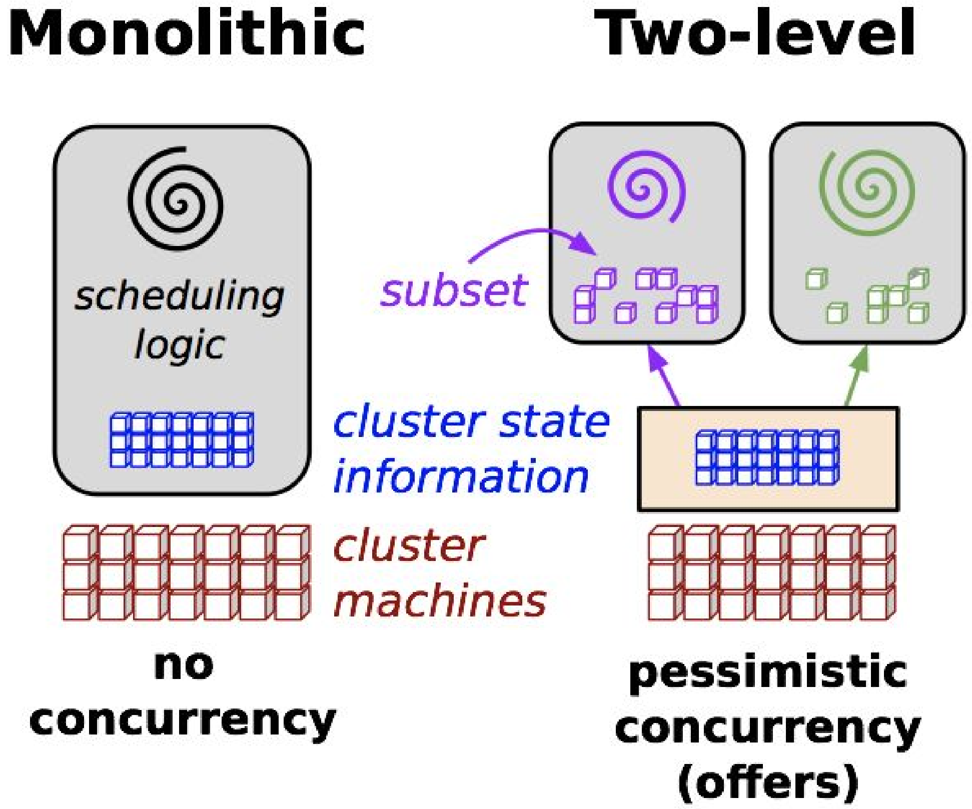
Issues with prevalent scheduler architectures:

- **Monolithic schedulers** risk becoming scalability bottlenecks.
- **Two-level schedulers** limit resource visibility and parallelism.

Scheduler definitions

Monolithic scheduler: a single process responsible for accepting workloads, ordering them, and sending them to appropriate machines for processing, all according to internal and user-defined policies. Single resource manager and single scheduler.

Two-level scheduler: Single resource manager serves multiple, parallel, independent schedulers using conservative resource allocation (pessimistic concurrency) and locking algorithms.


Monolithic
Two-level
scheduling logic
subset
cluster state information
cluster machines
no concurrency
pessimistic concurrency (offers)

[1:1]

Designing Omega

Workloads

- Different types of jobs have different requirements [1:2]:
  - Batch: >80% in Google data centers
  - Service: majority of resources (55-80%)
- “Head of line blocking” problem [1:3]:
  - Placing service jobs for best availability and performance is NP-hard.
  - Can be solved with parallelism
- New scheduler must be flexible, handling:
  - Job-specific policies
  - Ever-growing resources and workloads

Requirements

New scheduler architecture must meet these requirements *simultaneously*:

1. High resource utilization (despite increasing infrastructure *and* workloads)
2. Job-specific placement and policy constraints
3. Fast decision making
4. Varying degrees of fairness (based on business importance)
5. Highly available and robust

Taxonomy

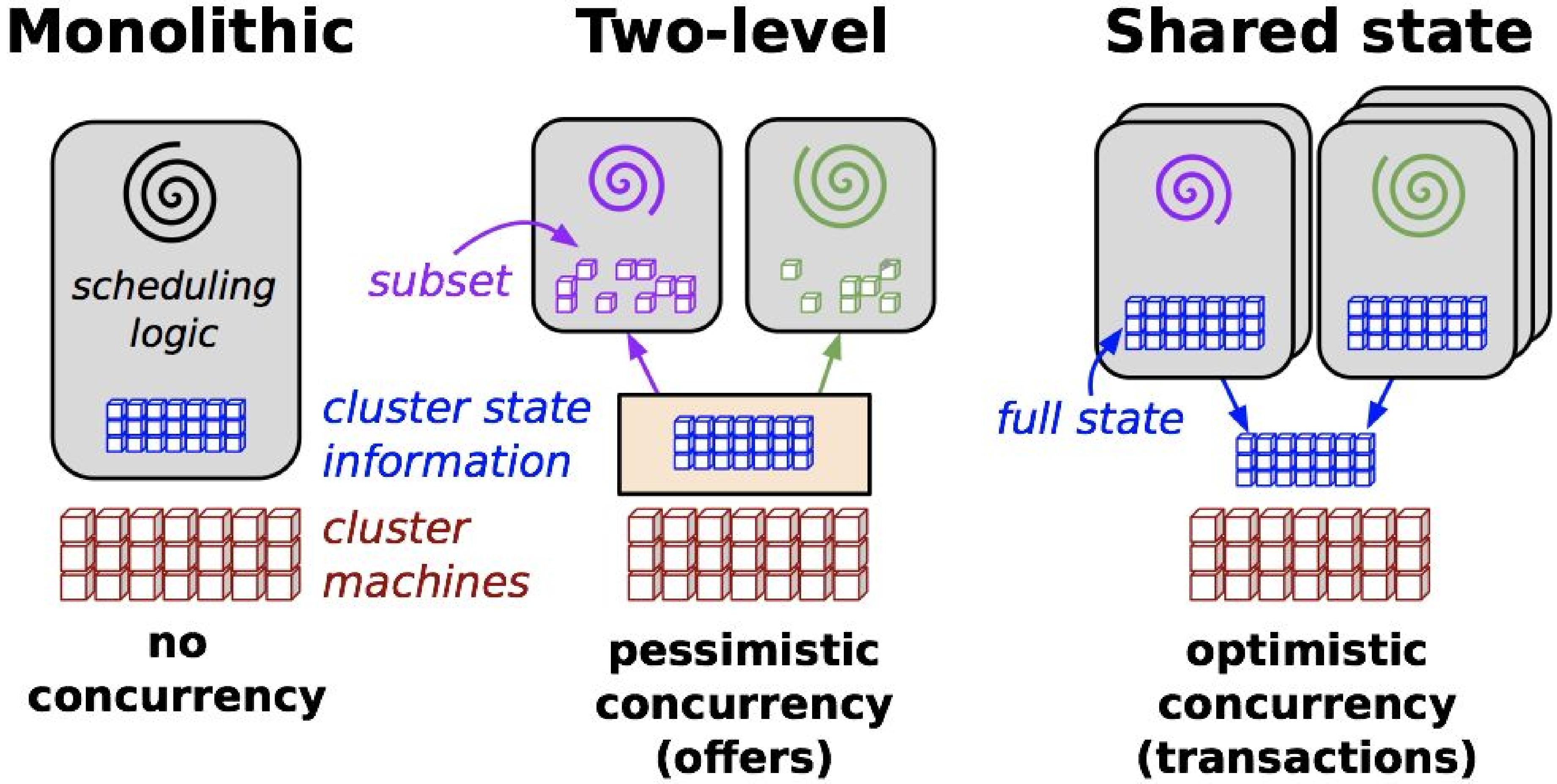
Cluster schedulers must address these design issues, including how to:

- **Partition** incoming jobs
- **Choose** resources
- **Interfere** when schedulers compete for resources
- **Allocate** jobs (atomically or incrementally as resources for tasks are found)
- **Moderate** cluster-wide behavior

Omega design choices

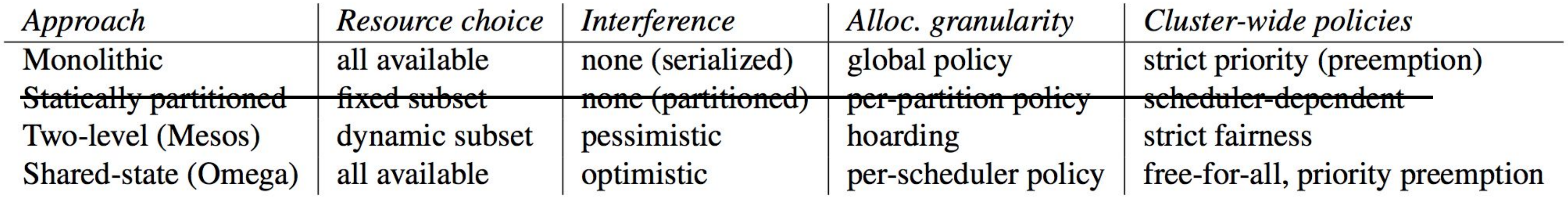
- **Partition incoming jobs:** Schedulers are omniscient; compete in free-for-all.
- **Choose resources:** Schedulers have complete freedom; use *cell state*.
- **Interfere when schedulers compete for resources:** Only one update to global cell state accepted at a time. If denied resources, try again.
- **Allocate jobs:** Schedulers can choose incremental or all-or-nothing transactions.
- **Moderate cluster-wide behavior:** Schedulers must agree on common definitions of resource status (such as machine fullness) as well as job *precedence*. There is no central policy-enforcement engine.

The performance of this approach is “determined by the frequency at which transactions fail and the costs of such failures [1:5].”


Monolithic
Two-level
Shared state
scheduling logic
subset
cluster state information
full state
cluster machines
no concurrency
pessimistic concurrency (offers)
optimistic concurrency (transactions)

[1:1]

Omega design choices



Approach	*Resource choice*	*Interference*	*Alloc. granularity*	*Cluster-wide policies*
Monolithic	all available	none (serialized)	global policy	strict priority (preemption)
~~Statically partitioned~~	~~fixed subset~~	~~none (partitioned)~~	~~per-partition policy~~	~~scheduler-dependent~~
Two-level (Mesos)	dynamic subset	pessimistic	hoarding	strict fairness
Shared-state (Omega)	all available	optimistic	per-scheduler policy	free-for-all, priority preemption

Table 1: Comparison of parallelized cluster scheduling approaches.

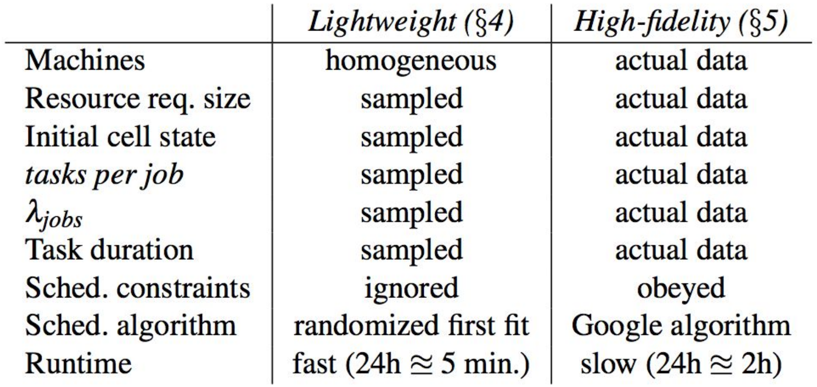
[1:4]

Evaluation

Simulation setup

Trade-offs between the three scheduling architectures (monolithic, two-level, and shared-state) are measured via two simulators:

1. **Lightweight simulator** uses synthetic, but simplified, workloads (inspired by Google workloads).
2. **High-fidelity simulator** replays actual Google production cluster workloads.



	Lightweight (§4)	*High-fidelity (§5)*
Machines	homogeneous	actual data
Resource req. size	sampled	actual data
Initial cell state	sampled	actual data
tasks per job	sampled	actual data
λ_{jobs}	sampled	actual data
Task duration	sampled	actual data
Sched. constraints	ignored	obeyed
Sched. algorithm	randomized first fit	Google algorithm
Runtime	fast (24h $\approxeq$ 5 min.)	slow (24h $\approxeq$ 2h)

[1:5]

Lightweight simulation setup

Parameters:

- Scheduler decision time: $\boldsymbol{t_{decision} = t_{job} + t_{task} * \text{tasks per job}}$
  - t_{job} is a per-job overhead cost.
  - t_{task} is cost to place each task.

Metrics:

- *job wait time* is time from submission to first scheduling attempt.
- *scheduler busyness* is time during which scheduler is busy making decisions.
- *conflict fraction* is average number of conflicts per successful transaction.

Lightweight simulator is simplified for better accuracy as shown in Table 2.

Lightweight simulation: Monolithic

Monolithic schedulers (baseline for comparison):

- Single and multi-path simulations are performed.
- Scheduler decision time varies on x-axis by changing t_{job}.
- Workload split by batch or service types.
- Scheduler busyness is low as long as scheduling is quick, scales linearly with increased t_{job}.
  - Job wait time increases at a similar rate until scheduler is saturated and can't keep up.
- Head-of-line blocking occurs when batch jobs get stuck in queue behind slow-to-schedule service jobs.
  - Scalability is limited.

Lightweight simulation: Two-level

Two-level scheduling (inspired by Apache Mesos):

- Single resource manager; two scheduler frameworks (batch, service).
  - Schedulers only see resources available to it when it begins a scheduling attempt.
- Decision time for batch scheduler constant, variable for service scheduler.
- Batch scheduler busyness is much higher than multi-path monolithic scheduler.
  - Mesos alternately offers *all* available cluster resources to different schedulers.
  - Thus, if scheduler takes a long time to decide, nearly all cluster resources are locked!
  - Additionally, batch jobs sit in limbo as their scheduler tries (up to 1000x) again and again to allocate resources for them.

Due to assumption that service jobs won’t consume most of the cluster’s resources, two-level is hindered by pessimistic locking and can’t handle Google’s workloads.

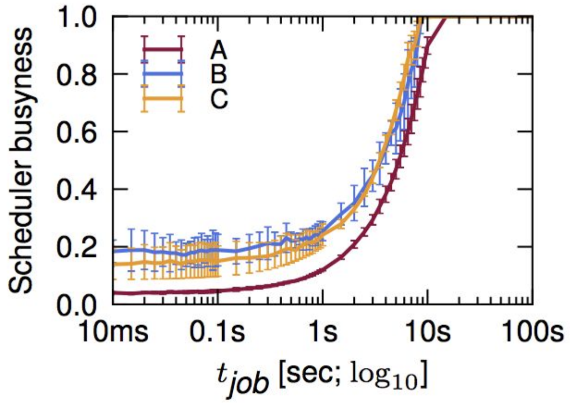
Lightweight simulation: Shared-state

Shared-state scheduling (Omega):

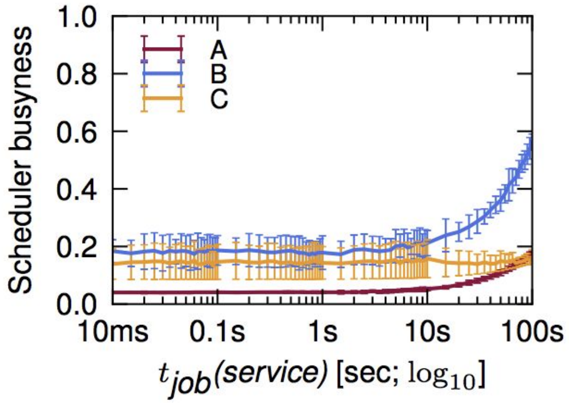
- Again, two schedulers (one batch, one service). Each refreshes cell state every time it looks for resources to allocate a job.
- Entire transaction accepted OR only tasks that will not result in an overcommitted machine.
- Conflicts and interference occur rarely.
- **No head-of-line blocking!** Lines for batch and service jobs are independent.
- Batch scheduler becomes scalability bottleneck:
  - Solved easily by adding more batch schedulers, load-balanced by simple hashing function.

Omega can scale to high batch workload while still providing good performance and availability for service jobs.

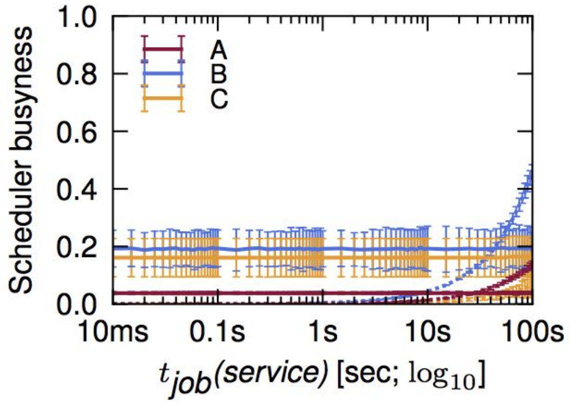
Scheduler busyness: Monolithic vs. shared-state



(a) Single-path.



(b) Multi-path.

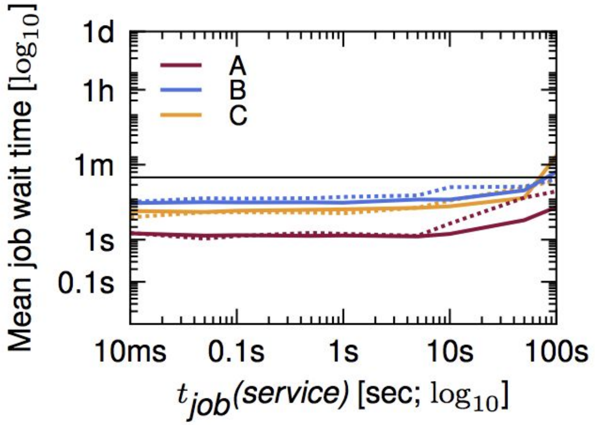


(c) Shared state.

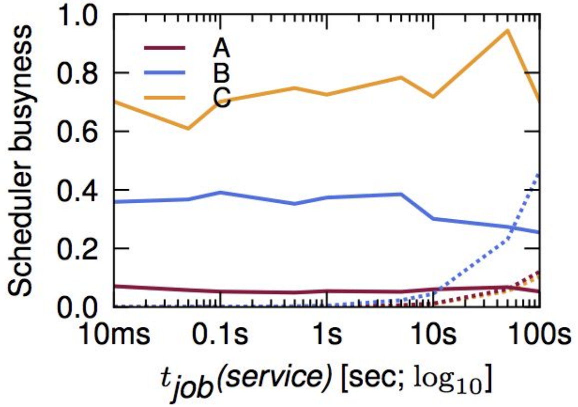
Figure 6: Schedulers' busyness, as a function of t_{job} in the monolithic single-path case, $t_{job}(service)$ in the monolithic multi-path and shared-state cases. The value is the median daily busyness over the 7-day experiment, and error bars are one $\pm$ median absolute deviation (MAD), i.e. the median deviation from the median value, a robust estimator of typical value dispersion.

[1:6]

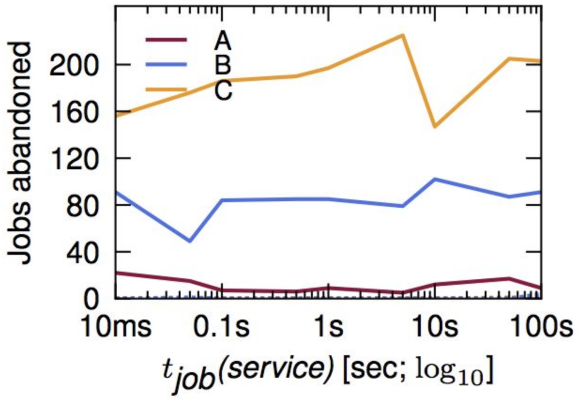
All metrics: Two-level scheduling



(a) Job wait time.



(b) Scheduler busyness.



(c) Unscheduled jobs.

Figure 7: Two-level scheduling (Mesos): performance as a function of $t_{job}(service)$.

[1:8]

Conclusion

Conclusion

- Monolithic scheduler is not scalable.
- Two-level model “is hampered by pessimistic locking” and can’t schedule heterogeneous workload offered by Google [1:8].
- Omega shared-state model scales, supports custom schedulers, and can handle a variety of workloads.

Future work:

- Take a look at the high-fidelity simulation in this paper.
- Explore Kubernetes scheduler as it is the heir of Omega and is open-source.
- Implement batch/service job types in Mishael’s existing simulation.

References

[1] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (EuroSys '13). ACM, New York, NY, USA, 351-364. DOI=http://0-dx.doi.org.torofind.csudh.edu/10.1145/2465351.2465386

[2] Google's Lightweight Cluster Simulator can be found here: https://github.com/google/cluster-scheduler-simulator