# Omega: flexible, scalable schedulers for large compute clusters

Malte Schwarzkopf[†] [*] Andy Konwinski[‡] [*] Michael Abd-El-Malek[§] John Wilkes[§]

[†]University of Cambridge Computer Laboratory [‡]University of California, Berkeley [§]Google, Inc.

[†]ms705@cl.cam.ac.uk [‡]andyk@berkeley.edu [§]{mabdelmalek,johnwilkes}@google.com

## Abstract

Increasing scale and the need for rapid response to changing requirements are hard to meet with current monolithic cluster scheduler architectures. This restricts the rate at which new features can be deployed, decreases efficiency and utilization, and will eventually limit cluster growth. We present a novel approach to address these needs using parallelism, shared state, and lock-free optimistic concurrency control.

We compare this approach to existing cluster scheduler designs, evaluate how much interference between schedulers occurs and how much it matters in practice, present some techniques to alleviate it, and finally discuss a use case highlighting the advantages of our approach – all driven by real-life Google production workloads.

***Categories and Subject Descriptors*** D.4.7 [*Operating Systems*]: Organization and Design—Distributed systems; K.6.4 [*Management of computing and information systems*]: System Management—Centralization/decentralization

***Keywords*** Cluster scheduling, optimistic concurrency control

## 1. Introduction

Large-scale compute clusters are expensive, so it is important to use them well. Utilization and efficiency can be increased by running a mix of workloads on the same machines: CPU- and memory-intensive jobs, small and large ones, and a mix of batch and low-latency jobs – ones that serve end user requests or provide infrastructure services such as storage, naming or locking. This consolidation reduces the amount of hardware required for a workload, but it makes the scheduling problem (assigning jobs to machines) more complicated: a wider range of requirements and policies have to be taken into account. Meanwhile, clusters and their workloads keep growing, and since the scheduler's workload is roughly proportional to the cluster size, the scheduler is at risk of becoming a scalability bottleneck.

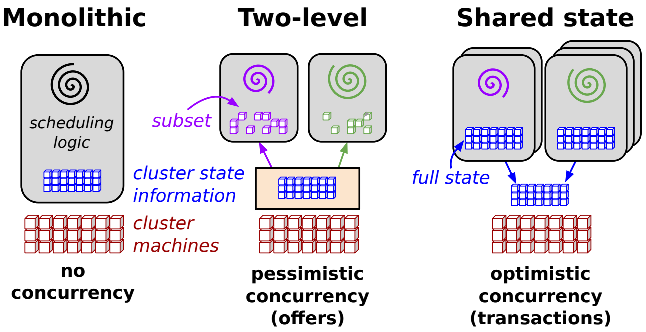

Monolithic
Two-level
Shared state
scheduling logic
subset
cluster state information
cluster machines
full state
no concurrency
pessimistic concurrency (offers)
optimistic concurrency (transactions)

**Figure 1:** Schematic overview of the scheduling architectures explored in this paper.

Google's production job scheduler has experienced all of this. Over the years, it has evolved into a complicated, sophisticated system that is hard to change. As part of a rewrite of this scheduler, we searched for a better approach.

We identified the two prevalent scheduler architectures shown in Figure 1. *Monolithic schedulers* use a single, centralized scheduling algorithm for all jobs (our existing scheduler is one of these). *Two-level* schedulers have a single active resource manager that offers compute resources to multiple parallel, independent "scheduler frameworks", as in Mesos [13] and Hadoop-on-Demand [4].

Neither of these models satisfied our needs. Monolithic schedulers do not make it easy to add new policies and specialized implementations, and may not scale up to the cluster sizes we are planning for. Two-level scheduling architectures do appear to provide flexibility and parallelism, but in practice their conservative resource-visibility and locking algorithms limit both, and make it hard to place difficult-to-schedule "picky" jobs or to make decisions that require access to the state of the entire cluster.

Our solution is a new parallel scheduler architecture built around *shared state*, using lock-free optimistic concurrency control, to achieve both implementation extensibility and performance scalability. This architecture is being used in

[*] Work done while interning at Google, Inc.

*EuroSys'13* April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04...$15.00

*Omega*, Google's next-generation cluster management system.

### 1.1 Contributions

The contributions of this paper are as follows. We:

1. present a lightweight taxonomy of the option space for cluster scheduler development (§3);
2. introduce a new scheduler architecture using shared state and lock-free optimistic concurrency control (§3.4);
3. compare the performance of monolithic, two-level and shared-state scheduling using simulations and synthetic workloads (§4);
4. explore the behavior of the shared-state approach in more detail using code based on a production scheduler and driven by real-world workload traces (§5); and
5. demonstrate the flexibility of the shared-state approach by means of a use case: we add a scheduler that uses knowledge of the global cluster utilization to adjust the resources given to running MapReduce jobs (§6).

We find that the Omega shared-state architecture can deliver performance competitive with or superior to other architectures, and that interference in real-world settings is low. The ability to access the entire cluster state in a scheduler brings other benefits, too, and we demonstrate this by showing how MapReduce jobs can be accelerated by using spare resources.

## 2. Requirements

Cluster schedulers must meet a number of goals simultaneously: high resource utilization, user-supplied placement constraints, rapid decision making, and various degrees of "fairness" and business importance – all while being robust and always available. These requirements evolve over time, and, in our experience, it becomes increasingly difficult to add new policies to a single monolithic scheduler. This is not just due to accumulation of code as functionality grows over time, but also because some of our users have come to rely on a detailed understanding of the internal behavior of the system to get their work done, which makes both its functionality and structure difficult to change.

### 2.1 Workload heterogeneity

One important driver of complexity is the hardware and workload heterogeneity that is commonplace in large compute clusters [24].

To demonstrate this, we examine the workload mix on three Google production compute clusters that we believe to be representative. Cluster A is a medium-sized, fairly busy one, while cluster B is one of the larger clusters currently in use at Google, and cluster C is the one for which a scheduler workload trace was recently published [24, 27]. The workloads are from May 2011. All the clusters run a wide variety of jobs; some are configured by hand; some by automated systems such as MapReduce [8], Pregel [19] and Percolator [23].

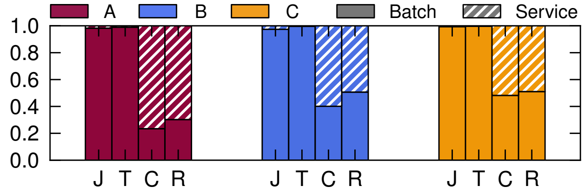


**Figure 2:** Batch and service workloads for the clusters A, B, and C: normalized numbers of jobs (**J**) and tasks (**T**), and aggregate requests for CPU-core-seconds (**C**) and RAM GB-seconds (**R**). The striped portion is the service jobs; the rest is batch jobs.

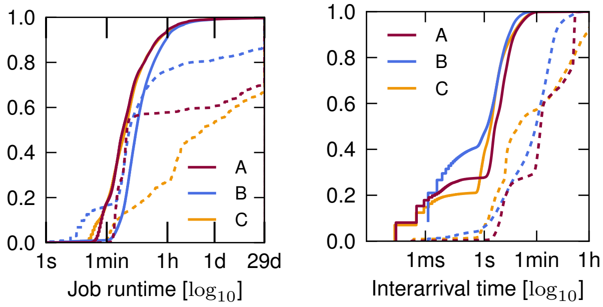


**Figure 3:** Cumulative distribution functions (CDFs) of job runtime and job inter-arrival times for clusters A, B, and C. Where the lines do not meet 1.0, some of the jobs ran for longer than the 30-day range. In this and subsequent graphs, solid lines represent batch jobs, and dashed lines are for service jobs.

There are many ways of partitioning a cluster's workload between different schedulers. Here, we pick a simple two-way split between long-running *service* jobs that provide end-user operations (e.g., web services) and internal infrastructure services (e.g., BigTable [5]), and *batch* jobs which perform a computation and then finish. Although many other splits are possible, for simplicity we put all low priority jobs[1] and those marked as "best effort" or "batch" into the batch category, and the rest into the service category.

A job is made up of one or more tasks – sometimes thousands of tasks. Most (>80%) jobs are batch jobs, but the majority of resources (55–80%) are allocated to service jobs (Figure 2); the latter typically run for much longer (Figure 3), and have fewer tasks than batch jobs (Figure 4). These results are broadly similar to other analyses of cluster traces from Yahoo [17], Facebook [7] and Google [20, 24, 25, 29].

Why does this matter? Many batch jobs are short, and fast turnaround is important, so a lightweight, low-quality

[1] In the public trace for cluster C, these are priority bands 0–8 [27].

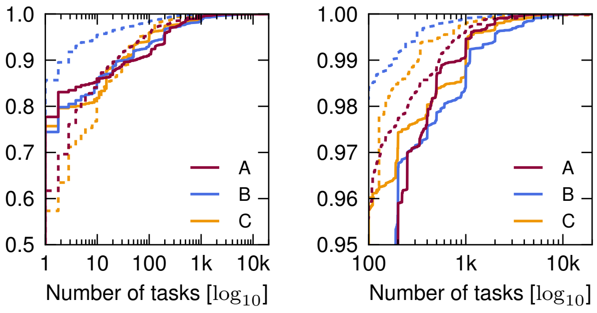


**Figure 4:** CDF of the number of tasks in a job for clusters A, B, and C. The right hand graph is an expansion of the tail of the left-hand one, looking at $\geq 95^{\text{th}}$ percentile, $\geq$ 100 tasks.

approach to placement works just fine. But long-running, high-priority service jobs (20–40% of them run for over a month) must meet stringent availability and performance targets, meaning that careful placement of their tasks is needed to maximize resistance to failures and provide good performance. Indeed, the Omega service scheduler will try to place tasks to resist both independent and coordinated failures, which is an NP-hard chance-constrained optimization problem with tens of failure domains that nest and overlap. Our previous implementation could take tens of seconds to do this. While it is very reasonable to spend a few seconds making a decision whose effects last for several weeks, it can be problematic if an interactive batch job has to wait for such a calculation. This problem is typically referred to as "head of line blocking", and can be avoided by introducing parallelism.

In summary, what we require is a scheduler architecture that can accommodate both types of jobs, flexibly support job-specific policies, and also scale to an ever-growing amount of scheduling work. The next section examines some of these requirements in greater detail, as well as some approaches to meeting them.

## 3. Taxonomy

We begin with a short survey of the design issues cluster schedulers must address, followed by an examination of some different scheduler architectures that might meet them.

**Partitioning the scheduling work.** Work can be spread across schedulers by (1) load-balancing that is oblivious to workload type; (2) dedicating specialized schedulers to different parts of the workload; or (3) a combination of the two. Some systems use multiple job queues to hold the job requests (e.g., for different priorities), but that does not affect the scheduling parallelism: we are more interested in how many schedulers are assigned to process the queues.

**Choice of resources.** Schedulers can be allowed to select from all of the cluster resources, or limited to a subset to streamline decision making. The former increases the opportunity to make better decisions, and is important when "picky" jobs need to be placed into a nearly-full cluster, or when decisions rely on overall state, such as the total amount of unused resources. Schedulers can have greater flexibility in placing tasks if they can preempt existing assignments, as opposed to merely considering idle resources, but this comes at the cost of wasting some work in the preempted tasks.

**Interference.** If schedulers compete for resources, multiple schedulers may attempt to claim the same resource simultaneously. A pessimistic approach avoids the issue by ensuring that a particular resource is only made available to one scheduler at a time; an optimistic one detects the (hopefully rare) conflicts, and undoes one or more of the conflicting claims. The optimistic approach increases parallelism, but potentially increases the amount of wasted scheduling work if conflicts occur too frequently.

**Allocation granularity.** Since jobs typically contain many tasks, schedulers can have different policies for how to schedule them: at one extreme is atomic all-or-nothing gang scheduling of the tasks in a job, at the other is incremental placement of tasks as resources are found for them. An all-or-nothing policy can be approximated by incrementally acquiring resources and hoarding them until the job can be started, at the cost of wasting those resources in the meantime.

All have downsides: gang scheduling may be needed by some jobs (e.g., MPI programs), but can unnecessarily delay the start of others that can make progress with only a fraction of their requested resources (e.g., MapReduce jobs). Incremental resource acquisition can lead to deadlock if no backoff mechanism is provided, while hoarding reduces cluster utilization and can also cause deadlock.

**Cluster-wide behaviors.** Some behaviors span multiple schedulers. Examples include achieving various types of fairness, and a common agreement on the relative importance of work, especially if one scheduler can preempt others' tasks. Strict enforcement of these behaviors can be achieved with centralized control, but it is also possible to rely on emergent behaviors to approximate the desired behavior. Techniques such as limiting the range of priorities that a scheduler can wield can provide partial enforcement of desired behaviors, and compliance to cluster-wide policies can be audited *post facto* to eliminate the need for checks in a scheduler's critical code path.

This space is obviously larger than can be explored in a single paper; we focus on the combinations that are summarized in Table 1, and described in greater detail in the next few sections.

### 3.1 Monolithic schedulers

Our baseline for comparisons is a monolithic scheduler that has but a single instance, no parallelism, and must implement all the policy choices in a single code base. This approach is common in the high-performance computing (HPC) world, where a monolithic scheduler usually runs a

| *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.

single instance of the scheduling code, and applies the same algorithm for all incoming jobs. HPC schedulers such as Maui [16] and its successor Moab, as well as Platform LSF [14], support different policies by means of a complicated calculation involving multiple weighting factors to calculate an overall priority, after which "the scheduler can roughly fulfill site objectives by starting the jobs in priority order" [1].

Another way to support different scheduling policies is to provide multiple code paths in the scheduler, running separate scheduling logic for different job types. But this is harder than it might appear. Google's current cluster scheduler is effectively monolithic, although it has acquired many-optimizations over the years to provide internal parallelism and multi-threading to address head-of-line blocking and scalability. This complicates an already difficult job: the scheduler has to minimize the time a job spends waiting before it starts running, while respecting priorities, per-job constraints [20, 25], and a number of other policy goals such as failure-tolerance and scaling to workloads that fill many thousands of machines. Although it has been hugely successful, our scheduler has experienced several years of evolution and organic software growth, and we have found that it is surprisingly difficult to support a wide range of policies in a sustainable manner using a single-algorithm implementation. In the end, this kind of software engineering consideration, rather than performance scalability, was our primary motivation to move to an architecture that supported concurrent, independent scheduling components.

### 3.2 Statically partitioned schedulers

Most "cloud computing" schedulers (e.g., Hadoop [28], and Dryad's Quincy [15]) assume they have complete control over a set of resources, as they are typically deployed onto dedicated, statically-partitioned clusters of machines; or by partitioning a single cluster into different parts that support different behaviors [6]. This leads to fragmentation and suboptimal utilization, which is not viable for us, and so we did not explore this option any further.

### 3.3 Two-level scheduling

An obvious fix to the issues of static partitioning is to adjust the allocation of resources to each scheduler dynamically, using a central coordinator to decide how many resources each sub-cluster can have. This *two-level scheduling* approach is used by a number of systems, including Mesos [13] and Hadoop-on-Demand (HOD) [4].

In Mesos, a centralized resource allocator dynamically partitions a cluster, allocating resources to different *scheduler frameworks*.[2] Resources are distributed to the frameworks in the form of *offers*, which contain only "available" resources – ones that are currently unused. The allocator avoids conflicts by only offering a given resource to one framework at a time, and attempts to achieve dominant resource fairness (DRF) [11] by choosing the order and the sizes of its offers.[3] Because only one framework is examining a resource at a time, it effectively holds a lock on that resource for the duration of a scheduling decision. In other words, concurrency control is pessimistic.

Mesos works best when tasks are short-lived and relinquish resources frequently, and when job sizes are small compared to the size of the cluster. As we explained in §2.1, our cluster workloads do not have these properties, especially in the case of service jobs, and §4.2 will show that this makes an offer-based two-level scheduling approach unsuitable for our needs.

While a Mesos framework can use "filters" to describe the kinds of resources that it would like to be offered, it does not have access to a view of the overall cluster state – just the resources it has been offered. As a result, it cannot support preemption or policies requiring access to the whole cluster state: a framework simply does not have any knowledge of resources that have been allocated to other schedulers. Mesos uses resource hoarding to achieve gang scheduling, and can potentially deadlock as a result.

It might appear that YARN [21] is a two-level scheduler, too. In YARN, resource requests from per-job *application masters* are sent to a single global scheduler in the *resource master*, which allocates resources on various machines, subject to application-specified constraints. But the application masters provide job-management services, not scheduling, so YARN is effectively a monolithic scheduler architecture. At the time of writing, YARN only supports one resource type (fixed-sized memory chunks). Our experience suggests that it will eventually need a rich API to the resource master

[2] We describe the most recently released version of Mesos at the time we did this work: *0.9.0-incubating* from May 8, 2012.

[3] The Mesos "simple allocator" offers all available resources to a framework every time it makes an offer, and does not limit the amount of resources that a framework can accept. This negatively impacts Mesos as framework decision times grow; see §4.2.

in order to cater for diverse application requirements, including multiple resource dimensions, constraints, and placement choices for failure-tolerance. Although YARN application masters can request resources on particular machines, it is unclear how they acquire and maintain the state needed to make such placement decisions.

### 3.4 Shared-state scheduling

The alternative used by Omega is the *shared state* approach: we grant each scheduler full access to the entire cluster, allow them to compete in a free-for-all manner, and use optimistic concurrency control to mediate clashes when they update the cluster state. This immediately eliminates two of the issues of the two-level scheduler approach – limited parallelism due to pessimistic concurrency control, and restricted visibility of resources in a scheduler framework – at the potential cost of redoing work when the optimistic concurrency assumptions are incorrect. Exploring this tradeoff is the primary purpose of this paper.

There is no central resource allocator in Omega; all of the resource-allocation decisions take place in the schedulers. We maintain a resilient master copy of the resource allocations in the cluster, which we call *cell state*.[4] Each scheduler is given a private, local, frequently-updated copy of cell state that it uses for making scheduling decisions. The scheduler can see the entire state of the cell and has complete freedom to lay claim to any available cluster resources provided it has the appropriate permissions and priority – even ones that another scheduler has already acquired. Once a scheduler makes a placement decision, it updates the shared copy of cell state in an atomic commit. At most one such commit will succeed in the case of conflict: effectively, the time from state synchronization to the commit attempt is a *transaction*. Whether or not the transaction succeeds, the scheduler resyncs its local copy of cell state afterwards and, if necessary, re-runs its scheduling algorithm and tries again.

Omega schedulers operate completely in parallel and do not have to wait for jobs in other schedulers, and there is no inter-scheduler head of line blocking. To prevent conflicts from causing starvation, Omega schedulers typically choose to use incremental transactions, which accept all but the conflicting changes (i.e., the transaction provides atomicity but not independence). A scheduler can instead use an all-or-nothing transaction to achieve gang scheduling: either all tasks of a job are scheduled together, or none are, and the scheduler must try to schedule the entire job again. This helps to avoid resource hoarding, since a gang-scheduled job can preempt lower-priority tasks once sufficient resources are available and its transaction commits, and allow other schedulers' jobs to use the resources in the meantime.

Different Omega schedulers can implement different policies, but all must agree on what resource allocations are permitted (e.g., a common notion of whether a machine is full), and a common scale for expressing the relative importance of jobs, called *precedence*. These rules are deliberately kept to a minimum. The two-level scheme's centralized resource allocator component is thus simplified to a persistent data store with validation code that enforces these common rules. Since there is no central policy-enforcement engine for high-level cluster-wide goals, we rely on these showing up as emergent behaviors that result from the decisions of individual schedulers. In this, it helps that fairness is not a primary concern in our environment: we are driven more by the need to meet business requirements. In support of these, individual schedulers have configuration settings to limit the total amount of resources they may claim, and to limit the number of jobs they admit. Finally, we also rely on *post-facto* enforcement, since we are monitoring the system's behavior anyway.

| | *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 |
| $\lambda_{jobs}$ | 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) |

**Table 2:** Comparison of the two simulators; "actual data" refers to use of information found in a detailed workload-execution trace taken from a production cluster.

The performance viability of the shared-state approach is ultimately determined by the frequency at which transactions fail and the costs of such failures. The rest of this paper explores these issues for typical cluster workloads at Google.

## 4. Design comparisons

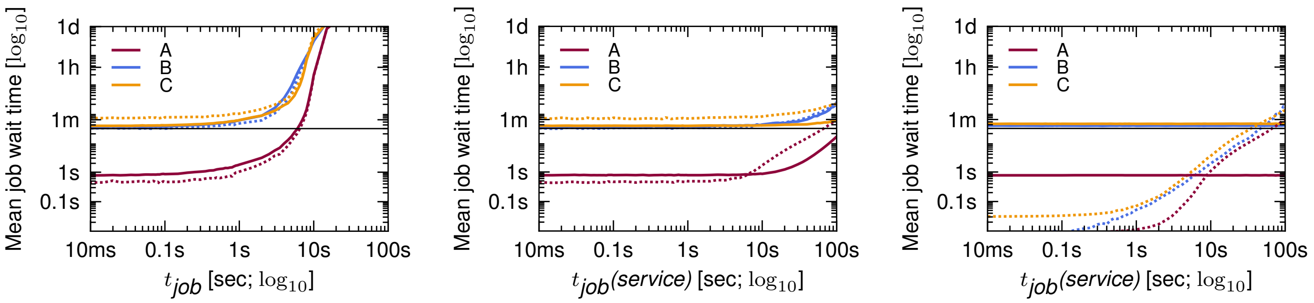
To understand the tradeoffs between the different approaches described before (monolithic, two-level and shared-state schedulers), we built two simulators:

1. A **lightweight simulator** driven by synthetic workloads using parameters drawn from empirical workload distributions. We use this to compare the behaviour of all three architectures under the same conditions and with identical workloads. By making some simplifications, this lightweight simulator allows us to sweep across a broad range of operating points within a reasonable runtime. The lightweight simulator also does not contain any proprietary Google code and is available as open source software.[5]

2. A **high-fidelity simulator** that replays historic workload traces from Google production clusters, and reuses much of the Google production scheduler's code. This gives

[4] A cell is the management unit for part of a physical cluster; a cluster may support more than one cell. Cells do not overlap.

[5] `https://code.google.com/p/cluster-scheduler-simulator/`.

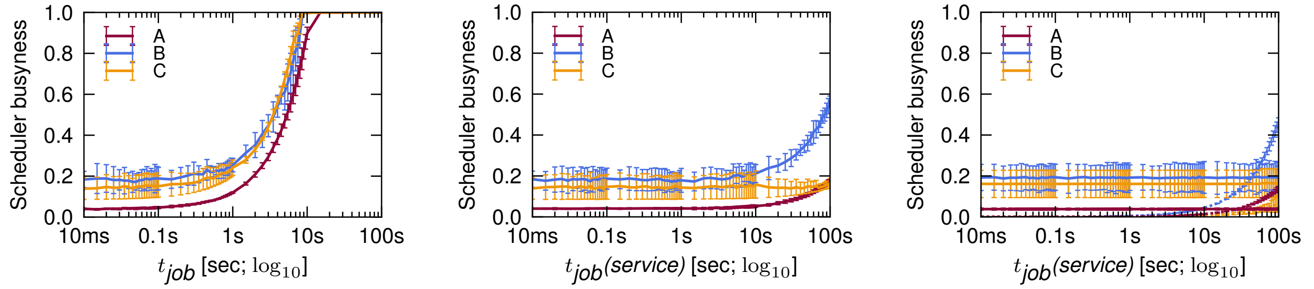


**(a)** Single-path.

**(b)** Multi-path.

**(c)** Shared state.

**Figure 5:** Schedulers' job wait time, 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 SLO (horizontal bar) is 30s.



**(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 ± median absolute deviation (MAD), i.e. the median deviation from the median value, a robust estimator of typical value dispersion.

us behavior closer to the real system, at the price of only supporting the Omega architecture and running a lot more slowly than the lightweight simulator: a single run can take days.

The rest of this section describes the simulators and our experimental setup.

**Simplifications in the lightweight simulator.** In the lightweight simulator, we trade speed and flexibility for accuracy by making some simplifying assumptions, summarized in Table 2.

The simulator is driven by a workload derived from from real workloads that ran on the same clusters and time periods discussed in §2.1. While the high-fidelity simulator is driven by the actual workload traces, for the lightweight simulator we analyze the workloads to obtain distributions of parameter values such as the number of tasks per job, the task duration, the per-task resources and job inter-arrival times, and then synthesize jobs and tasks that conform to these distributions.

At the start of a simulation, the lightweight simulator initializes cluster state using task-size data extracted from the relevant trace, but only instantiates sufficiently many tasks to utilize about 60% of cluster resources, which is comparable to the utilization level described in [24]. In production, Google speculatively over-commits resources, but the mechanisms and policies for this are too complicated to be replicated in the lightweight simulator.

The simulator can support multiple scheduler types, but initially we consider just two: *batch* and *service*. The two types of job have different parameter distributions, summarized in §2.1.

To improve simulation runtime in pathological situations, we limit any single job to 1,000 scheduling attempts, and the simulator abandons the job at this point if some tasks are still unscheduled. In practice, this only matters for the two-level scheduler (see §4.2), and is rarely triggered by the others.

**Parameters.** We model the scheduler decision time as a linear function of the form $t_{decision} = t_{job} + t_{task} \times$ *tasks per job*, where $t_{job}$ is a per-job overhead and $t_{task}$ represents the incremental cost to place each task. This turns out to be a reasonable approximation of Google's current cluster scheduling logic because most jobs in our real-life workloads have tasks with identical requirements [24]. Our values for $t_{job}$ and $t_{task}$ are based on somewhat conservative[6] estimates from measurements of our current production system's behavior: $t_{job} = 0.1\,\mathrm{s}$ and $t_{task} = 5\,\mathrm{ms}$.

Many of our experiments explore the effects of varying $t_{decision}(service)$ for the service scheduler because we are interested in exploring how Omega is affected by the longer decision times needed for sophisticated placement algorithms. We also vary the job arrival rate, $\lambda_{jobs}$, to model changes to the cluster workload level.

**Metrics.** Typically, users evaluate the perceived quality of cluster scheduling by considering the time until their jobs start running, as well as their runtime to completion. We refer to the former metric as *job wait time*, which we define as the difference between the job submission time and the beginning of the job's first scheduling attempt. Our schedulers process one request at a time, so a busy scheduler will cause enqueued jobs to be delayed. Job wait time thus measures the depth of scheduler queues, and will increase as the scheduler is busy for longer – either because it receives more jobs, or because they take longer to schedule. A common production service level objective (SLO) for job wait time is 30s.

Job wait time depends on the *scheduler busyness*: the fraction of time in which the scheduler is busy making scheduling decisions. It increases with the per-job decision time, and, in the shared-state approach, if scheduling work must be redone because of conflicts. To assess how much of the latter is occurring, we measure the *conflict fraction*, which denotes the average number of conflicts per successful transaction. A value of 0 means no conflicts took place; a value of 3 indicates that the average job experiences three conflicts, and thus requires four scheduling attempts.

Our values for scheduler busyness and conflict fraction are medians of the daily values, and wait time values are overall averages. Where present, error bars indicate how much variation exists across days in the experiment: they show the median absolute deviation (MAD) from the median value of the per-day averages. All experiments simulate seven days of cluster operation, except for the Mesos ones, which simulate only one day, as they take much longer to run because of the failed scheduling attempts that result from insufficient available resources (see §4.2).

### 4.1 Monolithic schedulers

Our baseline for comparison is a serial monolithic scheduler with the same decision time for batch and service jobs, to reflect the need to run much of the same code for every job type (a *single-path* implementation). We also consider a monolithic scheduler with a fast code path for batch jobs; we refer to this as a *multi-path* monolithic scheduler, since it still schedules only one job at a time. The current monolithic Google cluster scheduler is somewhere in between these pure designs: it does run some job-specific logic, but mostly applies identical scheduling logic for all jobs.

In the baseline case, we vary the scheduler decision time on the *x*-axis by changing $t_{job}$. In the multi-path case, we split the workload into batch and service workloads and use the defaults for the batch scheduler decision time while we vary $t_{job}(service)$.

The results are not surprising: in the single-path baseline case, the scheduler busyness is low as long as scheduling is quick, but scales linearly with increased $t_{job}$ (Figure 6a). As a consequence, job wait time increases at a similar rate until the scheduler is saturated, at which point it cannot keep up with the incoming workload any more. The wait time curves for service jobs closely track the ones for batch jobs, since all jobs take the same time to schedule (Figure 5a).

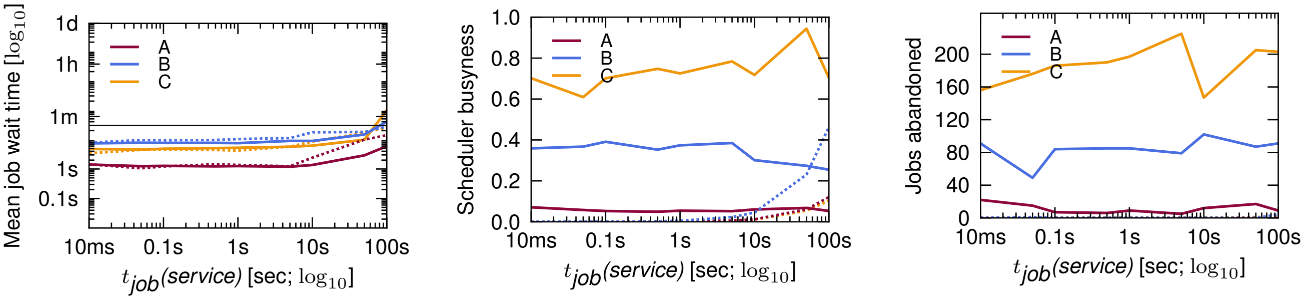
With a fast path for batch jobs in the multi-path case, both average job wait time and scheduler busyness decrease significantly even at long decision times for service jobs, since the majority of jobs are batch ones. But batch jobs can still get stuck in a queue behind the slow-to-schedule service jobs, and head-of-line blocking occurs: scalability is still limited by the processing capacity of a single scheduler (Figures 5b and 6b). To avoid this, we need some form of parallel processing.

### 4.2 Two-level scheduling: Mesos

Our two-level scheduler experiments are modeled on the offer-based Mesos design. We simulate a single resource manager and two scheduler frameworks, one handling batch jobs and one handling service jobs. To keep things simple, we assume that a scheduler only looks at the set of resources available to it when it begins a scheduling attempt for a job (i.e., any offers that arrive during the attempt are ignored). Resources not used at the end of scheduling a job are returned to the allocator; they may be re-offered again if the framework is the one furthest below its fair share. The DRF algorithm used by Mesos's centralized resource allocator is quite fast, so we assume it takes 1 ms to make a resource offer.

Since we now have two schedulers, we keep the decision time for the batch scheduler constant, and vary the decision time for the service scheduler by adjusting $t_{job}(service)$. However, the batch scheduler busyness (Figure 7b) turns out to be much higher than in the monolithic multi-path case. This is a consequence of an interaction between the Mesos offer model and the service scheduler's long scheduling decision times. Mesos achieves fairness by alternately offering *all* available cluster resources to different schedulers, predicated on assumptions that resources become available frequently and scheduler decisions are quick. As a result, a long scheduler decision time means that nearly all cluster resources are locked down for a long time, inaccessible to other schedulers. The only resources available for other schedulers in this situation are the few becoming available while the slow scheduler is busy. These are often insufficient

[6] In the sense that they are approximations least favorable to the Omega architecture.



**(a)** Job wait time.

**(b)** Scheduler busyness.

**(c)** Unscheduled jobs.

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

to schedule an above-average size batch job, meaning that the batch scheduler cannot make progress while the service scheduler holds an offer. It nonetheless keeps trying, and as a consequence, we find that a number of jobs are abandoned because they did not finish scheduling their tasks by the 1,000-attempt retry limit in the Mesos case (Figure 7c).

This pathology occurs because of Mesos's assumption of quick scheduling decisions, small jobs and high resource churn, which do not hold for our service jobs. Mesos could be extended to make only fair-share offers, although this would complicate the resource allocator logic, and the quality of the placement decisions for big or picky jobs would likely decrease, since each scheduler could only see a smaller fraction of the available resources. We have raised this point with the Mesos team; they agree about the limitation and are considering to address it in future work.

### 4.3 Shared-state scheduling: Omega

Finally, we use the lightweight simulator to explore the Omega shared-state approach. We again simulate two schedulers: one handling the batch workload, one handling the service workload. Both schedulers refresh their local copy of cell state by synchronizing it with the shared one when they start looking at a job, and work on their local copy for the duration of the decision time. Assuming at least one task got scheduled, a transaction to update the shared cell state is issued once finished. If there are no conflicts, then the entire transaction is accepted; otherwise only those changes that do not result in an overcommitted machine are accepted.

Figure 5c shows that the average job wait times for the Omega approach are comparable to those for multi-path monolithic (Figure 5b). This suggests that conflicts and interference are relatively rare, and this is confirmed by the graph of scheduler busyness (Figure 6c). Unlike Mesos (Figure 7c), the Omega-style scheduler manages to schedule all jobs in the workload. Unlike the monolithic multi-path implementation, it does not suffer from head-of-line blocking: the lines for batch and service jobs are independent.

We also investigate at how the Omega approach scales as the workload changes. For this purpose, we increase the job arrival rate of the batch scheduler, $\lambda_{jobs}$*(batch)*. Figure 8 shows that both job wait time and scheduler busyness increase. In the batch case, this is due to the higher job arrival rate, while in the service case, it is due to additional conflicts. As indicated by the dashed vertical lines, cluster A scales to about 2.5× the original workload before failing to keep up, while clusters B and C scale to 6× and 9.5×, respectively.

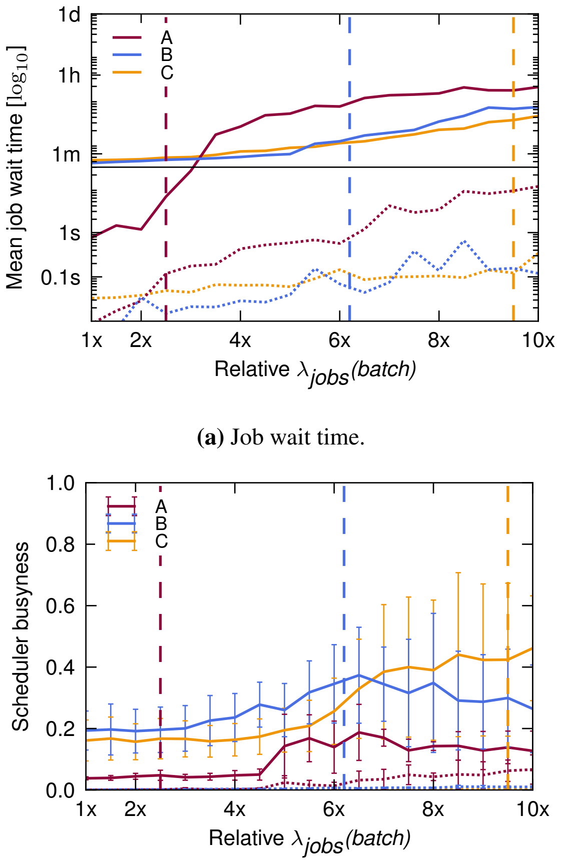
Since the batch scheduler is the main scalability bottleneck, we repeat the same scaling experiment with multiple batch schedulers in order to test the ability of the Omega model to scale to larger loads. The batch scheduling work is load-balanced across the schedulers using a simple hashing function. As expected, the conflict fraction increases with more schedulers as more opportunities for conflict exist (Figure 9a), but this is compensated – at least up to 32 batch schedulers – by the better per-scheduler busyness with more schedulers (Figure 9b). Similar results are seen with the job wait times (not shown here). This is an encouraging result: the Omega model can scale to a high batch workload while still providing good behavior for service jobs.

### 4.4 Summary

The lightweight simulator is a useful tool for comparing the different scheduler architectures. Figure 10 summarizes the results graphically, considering the impact of scaling $t_{task}$ as an additional dimension.

In short, the monolithic scheduler is not scalable. Although adding the multi-path feature reduces the average scheduling decision time, head-of-line blocking is still a problem for batch jobs, and means that this model may not be able to scale to the workloads we project for large clusters. The two-level model of Mesos can support independent scheduler implementations, but it is hampered by pessimistic locking, does not handle long decision times well, and could not schedule much of the heterogeneous load we offered it.

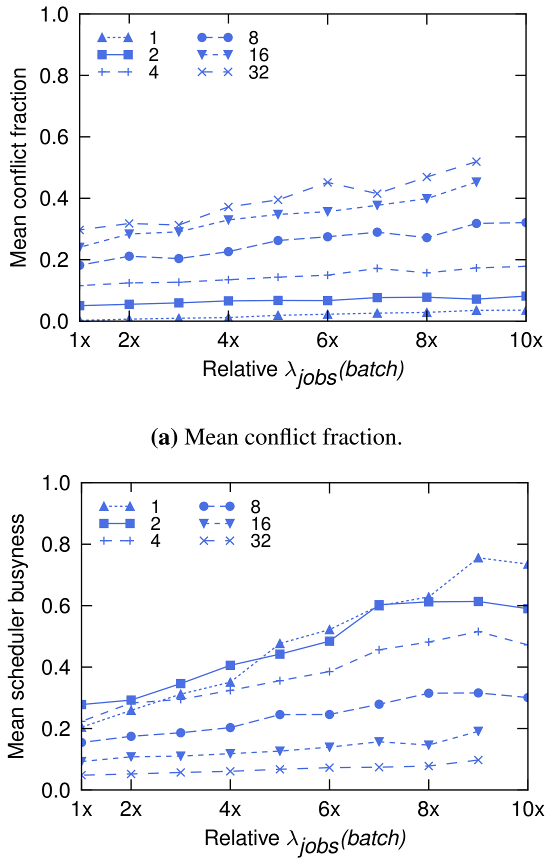
The shared-state Omega approach seems to offer competitive, scalable performance with little interference at realistic



**(a)** Job wait time.

**(b)** Scheduler busyness.

**Figure 8:** Shared-state scheduling (Omega): varying the arrival rate for the batch workload, $\lambda_{jobs}(batch)$, for cluster B. Dashed vertical lines indicate points of scheduler saturation; i.e., only partial scheduling of the workload to their right.



**(a)** Mean conflict fraction.

**(b)** Mean sched. busyness.

**Figure 9:** Shared-state scheduling (Omega): varying the arrival rate for the batch workload ($\lambda_{jobs}(batch)$) for cluster B; 1.0 is the default rate. Each line represents a different number of batch schedulers.

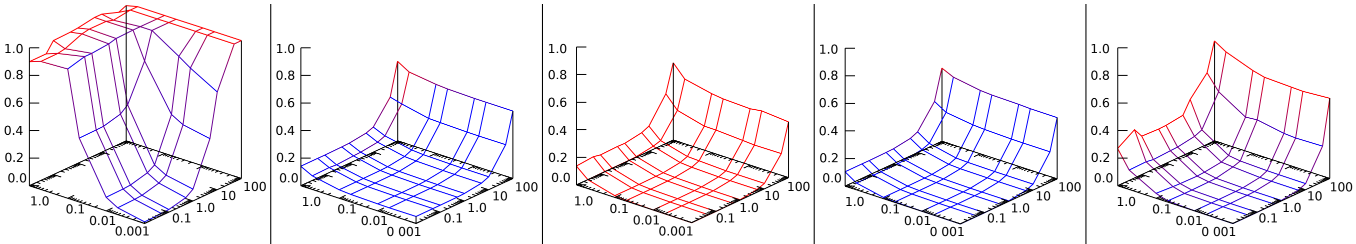
operating points, supports independent scheduler implementations, and exposes the entire allocation state to the schedulers. We show how this is helpful in §6. Our results indicate that Omega can scale to many schedulers, as well as to challenging workloads.

## 5. Trace-driven simulation

Having compared the different scheduler architectures using the lightweight simulator, we use the high-fidelity simulator to explore some of the properties of the Omega shared-state approach in greater detail and without the simplifying assumptions made by the lightweight simulator. The core of the high-fidelity simulator is the code used in Google's production scheduling system. It respects task placement constraints, uses the same algorithms as the production version, and can be given initial cell descriptions and detailed workload traces obtained from live production cells. It lets us evaluate the shared-state design with high confidence on real-world workloads. We use it to answer the following questions:

1. How much scheduling interference is present in real-world workloads and what scheduler decision times can we afford in production (§5.1)?
2. What are the effects of different conflict detection and resolution techniques on real workloads (§5.2)?
3. Can we take advantage of having access to the entire state of the cell in a scheduler? (§6)

Large-scale production systems are enormously complicated, and thus even the high-fidelity simulator employs a few simplifications. It does not model machine failures (as these only generate a small load on the scheduler); it does not model the disparity between resource requests and the actual usage of those resources in the traces (further discussed elsewhere [24]); it fixes the allocations at the initially-requested sizes (a consequence of limitations in the trace data); and it



**(a)** Monolithic scheduler, single-path.

**(b)** Monolithic scheduler, multi-path.

**(c)** Two-level scheduling (Mesos).

**(d)** Shared-state (Omega).

**(e)** Shared-state, coarse, gang scheduling.

**Figure 10:** Lightweight simulator: impact of varying $t_{job}$(service) (right axis) and $t_{task}$(service) (left axis) on scheduler busyness ($z$-axis) in different scheduling schemes, on cluster B. Red shading of a 3D graph means that part of the workload remained unscheduled.

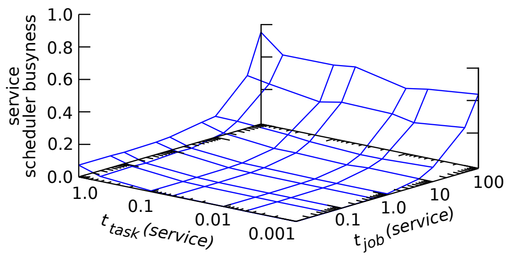


**Figure 11:** Shared-state scheduling (Omega): effect on service scheduler busyness of varying $t_{job}$*(service)* and $t_{task}$*(service)*, using the high-fidelity simulator and a 29-day trace from cluster C.

disables preemptions, because we found that they make little difference to the results, but significantly slow down the simulations.

As expected, the outputs of the two simulators generally agree. The main difference is that the lightweight simulator runs experience less interference, which is likely a result of the lightweight simulator's lack of support for placement constraints (which makes "picky" jobs seem easier to schedule than they are), and its simpler notion of when a machine is considered full (which means it sees fewer conflicts with fine-grained conflict detection, cf. §5.2).

We can nonetheless confirm all the trends the lightweight simulator demonstrates for the Omega shared-state model using the high-fidelity simulator. We believe this confirms that the lightweight simulator experiments provide plausible comparisons between different scheduling architectures under a common set of assumptions.

### 5.1 Scheduling performance

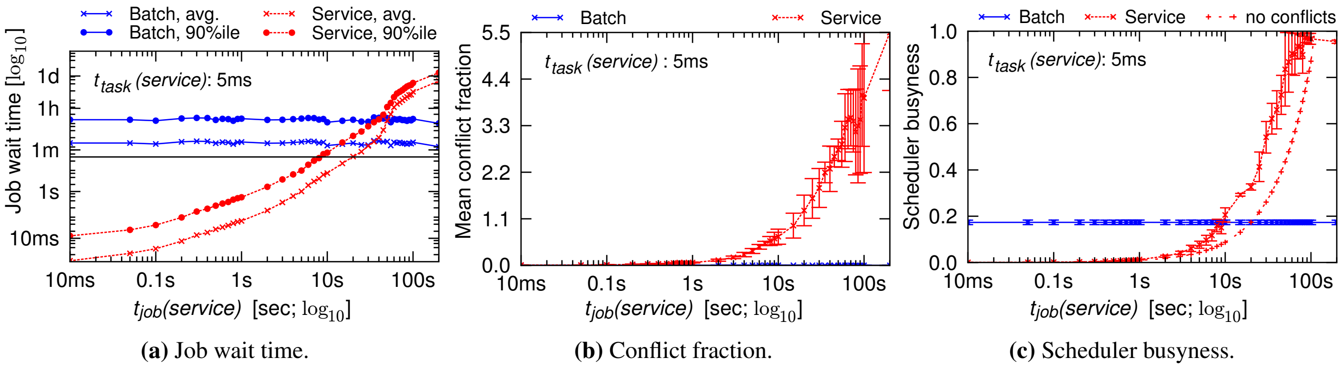
Figure 11 shows how service scheduler busyness varies as a function of both $t_{job}$*(service)* and $t_{task}$*(service)* for a month-long trace of cluster C (covering the same workload as the public trace). Encouragingly, the scheduler busyness remains low across almost the entire range for both, which means that the Omega architecture scales well to long decision times for service jobs.

**Scaling the workload.** We also investigate the performance of the shared-state architecture using a 7-day trace from cluster B, which is one of the largest and busiest Google clusters. Again, we vary $t_{job}$*(service)*. In Figure 12b, once $t_{job}$*(service)* reaches about 10s, the conflict fraction increases beyond 1.0, so that scheduling a service job requires at least one retry, on average.

At around the same point, we fail to meet the 30s job wait time SLO for the service scheduler (Figure 12a), even though the scheduler itself is not yet saturated: the additional wait time is purely due to the impact of conflicts. To confirm this, we approximate the time that the scheduler would have taken if it had experienced no conflicts or retries (the "no conflict" case in Figure 12c), and find that the service scheduler busyness with conflicts is about 40% higher than in the no-conflict case. This is a higher level of interference compared to cluster C, most likely because of a much higher batch load in cluster B.

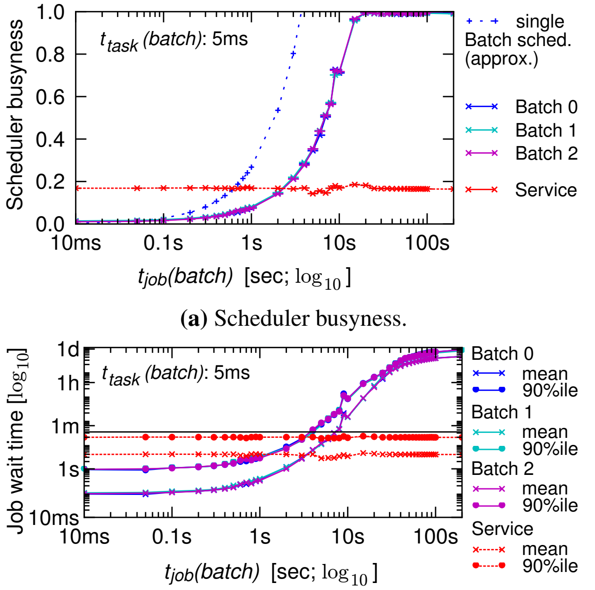
Despite these relatively high conflict rates, our experiments show that the shared-state Omega architecture can support service schedulers that take several seconds to make a decision. We also investigate scaling the per-task decision time, and found that we can support $t_{task}$*(service)* of 1 second (at a $t_{job}$*(service)* of 0.1s), resulting in a conflict fraction $\leq$ 0.2. This means that we can support schedulers with a high one-off per-job decision time, and ones with a large per-task decision time.

**Load-balancing the batch scheduler.** With the monolithic single-path scheduler (§4.1), the high batch job arrival rate requires the use of basic, simple scheduling algorithms: it simply is not possible to use smarter, more time-consuming scheduling algorithms for these jobs as we already miss the SLO on cluster B due to the high load. Batch jobs want to survive failures, too, and the placement quality would doubtless improve if a scheduler could be given a little more time to make a decision. Fortunately, the Omega archi-



**(a)** Job wait time. **(b)** Conflict fraction. **(c)** Scheduler busyness.

**Figure 12:** Shared-state scheduling (Omega): performance effects of varying $t_{job}$*(service)* on a 7-day trace from cluster B.



**(a)** Scheduler busyness.

**(b)** Job wait time.

**Figure 13:** Shared-state scheduling (Omega): performance effects of splitting the batch workload across 3 batch schedulers, varying $t_{job}$*(batch)* in a 24h trace from cluster C.

tecture can easily achieve this by load-balancing the scheduling of batch jobs across multiple batch schedulers.

To test this, we run an experiment with three parallel batch schedulers, partitioning the workload across them by hashing the job identifiers, akin to the earlier experiment with the simple simulator. We achieve an increase in scalability of $\simeq 3\times$, moving the saturation point from $t_{job}$*(batch)* of about 4s to 15s (Figure 13a). At the same time, the conflict rate remains low (around 0.1), and all schedulers meet the 30s job wait time SLO until the saturation point (Figure 13b).

In short, load-balancing across multiple schedulers can increase scalability to increasing job arrival rates. Of course, the scale-up must be sub-linear due to of the overhead of maintaining and updating the local copies of cell state, and this approach will not easily handle hundreds of schedulers. Our comparison point, however, is a single monolithic scheduler, so even a single-digit speedup is helpful.

In summary, the Omega architecture scales well, and tolerates large decision times on real cluster workloads.

### 5.2 Dealing with conflicts

We also use the high-fidelity simulator to explore two implementation choices we were considering for Omega.

In the first, *coarse-grained conflict detection*, a scheduler's placement choice would be rejected if *any* changes had been made to the target machine since the local copy of cell state was synchronized at the beginning of the transaction. This can be implemented with a simple sequence number in the machine's state object.

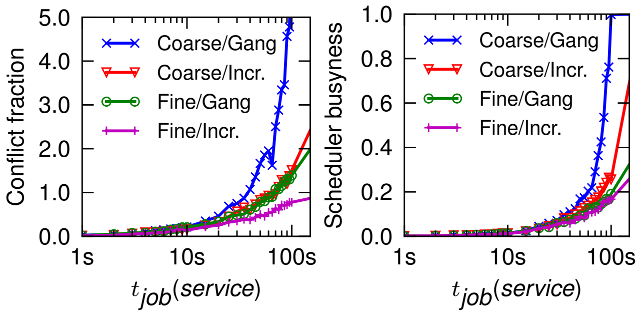
In the second, *all-or-nothing scheduling*, an entire cell state transaction would be rejected if it would cause *any* machine to be over-committed. The goal here was to support jobs that require gang scheduling, or that cannot perform any useful work until all their tasks are running.[7]

Not surprisingly, both alternatives lead to additional conflicts and higher scheduler busyness (Figure 14). While turning on all-or-nothing scheduling for all jobs only leads to a minor increase in scheduler busyness when using fine-grained conflict detection (Figure 14a), it does increase conflict fraction by about 2× as retries now must re-place all tasks, increasing their chance of failing again (Figure 14a). Thus, this option should only be used on a job-level granularity. Relying on coarse-grained conflict detection makes things even worse: spurious conflicts lead to increases in conflict rate, and consequently scheduler busyness, by 2–3×. Clearly, incremental transactions should be the default.

## 6. Flexibility: a MapReduce scheduler

Finally, we explore how well we can meet two additional design goals of the Omega shared-state model: supporting spe-

[7] This is supported by Google's current scheduler, but it is only rarely used due to the expectation of machine failures, which disrupt jobs anyway.



**(a)** Conflict fraction. **(b)** Scheduler busyness.

**Figure 14:** Shared-state scheduling (Omega): effect of gang scheduling and coarse-grained conflict detection as a function of $t_{job}(service)$ (cluster C, 29 days); mean daily values.

cialized schedulers, and broadening the kinds of decisions that schedulers can perform compared to the two-level approach. This is somewhat challenging to evaluate quantitatively, so we proceed by way of a case study that adds a specialized scheduler for MapReduce jobs.

Cluster users at Google currently specify the number of workers for a MapReduce job and their resource requirements at job submission time, and the MapReduce framework schedules map and reduce activities[8] onto these workers. Because the available resources vary over time and between clusters, most users pick the number of workers based on a combination of intuition, trial-and-error and experience: data from a month's worth of MapReduce jobs run at Google showed that frequently observed values were 5, 11, 200 and 1,000 workers.

What if the number of workers could be chosen automatically if additional resources were available, so that jobs could complete sooner? Our specialized MapReduce scheduler does just this by opportunistically using idle cluster resources to speed up MapReduce jobs. It observes the overall resource utilization in the cluster, predicts the benefits of scaling up current and pending MapReduce jobs, and apportions some fraction of the unused resources across those jobs according to some policy.

MapReduce jobs are particularly well-suited to this approach because it is possible to build reasonably accurate models of how a job's resource allocation affects its running time [12, 26]. About 20% of jobs in Google are MapReduce ones, and many of them are run repeatedly, so historical data is available to build models. Many of the jobs are low-priority, "best effort" computations that have to make way for higher-priority service jobs, and so may benefit from exploiting spare resources in the meantime [3].

### 6.1 Implementation

Since our goal is to investigate scheduler flexibility rather than demonstrate accurate MapReduce modelling, we deliberately use a simple performance model that only relies on historical data about the job's average map and reduce activity duration. It assumes that adding more workers results in an idealized linear speedup (modulo dependencies between mappers and reducers), up to the point where map activities and all reduce activities respectively run in parallel. Since large MapReduce jobs typically have many more of these activities than configured workers, we usually run out of available resources before this point.

We consider three different policies for adding resources: *max-parallelism*, which keeps on adding workers as long as benefit is obtained, *global cap*, which stops the MapReduce scheduler using idle resources if the total cluster utilization is above a target value, and *relative job size*, which limits the maximum number of workers to four times as many as it initially requested. In each case, a set of resource allocations to be investigated is run through the predictive model, and the allocation leading to the earliest possible finish time is used. More elaborate approaches and objective functions, such as used in deadline-based schedulering [10], are certainly possible, but not the focus of this case study.

### 6.2 Evaluation

We evaluate the three different resource-allocation policies using traces from clusters A and C, plus cluster D, which is a small, lightly-loaded cluster that is about a quarter of the size of cluster C. Our results suggest that 50–70% of MapReduce jobs can benefit from acceleration using opportunistic resources (Figure 15). The huge speedups seen in the tail should be taken with a pinch of salt due to our simple linear speedup model, but we have more confidence in the values for the 80$^{th}$ percentile, and here, our simulations predict a speedup of 3–4× using the eager max-parallelism policy.

Although the max-parallelism policy produces the largest improvements, the relative job size policy also does quite well, and its speedups probably have a higher likelihood of being achieved because it requires fewer new MapReduce workers to be constructed: the time to set up a worker on a new machine is not fully accounted for in the simple model. The global cap policy performs almost as well as max-parallelism in the small, under-utilized cluster D, but achieves little or no benefit elsewhere, since the cluster utilization is usually above the threshold, which was set at 60%.

Adding resources to a MapReduce job will cause the cluster's resource utilization to increase, and should result in the job completing sooner, at which point all of the job's resources will free up. An effect of this is an increase in the variability of the cluster's resource utilization (Figure 16).

To do its work, the MapReduce scheduler relies on being able to see the entire cluster's state, which is straightforward in the Omega architecture. A similar argument can be made

[8] These are typically called "tasks" in literature, but we have renamed them to avoid confusion with the cluster-scheduler level tasks that substantiate MapReduce "workers".

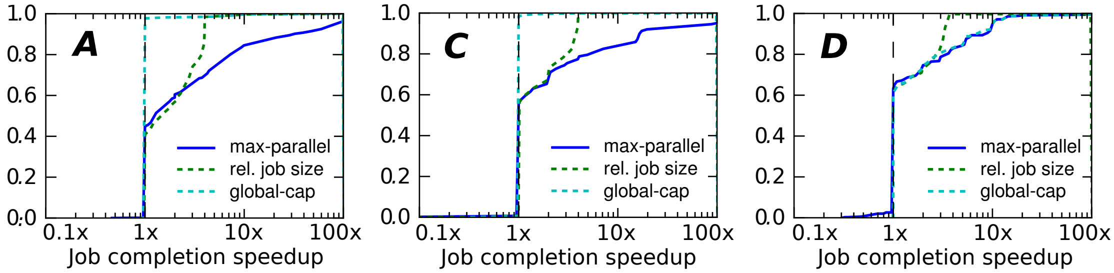


**Figure 15:** CDF of potential per-job speedups using different policies on clusters A, C and D (a small, lightly-utilized cluster).

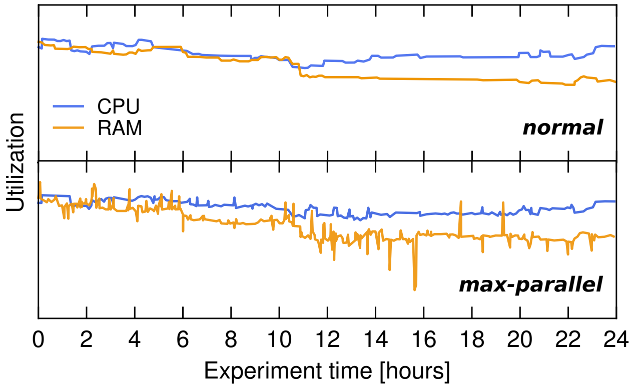


**Figure 16:** Time series of normalized cluster utilization on cluster C without the specialized Omega MapReduce scheduler (top), and in max-parallelism mode (bottom).

for a specialized service scheduler for highly-constrained, high-priority jobs. Scheduling them requires determining which machines are applicable, and deciding how best to place the new job while minimizing the number of preemptions caused to lower-priority jobs. The shared-state model is ideally suited to this. Our prototype MapReduce scheduler demonstrates that adding a specialized functionality to the Omega system is straightforward (unlike with our current production scheduler).

## 7. Additional related work

Large-scale cluster resource scheduling is not a novel challenge. Many researchers have considered this problem before, and different solutions have been proposed in the HPC, middleware and "cloud" communities. We discussed several examples in §3, and further discussed the relative merits of these approaches in §4.

The Omega approach builds on many prior ideas. Scheduling using shared state is an example of optimistic concurrency control, which has been explored by the database community for a long time [18], and, more recently, considered for general memory access in the transactional memory community [2].

Exposing the entire cluster state to each scheduler is not unlike the Exokernel approach of removing abstractions and exposing maximal information to applications [9]. The programming language and OS communities have recently revisited application level scheduling as an alternative to general-purpose thread and process schedulers, arguing that a single, global OS scheduler is neither scalable, nor flexible enough for modern multi-core applications' demands [22].

Amoeba [3] implements opportunistic allocation of spare resources to jobs, with motivation similar to our MapReduce scheduler use-case. However, it achieves this by complex communication between resource and application managers, whereas Omega naturally lends itself to such designs as it exposes the entire cluster state to all schedulers.

## 8. Conclusions and future work

This investigation is part of a wider effort to build Omega, Google's next-generation cluster management platform. Here, we specifically focused on a cluster scheduling architecture that uses parallelism, shared state, and optimistic concurrency control. Our performance evaluation of the Omega model using both lightweight simulations with synthetic workloads, and high-fidelity, trace-based simulations of production workloads at Google, shows that optimistic concurrency over shared state is a viable, attractive approach to cluster scheduling.

Although this approach will do strictly more work than a pessimistic locking scheme as work may need to be re-done, we found the overhead to be acceptable at reasonable operating points, and the resulting benefits in eliminating head-of-line blocking and better scalability to often outweigh it. We also found that Omega's approach offers an attractive platform for development of specialized schedulers, and illustrated its flexibility by adding a MapReduce scheduler with opportunistic resource adjustment.

Future work could usefully focus on ways to provide global guarantees (fairness, starvation avoidance, etc.) in the Omega model: this is an area where centralized control makes life easier. Furthermore, we believe there are some techniques from the database community that could be applied to reduce the likelihood and effects of interference for schedulers with long decision times. We hope to explore some of these in the future.

## Acknowledgements

Many people contributed to the work described in this paper. Members of the Omega team at Google who contributed to this design include Brian Grant, David Oppenheimer, Jason Hickey, Jutta Degener, Rune Dahl, Todd Wang and Walfredo Cirne. We would like to thank the Mesos team at UC Berkeley for many fruitful and interesting discussions about Mesos, and Joseph Hellerstein for his early work on modeling scheduler interference in Omega. Derek Murray, Steven Hand and Alexey Tumanov provided valuable feedback on draft versions of this paper. The final version was much improved by comments from the anonymous reviewers.

## References

[1] Adaptive Computing Enterprises Inc. *Maui Scheduler Administrator's Guide*, 3.2 ed. Provo, UT, 2011.

[2] Adl-Tabatabai, A.-R., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., and Shpeisman, T. Compiler and runtime support for efficient software transactional memory. In *Proceedings of PLDI* (2006), pp. 26–37.

[3] Ananthanarayanan, G., Douglas, C., Ramakrishnan, R., Rao, S., and Stoica, I. True elasticity in multi-tenant data-intensive compute clusters. In *Proceedings of SoCC* (2012), p. 24.

[4] Apache. Hadoop On Demand. `http://goo.gl/px8Yd`, 2007. Accessed 20/06/2012.

[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems 26*, 2 (June 2008), 4:1–4:26.

[6] Chen, Y., Alspaugh, S., Borthakur, D., and Katz, R. Energy efficiency for large-scale MapReduce workloads with significant interactive analysis. In *Proceedings of EuroSys* (2012).

[7] Chen, Y., Ganapathi, A. S., Griffith, R., and Katz, R. H. Design insights for MapReduce from diverse production workloads. Tech. Rep. UCB/EECS–2012–17, UC Berkeley, Jan. 2012.

[8] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *CACM 51*, 1 (2008), 107–113.

[9] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of SOSP* (1995), pp. 251–266.

[10] Ferguson, A. D., Bodik, P., Kandula, S., Boutin, E., and Fonseca, R. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of EuroSys* (2012), pp. 99–112.

[11] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of NSDI* (2011), pp. 323–336.

[12] Herodotou, H., Dong, F., and Babu, S. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of SoCC* (2011).

[13] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A., Katz, R., Shenker, S., and Stoica, I. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI* (2011).

[14] Iqbal, S., Gupta, R., and Fang, Y.-C. Planning considerations for job scheduling in HPC clusters. *Dell Power Solutions* (Feb. 2005).

[15] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of SOSP* (2009).

[16] Jackson, D. and Snell, Q. and Clement, M. Core algorithms of the Maui scheduler. In *Job Scheduling Strategies for Parallel Processing*. 2001, pp. 87–102.

[17] Kavulya, S., Tan, J., Gandhi, R., and Narasimhan, P. An analysis of traces from a production MapReduce cluster. In *Proceedings of CCGrid* (2010), pp. 94–103.

[18] Kung, H. T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6*, 2 (June 1981), 213–226.

[19] Malewicz, G., Austern, M., Bik, A., Dehnert, J., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *Proceedings of SIGMOD* (2010), pp. 135–146.

[20] Mishra, A. K., Hellerstein, J. L., Cirne, W., and Das, C. R. Towards characterizing cloud backend workloads: insights from Google compute clusters. *SIGMETRICS Performance Evaluation Review 37* (Mar. 2010), 34–41.

[21] Murthy, A. C., Douglas, C., Konar, M., O'Malley, O., Radia, S., Agarwal, S., and K V, V. Architecture of next generation Apache Hadoop MapReduce framework. Tech. rep., Apache Hadoop, 2011.

[22] Pan, H., Hindman, B., and Asanović, K. Lithe: enabling efficient composition of parallel libraries. In *Proceedings of HotPar* (2009).

[23] Peng, D., and Dabek, F. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of OSDI* (2010).

[24] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SoCC* (2012).

[25] Sharma, B., Chudnovsky, V., Hellerstein, J., Rifaat, R., and Das, C. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proceedings of SoCC* (2011).

[26] Verma, A., Cherkasova, L., and Campbell, R. SLO-driven right-sizing and resource provisioning of MapReduce jobs. In *Proceedings of LADIS* (2011).

[27] Wilkes, J. More Google cluster data. Google research blog, Nov. 2011. Posted at `http://goo.gl/9B7PA`.

[28] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys* (2010), pp. 265–278.

[29] Zhang, Q., Hellerstein, J., and Boutaba, R. Characterizing task usage shapes in Google's compute clusters. In *Proceedings of LADIS* (2011).