Large-scale cluster management at Google with Borg

Abhishek Verma[†] Luis Pedrosa[‡] Madhukar Korupolu
David Oppenheimer Eric Tune John Wilkes

Google Inc.

Abstract

Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.

It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.

1. Introduction

The cluster management system we internally call Borg admits, schedules, starts, restarts, and monitors the full range of applications that Google runs. This paper explains how.

Borg provides three main benefits: it (1) hides the details of resource management and failure handling so its users can focus on application development instead; (2) operates with very high reliability and availability, and supports applications that do the same; and (3) lets us run workloads across tens of thousands of machines effectively. Borg is not the first system to address these issues, but it's one of the few operating at this scale, with this degree of resiliency and completeness. This paper is organized around these topics, concluding with a set of qualitative observations we have made from operating Borg in production for more than a decade.

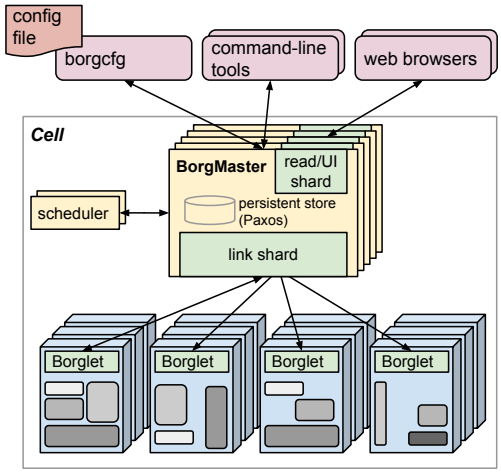


Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*

2. The user perspective

Borg's users are Google developers and system administrators (site reliability engineers or SREs) that run Google's applications and services. Users submit their work to Borg in the form of *jobs*, each of which consists of one or more *tasks* that all run the same program (binary). Each job runs in one Borg *cell*, a set of machines that are managed as a unit. The remainder of this section describes the main features exposed in the user view of Borg.

2.1 The workload

Borg cells run a heterogenous workload with two main parts. The first is long-running services that should "never" go down, and handle short-lived latency-sensitive requests (a few μs to a few hundred ms). Such services are used for end-user-facing products such as Gmail, Google Docs, and web search, and for internal infrastructure services (e.g., BigTable). The second is batch jobs that take from a few seconds to a few days to complete; these are much less sensitive to short-term performance fluctuations. The workload mix varies across cells, which run different mixes of applications depending on their major tenants (e.g., some cells are quite batch-intensive), and also varies over time: batch jobs

[†] Work done while author was at Google.

[‡] Currently at University of Southern California.

EuroSys'15, April 21–24, 2015, Bordeaux, France.

ACM 978-1-4503-3238-5/15/04.
http://dx.doi.org/10.1145/2741948.2741964

come and go, and many end-user-facing service jobs see a diurnal usage pattern. Borg is required to handle all these cases equally well.

A representative Borg workload can be found in a publicly-available month-long trace from May 2011 [80], which has been extensively analyzed (e.g., [68] and [1, 26, 27, 57]).

Many application frameworks have been built on top of Borg over the last few years, including our internal MapReduce system [23], FlumeJava [18], Millwheel [3], and Pregel [59]. Most of these have a controller that submits a master job and one or more worker jobs; the first two play a similar role to YARN's application manager [76]. Our distributed storage systems such as GFS [34] and its successor CFS, Bigtable [19], and Megastore [8] all run on Borg.

For this paper, we classify higher-priority Borg jobs as "production" (prod) ones, and the rest as "non-production" (non-prod). Most long-running server jobs are prod; most batch jobs are non-prod. In a representative cell, prod jobs are allocated about 70% of the total CPU resources and represent about 60% of the total CPU usage; they are allocated about 55% of the total memory and represent about 85% of the total memory usage. The discrepancies between allocation and usage will prove important in §5.5.

2.2 Clusters and cells

The machines in a cell belong to a single *cluster*, defined by the high-performance datacenter-scale network fabric that connects them. A cluster lives inside a single datacenter building, and a collection of buildings makes up a site.[1] A cluster usually hosts one large cell and may have a few smaller-scale test or special-purpose cells. We assiduously avoid any single point of failure.

Our median cell size is about 10 k machines after excluding test cells; some are much larger. The machines in a cell are heterogeneous in many dimensions: sizes (CPU, RAM, disk, network), processor type, performance, and capabilities such as an external IP address or flash storage. Borg isolates users from most of these differences by determining where in a cell to run tasks, allocating their resources, installing their programs and other dependencies, monitoring their health, and restarting them if they fail.

2.3 Jobs and tasks

A Borg job's properties include its name, owner, and the number of tasks it has. Jobs can have *constraints* to force its tasks to run on machines with particular *attributes* such as processor architecture, OS version, or an external IP address. Constraints can be hard or soft; the latter act like preferences rather than requirements. The start of a job can be deferred until a prior one finishes. A job runs in just one cell.

Each task maps to a set of Linux processes running in a container on a machine [62]. The vast majority of the Borg workload does not run inside virtual machines (VMs), because we don't want to pay the cost of virtualization. Also, the system was designed at a time when we had a considerable investment in processors with no virtualization support in hardware.

A task has properties too, such as its resource requirements and the task's index within the job. Most task properties are the same across all tasks in a job, but can be overridden – e.g., to provide task-specific command-line flags. Each resource dimension (CPU cores, RAM, disk space, disk access rate, TCP ports,[2] etc.) is specified independently at fine granularity; we don't impose fixed-sized buckets or slots (§5.4). Borg programs are statically linked to reduce dependencies on their runtime environment, and structured as *packages* of binaries and data files, whose installation is orchestrated by Borg.

Users operate on jobs by issuing remote procedure calls (RPCs) to Borg, most commonly from a command-line tool, other Borg jobs, or our monitoring systems (§2.6). Most job descriptions are written in the declarative configuration language BCL. This is a variant of GCL [12], which generates protobuf files [67], extended with some Borg-specific keywords. GCL provides lambda functions to allow calculations, and these are used by applications to adjust their configurations to their environment; tens of thousands of BCL files are over 1 k lines long, and we have accumulated tens of millions of lines of BCL. Borg job configurations have similarities to Aurora configuration files [6].

Figure 2 illustrates the states that jobs and tasks go through during their lifetime.

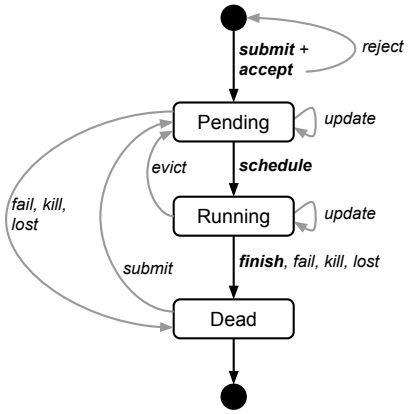


Figure 2: The state diagram for both jobs and tasks. *Users can trigger* submit, kill, *and* update *transitions.*

A user can change the properties of some or all of the tasks in a running job by pushing a new job configuration to Borg, and then instructing Borg to *update* the tasks to the new specification. This acts as a lightweight, non-atomic transaction that can easily be undone until it is closed (committed). Updates are generally done in a rolling fashion, and a limit can be imposed on the number of task disruptions

[1] There are a few exceptions for each of these relationships.

[2] Borg manages the available ports on a machine and allocates them to tasks.

(reschedules or preemptions) an update causes; any changes that would cause more disruptions are skipped.

Some task updates (e.g., pushing a new binary) will always require the task to be restarted; some (e.g., increasing resource requirements or changing constraints) might make the task no longer fit on the machine, and cause it to be stopped and rescheduled; and some (e.g., changing priority) can always be done without restarting or moving the task.

Tasks can ask to be notified via a Unix SIGTERM signal before they are preempted by a SIGKILL, so they have time to clean up, save state, finish any currently-executing requests, and decline new ones. The actual notice may be less if the preemptor sets a delay bound. In practice, a notice is delivered about 80% of the time.

2.4 Allocs

A Borg *alloc* (short for allocation) is a reserved set of resources on a machine in which one or more tasks can be run; the resources remain assigned whether or not they are used. Allocs can be used to set resources aside for future tasks, to retain resources between stopping a task and starting it again, and to gather tasks from different jobs onto the same machine – e.g., a web server instance and an associated logsaver task that copies the server's URL logs from the local disk to a distributed file system. The resources of an alloc are treated in a similar way to the resources of a machine; multiple tasks running inside one share its resources. If an alloc must be relocated to another machine, its tasks are rescheduled with it.

An *alloc set* is like a job: it is a group of allocs that reserve resources on multiple machines. Once an alloc set has been created, one or more jobs can be submitted to run in it. For brevity, we will generally use "task" to refer to an alloc or a top-level task (one outside an alloc) and "job" to refer to a job or alloc set.

2.5 Priority, quota, and admission control

What happens when more work shows up than can be accommodated? Our solutions for this are priority and quota.

Every job has a *priority*, a small positive integer. A high-priority task can obtain resources at the expense of a lower-priority one, even if that involves preempting (killing) the latter. Borg defines non-overlapping *priority bands* for different uses, including (in decreasing-priority order): monitoring, production, batch, and best effort (also known as testing or free). For this paper, prod jobs are the ones in the monitoring and production bands.

Although a preempted task will often be rescheduled elsewhere in the cell, preemption cascades could occur if a high-priority task bumped out a slightly lower-priority one, which bumped out another slightly-lower priority task, and so on. To eliminate most of this, we disallow tasks in the production priority band to preempt one another. Fine-grained priorities are still useful in other circumstances – e.g., MapReduce master tasks run at a slightly higher priority than the workers they control, to improve their reliability.

Priority expresses relative importance for jobs that are running or waiting to run in a cell. *Quota* is used to decide which jobs to *admit* for scheduling. Quota is expressed as a vector of resource quantities (CPU, RAM, disk, etc.) at a given priority, for a period of time (typically months). The quantities specify the maximum amount of resources that a user's job requests can ask for at a time (e.g., "20 TiB of RAM at prod priority from now until the end of July in cell xx"). [illegible]: jobs with insufficient quota are immediately rejected upon submission.

Higher-priority quota costs more than quota at lower-priority. Production-priority quota is limited to the actual resources available in the cell, so that a user who submits a production-priority job that fits in their quota can expect it to run, modulo fragmentation and constraints. Even though we encourage users to purchase no more quota than they need, many users overbuy because it insulates them against future shortages when their application's user base grows. We respond to this by over-selling quota at lower-priority levels: every user has infinite quota at priority zero, although this is frequently hard to exercise because resources are oversubscribed. A low-priority job may be admitted but remain pending (unscheduled) due to insufficient resources.

Quota allocation is handled outside of Borg, and is intimately tied to our physical capacity planning, whose results are reflected in the price and availability of quota in different datacenters. User jobs are admitted only if they have sufficient quota at the required priority. The use of quota reduces the need for policies like Dominant Resource Fairness (DRF) [29, 35, 36, 66].

Borg has a capability system that gives special privileges to some users; for example, allowing administrators to delete or modify any job in the cell, or allowing a user to access restricted kernel features or Borg behaviors such as disabling resource estimation (§5.5) on their jobs.

2.6 Naming and monitoring

It's not enough to create and place tasks: a service's clients and other systems need to be able to find them, even after they are relocated to a new machine. To enable this, Borg creates a stable "Borg name service" (BNS) name for each task that includes the cell name, job name, and task number. Borg writes the task's hostname and port into a consistent, highly-available file in Chubby [14] with this name, which is used by our RPC system to find the task endpoint. The BNS name also forms the basis of the task's DNS name, so the fiftieth task in job jfoo owned by user ubar in cell cc would be reachable via 50.jfoo.ubar.cc.borg.google.com. Borg also writes job size and task health information into Chubby whenever it changes, so load balancers can see where to route requests to.

Almost every task run under Borg contains a built-in HTTP server that publishes information about the health of the task and thousands of performance metrics (e.g., RPC latencies). Borg monitors the health-check URL and restarts tasks that do not respond promptly or return an HTTP error code. Other data is tracked by monitoring tools for dashboards and alerts on service level objective (SLO) violations.

A service called Sigma provides a web-based user interface (UI) through which a user can examine the state of all their jobs, a particular cell, or drill down to individual jobs and tasks to examine their resource behavior, detailed logs, execution history, and eventual fate. Our applications generate voluminous logs; these are automatically rotated to avoid running out of disk space, and preserved for a while after the task's exit to assist with debugging. If a job is not running Borg provides a "why pending?" annotation, together with guidance on how to modify the job's resource requests to better fit the cell. We publish guidelines for "conforming" resource shapes that are likely to schedule easily.

Borg records all job submissions and task events, as well as detailed per-task resource usage information in Infrastore, a scalable read-only data store with an interactive SQL-like interface via Dremel [61]. This data is used for usage-based charging, debugging job and system failures, and long-term capacity planning. It also provided the data for the Google cluster workload trace [80].

All of these features help users to understand and debug the behavior of Borg and their jobs, and help our SREs manage a few tens of thousands of machines per person.

3. Borg architecture

A Borg cell consists of a set of machines, a logically centralized controller called the Borgmaster, and an agent process called the Borglet that runs on each machine in a cell (see Figure 1). All components of Borg are written in C++.

3.1 Borgmaster

Each cell's Borgmaster consists of two processes: the main Borgmaster process and a separate scheduler (§3.2). The main Borgmaster process handles client RPCs that either mutate state (e.g., `create job`) or provide read-only access to data (e.g., `lookup job`). It also manages state machines for all of the objects in the system (machines, tasks, allocs, etc.), communicates with the Borglets, and offers a web UI as a backup to Sigma.

The Borgmaster is logically a single process but is actually replicated five times. Each replica maintains an in-memory copy of most of the state of the cell, and this state is also recorded in a highly-available, distributed, Paxos-based store [55] on the replicas' local disks. A single *elected master* per cell serves both as the Paxos leader and the state mutator, handling all operations that change the cell's state, such as submitting a job or terminating a task on a machine. A master is elected (using Paxos) when the cell is brought up and whenever the elected master fails; it acquires a Chubby lock so other systems can find it. Electing a master and failing-over to the new one typically takes about 10 s, but can take up to a minute in a big cell because some in-memory state has to be reconstructed. When a replica recovers from an outage, it dynamically re-synchronizes its state from other Paxos replicas that are up-to-date.

The Borgmaster's state at a point in time is called a *checkpoint*, and takes the form of a periodic snapshot plus a change log kept in the Paxos store. Checkpoints have many uses, including restoring a Borgmaster's state to an arbitrary point in the past (e.g., just before accepting a request that triggered a software defect in Borg so it can be debugged); fixing it by hand *in extremis*; building a persistent log of events for future queries; and offline simulations.

A high-fidelity Borgmaster simulator called Fauxmaster can be used to read checkpoint files, and contains a complete copy of the production Borgmaster code, with stubbed-out interfaces to the Borglets. It accepts RPCs to make state machine changes and perform operations, such as "schedule all pending tasks", and we use it to debug failures, by interacting with it as if it were a live Borgmaster, with simulated Borglets replaying real interactions from the checkpoint file. A user can step through and observe the changes to the system state that actually occurred in the past. Fauxmaster is also useful for capacity planning ("how many new jobs of this type would fit?"), as well as sanity checks before making a change to a cell's configuration ("will this change evict any important jobs?").

3.2 Scheduling

When a job is submitted, the Borgmaster records it persistently in the Paxos store and adds the job's tasks to the *pending* queue. This is scanned asynchronously by the *scheduler*, which assigns tasks to machines if there are sufficient available resources that meet the job's constraints. (The scheduler primarily operates on tasks, not jobs.) The scan proceeds from high to low priority, modulated by a round-robin scheme within a priority to ensure fairness across users and avoid head-of-line blocking behind a large job. The scheduling algorithm has two parts: *feasibility checking*, to find machines on which the task could run, and *scoring*, which picks one of the feasible machines.

In feasibility checking, the scheduler finds a set of machines that meet the task's constraints and also have enough "available" resources – which includes resources assigned to lower-priority tasks that can be evicted. In scoring, the scheduler determines the "goodness" of each feasible machine. The score takes into account user-specified preferences, but is mostly driven by built-in criteria such as minimizing the number and priority of preempted tasks, picking machines that already have a copy of the task's packages, spreading tasks across power and failure domains, and packing quality including putting a mix of high and low priority

tasks onto a single machine to allow the high-priority ones to expand in a load spike.

Borg originally used a variant of E-PVM [4] for scoring, which generates a single cost value across heterogeneous resources and minimizes the change in cost when placing a task. In practice, E-PVM ends up spreading load across all the machines, leaving headroom for load spikes – but at the expense of increased fragmentation, especially for large tasks that need most of the machine; we sometimes call this "worst fit".

The opposite end of the spectrum is "best fit", which tries to fill machines as tightly as possible. This leaves some machines empty of user jobs (they still run storage servers), so placing large tasks is straightforward, but the tight packing penalizes any mis-estimations in resource requirements by users or Borg. This hurts applications with bursty loads, and is particularly bad for batch jobs which specify low CPU needs so they can schedule easily and try to run opportunistically in unused resources: 20% of non-prod tasks request less than 0.1 CPU cores.

Our current scoring model is a hybrid one that tries to reduce the amount of stranded resources – ones that cannot be used because another resource on the machine is fully allocated. It provides about 3–5% better packing efficiency (defined in [78]) than best fit for our workloads.

If the machine selected by the scoring phase doesn't have enough available resources to fit the new task, Borg *preempts* (kills) lower-priority tasks, from lowest to highest priority, until it does. We add the preempted tasks to the scheduler's pending queue, rather than migrate or hibernate them.[3]

Task startup latency (the time from job submission to a task running) is an area that has received and continues to receive significant attention. It is highly variable, with the median typically about 25 s. Package installation takes about 80% of the total: one of the known bottlenecks is contention for the local disk where packages are written to. To reduce task startup time, the scheduler prefers to assign tasks to machines that already have the necessary packages (programs and data) installed: most packages are immutable and so can be shared and cached. (This is the only form of data locality supported by the Borg scheduler.) In addition, Borg distributes packages to machines in parallel using tree- and torrent-like protocols.

Additionally, the scheduler uses several techniques to let it scale up to cells with tens of thousands of machines (§3.4).

3.3 Borglet

The Borglet is a local Borg agent that is present on every machine in a cell. It starts and stops tasks; restarts them if they fail; manages local resources by manipulating OS kernel settings; rolls over debug logs; and reports the state of the machine to the Borgmaster and other monitoring systems.

[3] Exception: tasks that provide virtual machines for Google Compute Engine users are migrated.

The Borgmaster polls each Borglet every few seconds to retrieve the machine's current state and send it any outstanding requests. This gives Borgmaster control over the rate of communication, avoids the need for an explicit flow control mechanism, and prevents recovery storms [9].

The elected master is responsible for preparing messages to send to the Borglets and for updating the cell's state with their responses. For performance scalability, each Borgmaster replica runs a stateless *link shard* to handle the communication with some of the Borglets; the partitioning is recalculated whenever a Borgmaster election occurs. For resiliency, the Borglet always reports its full state, but the link shards aggregate and compress this information by reporting only differences to the state machines, to reduce the update load at the elected master.

If a Borglet does not respond to several poll messages its machine is marked as down and any tasks it was running are rescheduled on other machines. If communication is restored the Borgmaster tells the Borglet to kill those tasks that have been rescheduled, to avoid duplicates. A Borglet continues normal operation even if it loses contact with the Borgmaster, so currently-running tasks and services stay up even if all Borgmaster replicas fail.

3.4 Scalability

We are not sure where the ultimate scalability limit to Borg's centralized architecture will come from; so far, every time we have approached a limit, we've managed to eliminate it. A single Borgmaster can manage many thousands of machines in a cell, and several cells have arrival rates above 10 000 tasks per minute. A busy Borgmaster uses 10–14 CPU cores and up to 50 GiB RAM. We use several techniques to achieve this scale.

Early versions of Borgmaster had a simple, synchronous loop that accepted requests, scheduled tasks, and communicated with Borglets. To handle larger cells, we split the scheduler into a separate process so it could operate in parallel with the other Borgmaster functions that are replicated for failure tolerance. A scheduler replica operates on a cached copy of the cell state. It repeatedly: retrieves state changes from the elected master (including both assigned and pending work); updates its local copy; does a scheduling pass to assign tasks; and informs the elected master of those assignments. The master will accept and apply these assignments unless they are inappropriate (e.g., based on out of date state), which will cause them to be reconsidered in the scheduler's next pass. This is quite similar in spirit to the optimistic concurrency control used in Omega [69], and indeed we recently added the ability for Borg to use different schedulers for different workload types.

To improve response times, we added separate threads to talk to the Borglets and respond to read-only RPCs. For greater performance, we sharded (partitioned) these functions across the five Borgmaster replicas §3.3. Together,

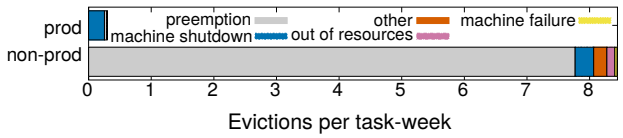


Figure 3: Task-eviction rates and causes for production and non-production workloads. *Data from August 1st 2013.*

these keep the 99%ile response time of the UI below 1 s and the 95%ile of the Borglet polling interval below 10 s.

Several things make the Borg scheduler more scalable:

Score caching: Evaluating feasibility and scoring a machine is expensive, so Borg caches the scores until the properties of the machine or task change – e.g., a task on the machine terminates, an attribute is altered, or a task's requirements change. Ignoring small changes in resource quantities reduces cache invalidations.

Equivalence classes: Tasks in a Borg job usually have identical requirements and constraints, so rather than determining feasibility for every pending task on every machine, and scoring all the feasible machines, Borg only does feasibility and scoring for one task per *equivalence class* – a group of tasks with identical requirements.

Relaxed randomization: It is wasteful to calculate feasibility and scores for all the machines in a large cell, so the scheduler examines machines in a random order until it has found "enough" feasible machines to score, and then selects the best within that set. This reduces the amount of scoring and cache invalidations needed when tasks enter and leave the system, and speeds up assignment of tasks to machines. Relaxed randomization is somewhat akin to the batch sampling of Sparrow [65] while also handling priorities, preemptions, heterogeneity and the costs of package installation.

In our experiments (§5), scheduling a cell's entire workload from scratch typically took a few hundred seconds, but did not finish after more than 3 days when the above techniques were disabled. Normally, though, an online scheduling pass over the pending queue completes in less than half a second.

4. Availability

Failures are the norm in large scale systems [10, 11, 22]. Figure 3 provides a breakdown of task eviction causes in 15 sample cells. Applications that run on Borg are expected to handle such events, using techniques such as replication, storing persistent state in a distributed file system, and (if appropriate) taking occasional checkpoints. Even so, we try to mitigate the impact of these events. For example, Borg:

- automatically reschedules evicted tasks, on a new machine if necessary;
- reduces correlated failures by spreading tasks of a job across failure domains such as machines, racks, and power domains;
- limits the allowed rate of task disruptions and the number of tasks from a job that can be simultaneously down during maintenance activities such as OS or machine upgrades;
- uses declarative desired-state representations and idempotent mutating operations, so that a failed client can harmlessly resubmit any forgotten requests;
- rate-limits finding new places for tasks from machines that become unreachable, because it cannot distinguish between large-scale machine failure and a network partition;
- avoids repeating task::machine pairings that cause task or machine crashes; and
- recovers critical intermediate data written to local disk by repeatedly re-running a logsaver task (§2.4), even if the alloc it was attached to is terminated or moved to another machine. Users can set how long the system keeps trying; a few days is common.

A key design feature in Borg is that already-running tasks continue to run even if the Borgmaster or a task's Borglet goes down. But keeping the master up is still important because when it is down new jobs cannot be submitted or existing ones updated, and tasks from failed machines cannot be rescheduled.

Borgmaster uses a combination of techniques that enable it to achieve 99.99% availability in practice: replication for machine failures; admission control to avoid overload; and deploying instances using simple, low-level tools to minimize external dependencies. Each cell is independent of the others to minimize the chance of correlated operator errors and failure propagation. These goals, not scalability limitations, are the primary argument against larger cells.

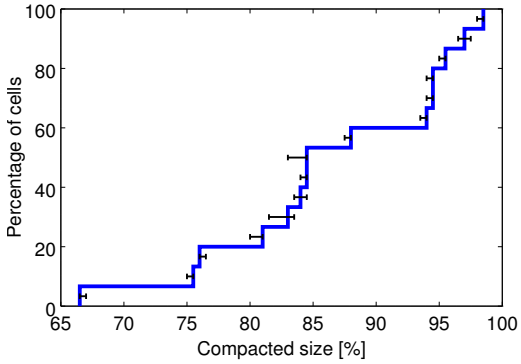
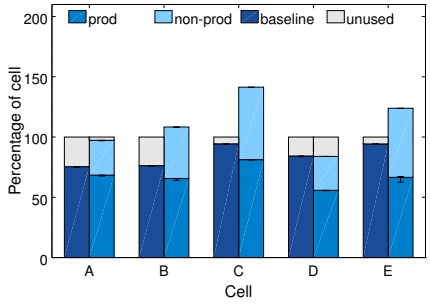


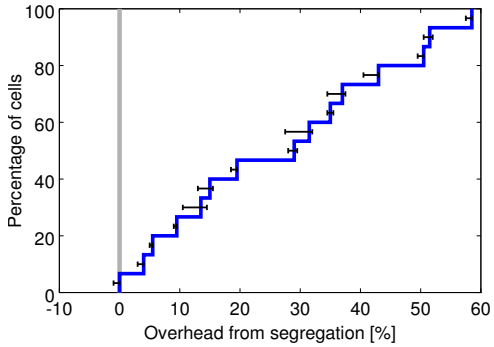
Figure 4: The effects of compaction. *A CDF of the percentage of original cell size achieved after compaction, across 15 cells.*

5. Utilization

One of Borg's primary goals is to make efficient use of Google's fleet of machines, which represents a significant financial investment: increasing utilization by a few percentage points can save millions of dollars. This section discusses and evaluates some of the policies and techniques that Borg uses to do so.



(a) *The left column for each cell shows the original size and the combined workload; the right one shows the segregated case.*



(b) *CDF of additional machines that would be needed if we segregated the workload of 15 representative cells.*

Figure 5: Segregating prod and non-prod work into different cells would need more machines. *Both graphs show how many extra machines would be needed if the* prod *and* non-prod *workloads were sent to separate cells, expressed as a percentage of the minimum number of machines required to run the workload in a single cell. In this, and subsequent CDF plots, the value shown for each cell is derived from the 90%ile of the different cell sizes our experiment trials produced; the error bars show the complete range of values from the trials.*

5.1 Evaluation methodology

Our jobs have placement constraints and need to handle rare workload spikes, our machines are heterogenous, and we run batch jobs in resources reclaimed from service jobs. So, to evaluate our policy choices we needed a more sophisticated metric than "average utilization". After much experimentation we picked *cell compaction*: given a workload, we found out how small a cell it could be fitted into by removing machines until the workload no longer fitted, repeatedly re-packing the workload from scratch to ensure that we didn't get hung up on an unlucky configuration. This provided clean termination conditions and facilitated automated comparisons without the pitfalls of synthetic workload generation and modeling [31]. A quantitative comparison of evaluation techniques can be found in [78]: the details are surprisingly subtle.

It wasn't possible to perform experiments on live production cells, but we used Fauxmaster to obtain high-fidelity simulation results, using data from real production cells and workloads, including all their constraints, actual limits, reservations, and usage data (§5.5). This data came from Borg checkpoints taken on Wednesday 2014-10-01 14:00 PDT. (Other checkpoints produced similar results.) We picked 15 Borg cells to report on by first eliminating special-purpose, test, and small (< 5000 machines) cells, and then sampled the remaining population to achieve a roughly even spread across the range of sizes.

To maintain machine heterogeneity in the compacted cell we randomly selected machines to remove. To maintain workload heterogeneity, we kept it all, except for server and storage tasks tied to a particular machine (e.g., the Borglets). We changed hard constraints to soft ones for jobs larger than half the original cell size, and allowed up to 0.2% tasks to go pending if they were very "picky" and could only be placed on a handful of machines; extensive experiments showed that this produced repeatable results with low variance. If

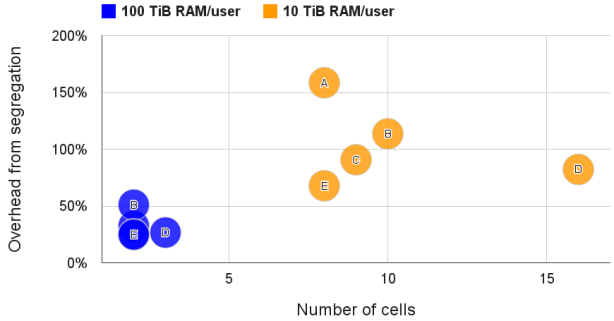
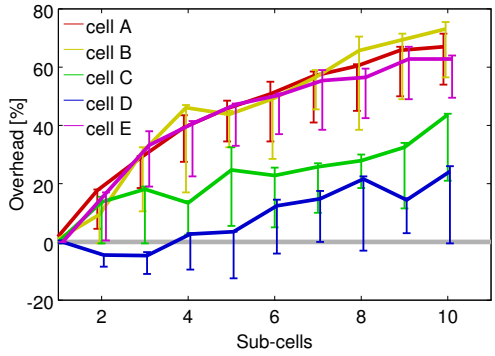


Figure 6: Segregating users would need more machines. *The total number of cells and the additional machines that would be needed if users larger than the threshold shown were given their own private cells, for 5 different cells.*

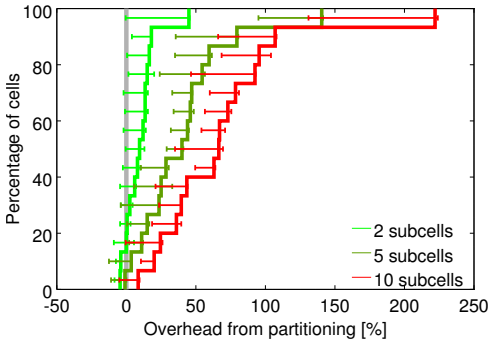
we needed a larger cell than the original we cloned the original cell a few times before compaction; if we needed more cells, we just cloned the original.

Each experiment was repeated 11 times for each cell with different random-number seeds. In the graphs, we use an error bar to display the min and max of the number of machines needed, and select the 90%ile value as the "result" – the mean or median would not reflect what a system administrator would do if they wanted to be reasonably sure that the workload would fit. We believe cell compaction provides a fair, consistent way to compare scheduling policies, and it translates directly into a cost/benefit result: better policies require fewer machines to run the same workload.

Our experiments focused on scheduling (packing) a workload from a point in time, rather than replaying a long-term workload trace. This was partly to avoid the difficulties of coping with open and closed queueing models [71, 79], partly because traditional time-to-completion metrics don't apply to our environment with its long-running services, partly to provide clean signals for making comparisons,



(a) *Additional machines that would be needed as a function of the number of smaller cells for five different original cells.*



(b) *A CDF of additional machines that would be needed to divide each of 15 different cells into 2, 5 or 10 cells.*

Figure 7: Subdividing cells into smaller ones would require more machines. *The additional machines (as a percentage of the single-cell case) that would be needed if we divided these particular cells into a varying number of smaller cells.*

partly because we don't believe the results would be significantly different, and partly a practical matter: we found ourselves consuming 200 000 Borg CPU cores for our experiments at one point—even at Google's scale, this is a non-trivial investment.

In production, we deliberately leave significant headroom for workload growth, occasional "black swan" events, load spikes, machine failures, hardware upgrades, and large-scale partial failures (e.g., a power supply bus duct). Figure 4 shows how much smaller our real-world cells would be if we were to apply cell compaction to them. The baselines in the graphs that follow use these compacted sizes.

5.2 Cell sharing

Nearly all of our machines run both prod and non-prod tasks at the same time: 98% of the machines in shared Borg cells, 83% across the entire set of machines managed by Borg. (We have a few dedicated cells for special uses.)

Since many other organizations run user-facing and batch jobs in separate clusters, we examined what would happen if we did the same. Figure 5 shows that segregating prod and non-prod work would need 20–30% more machines in the median cell to run our workload. That's because prod jobs usually reserve resources to handle rare workload spikes, but don't use these resources most of the time. Borg reclaims the unused resources (§5.5) to run much of the non-prod work, so we need fewer machines overall.

Most Borg cells are shared by thousands of users. Figure 6 shows why. For this test, we split off a user's workload into a new cell if they consumed at least 10 TiB of memory (or 100 TiB). Our existing policy looks good: even with the larger threshold, we would need 2–16× as many cells, and 20–150% additional machines. Once again, pooling resources significantly reduces costs.

But perhaps packing unrelated users and job types onto the same machines results in CPU interference, and so we would need more machines to compensate? To assess this, we looked at how the CPI (cycles per instruction) changed for tasks in different environments running on the same machine type with the same clock speed. Under these conditions, CPI values are comparable and can be used as a proxy for performance interference, since a doubling of CPI doubles the runtime of a CPU-bound program. The data was gathered from ∼ 12 000 randomly selected prod tasks over a week, counting cycles and instructions over a 5 minute interval using the hardware profiling infrastructure described in [83], and weighting samples so that every second of CPU time is counted equally. The results were not clear-cut.

(1) We found that CPI was positively correlated with two measurements over the same time interval: the overall CPU usage on the machine, and (largely independently) the number of tasks on the machine; adding a task to a machine increases the CPI of other tasks by 0.3% (using a linear model fitted to the data); increasing machine CPU usage by 10% increases CPI by less than 2%. But even though the correlations are statistically significant, they only explain 5% of the variance we saw in CPI measurements; other factors dominate, such as inherent differences in applications and specific interference patterns [24, 83].

(2) Comparing the CPIs we sampled from shared cells to ones from a few dedicated cells with less diverse applications, we saw a mean CPI of 1.58 ($\sigma = 0.35$) in shared cells and a mean of 1.53 ($\sigma = 0.32$) in dedicated cells – i.e., CPU performance is about 3% worse in shared cells.

(3) To address the concern that applications in different cells might have different workloads, or even suffer selection bias (maybe programs that are more sensitive to interference had been moved to dedicated cells), we looked at the CPI of the Borglet, which runs on all the machines in both types of cell. We found it had a CPI of 1.20 ($\sigma = 0.29$) in dedicated cells and 1.43 ($\sigma = 0.45$) in shared ones, suggesting that it runs 1.19× as fast in a dedicated cell as in a shared one, although this over-weights the effect of lightly loaded machines, slightly biasing the result in favor of dedicated cells.

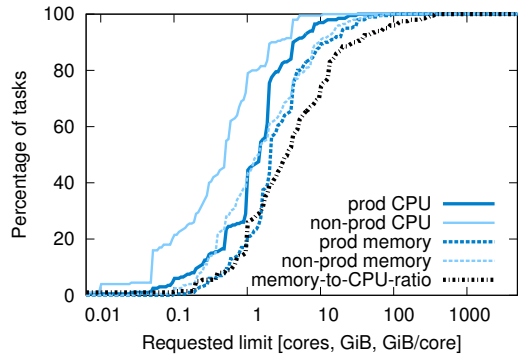


Figure 8: No bucket sizes fit most of the tasks well. *CDF of requested CPU and memory requests across our sample cells. No one value stands out, although a few integer CPU core sizes are somewhat more popular.*

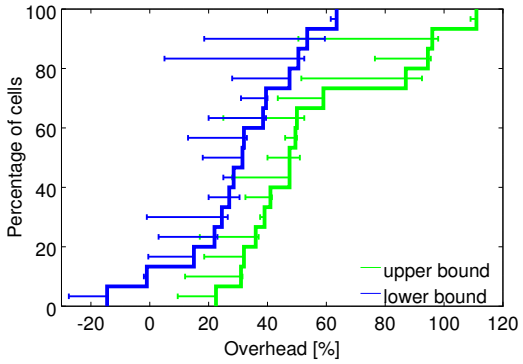


Figure 9: "Bucketing" resource requirements would need more machines. *A CDF of the additional overheads that would result from rounding up CPU and memory requests to the next nearest powers of 2 across 15 cells. The lower and upper bounds straddle the actual values (see the text).*

These experiments confirm that performance comparisons at warehouse-scale are tricky, reinforcing the observations in [51], and also suggest that sharing doesn't drastically increase the cost of running programs.

But even assuming the least-favorable of our results, sharing is still a win: the CPU slowdown is outweighed by the decrease in machines required over several different partitioning schemes, and the sharing advantages apply to all resources including memory and disk, not just CPU.

5.3 Large cells

Google builds large cells, both to allow large computations to be run, and to decrease resource fragmentation. We tested the effects of the latter by partitioning the workload for a cell across multiple smaller cells – by first randomly permuting the jobs and then assigning them in a round-robin manner among the partitions. Figure 7 confirms that using smaller cells would require significantly more machines.

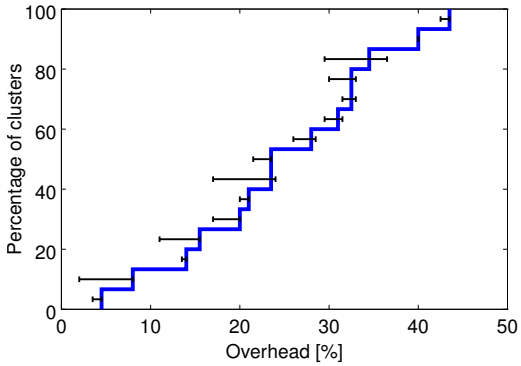


Figure 10: Resource reclamation is quite effective. *A CDF of the additional machines that would be needed if we disabled it for 15 representative cells.*

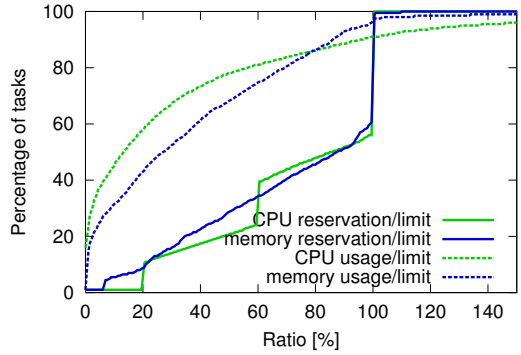


Figure 11: Resource estimation is successful at identifying unused resources. *The dotted lines shows CDFs of the ratio of CPU and memory usage to the request (limit) for tasks across 15 cells. Most tasks use much less than their limit, although a few use more CPU than requested. The solid lines show the CDFs of the ratio of CPU and memory reservations to the limits; these are closer to 100%. The straight lines are artifacts of the resource-estimation process.*

5.4 Fine-grained resource requests

Borg users request CPU in units of milli-cores, and memory and disk space in bytes. (A *core* is a processor hyperthread, normalized for performance across machine types.) Figure 8 shows that they take advantage of this granularity: there are few obvious "sweet spots" in the amount of memory or CPU cores requested, and few obvious correlations between these resources. These distributions are quite similar to the ones presented in [68], except that we see slightly larger memory requests at the 90%ile and above.

Offering a set of fixed-size containers or virtual machines, although common among IaaS (infrastructure-as-a-service) providers [7, 33], would not be a good match to our needs. To show this, we "bucketed" CPU core and memory resource limits for prod jobs and allocs (§2.4) by rounding them up to the next nearest power of two in each resource dimension, starting at 0.5 cores for CPU and 1 GiB for RAM. Figure 9 shows that doing so would require 30–50% more resources in the median case. The upper bound comes from allocating an entire machine to large tasks that didn't fit after quadru-

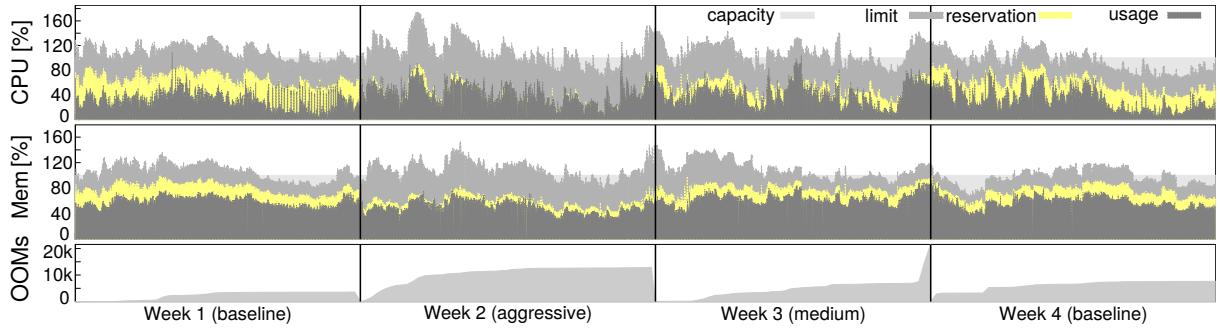


Figure 12: More aggressive resource estimation can reclaim more resources, with little effect on out-of-memory events (OOMs). *A timeline (starting on 2013-11-11) for one production cell of usage, reservation and limit averaged over 5-minute windows and cumulative out-of-memory events; the slope of the latter is the aggregate rate of OOMs. Vertical bars separate weeks with different resource estimation settings.*

pling the original cell before compaction began; the lower bound from allowing these tasks to go pending. (This is less than the roughly 100% overhead reported in [37] because we supported more than 4 buckets and permitted CPU and RAM capacity to scale independently.)

5.5 Resource reclamation

A job can specify a resource *limit* – an upper bound on the resources that each task should be granted. The limit is used by Borg to determine if the user has enough quota to admit the job, and to determine if a particular machine has enough free resources to schedule the task. Just as there are users who buy more quota than they need, there are users who request more resources than their tasks will use, because Borg will normally kill a task that tries to use more RAM or disk space than it requested, or throttle CPU to what it asked for. In addition, some tasks occasionally need to use all their resources (e.g., at peak times of day or while coping with a denial-of-service attack), but most of the time do not.

Rather than waste allocated resources that are not currently being consumed, we estimate how many resources a task will use and reclaim the rest for work that can tolerate lower-quality resources, such as batch jobs. This whole process is called *resource reclamation*. The estimate is called the task's *reservation*, and is computed by the Borgmaster every few seconds, using fine-grained *usage* (resource-consumption) information captured by the Borglet. The initial reservation is set equal to the resource request (the limit); after 300 s, to allow for startup transients, it decays slowly towards the actual usage plus a safety margin. The reservation is rapidly increased if the usage exceeds it.

The Borg scheduler uses limits to calculate feasibility (§3.2) for prod tasks,[4] so they never rely on reclaimed resources and aren't exposed to resource oversubscription; for non-prod tasks, it uses the reservations of existing tasks so the new tasks can be scheduled into reclaimed resources.

A machine may run out of resources at runtime if the reservations (predictions) are wrong – even if all tasks use less than their limits. If this happens, we kill or throttle non-prod tasks, never prod ones.

[4] To be precise, high-priority latency-sensitive ones – see §6.2.

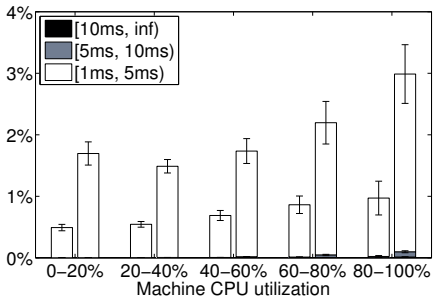


Figure 13: Scheduling delays as a function of load. *A plot of how often a runnable thread had to wait longer than 1ms to get access to a CPU, as a function of how busy the machine was. In each pair of bars, latency-sensitive tasks are on the left, batch ones on the right. In only a few percent of the time did a thread have to wait longer than 5 ms to access a CPU (the white bars); they almost never had to wait longer (the darker bars). Data from a representative cell for the month of December 2013; error bars show day-to-day variance.*

Figure 10 shows that many more machines would be required without resource reclamation. About 20% of the workload (§6.2) runs in reclaimed resources in a median cell.

We can see more details in Figure 11, which shows the ratio of reservations and usage to limits. A task that exceeds its memory limit will be the first to be preempted if resources are needed, regardless of its priority, so it is rare for tasks to exceed their memory limit. On the other hand, CPU can readily be throttled, so short-term spikes can push usage above reservation fairly harmlessly.

Figure 11 suggests that resource reclamation may be unnecessarily conservative: there is significant area between the reservation and usage lines. To test this, we picked a live production cell and adjusted the parameters of its resource estimation algorithm to an aggressive setting for a week by reducing the safety margin, and then to an medium setting that was mid-way between the baseline and aggressive settings for the next week, and then reverted to the baseline.

Figure 12 shows what happened. Reservations are clearly closer to usage in the second week, and somewhat less so in the third, with the biggest gaps shown in the baseline weeks (1st and 4th). As anticipated, the rate of out-of-memory (OOM) events increased slightly in weeks 2 and 3.[5] After reviewing these results, we decided that the net gains outweighed the downsides, and deployed the medium resource reclamation parameters to other cells.

6. Isolation

50% of our machines run 9 or more tasks; a 90%ile machine has about 25 tasks and will be running about 4500 threads [83]. Although sharing machines between applications increases utilization, it also requires good mechanisms to prevent tasks from interfering with one another. This applies to both security and performance.

6.1 Security isolation

We use a Linux chroot jail as the primary security isolation mechanism between multiple tasks on the same machine. To allow remote debugging, we used to distribute (and rescind) ssh keys automatically to give a user access to a machine only while it was running tasks for the user. For most users, this has been replaced by the borgssh command, which collaborates with the Borglet to construct an ssh connection to a shell that runs in the same chroot and cgroup as the task, locking down access even more tightly.

VMs and security sandboxing techniques are used to run external software by Google's AppEngine (GAE) [38] and Google Compute Engine (GCE). We run each hosted VM in a KVM process [54] that runs as a Borg task.

6.2 Performance isolation

Early versions of Borglet had relatively primitive resource isolation enforcement: post-hoc usage checking of memory, disk space and CPU cycles, combined with termination of tasks that used too much memory or disk and aggressive application of Linux's CPU priorities to rein in tasks that used too much CPU. But it was still too easy for rogue tasks to affect the performance of other tasks on the machine, so some users inflated their resource requests to reduce the number of tasks that Borg could co-schedule with theirs, thus decreasing utilization. Resource reclamation could claw back some of the surplus, but not all, because of the safety margins involved. In the most extreme cases, users petitioned to use dedicated machines or cells.

Now, all Borg tasks run inside a Linux cgroup-based resource container [17, 58, 62] and the Borglet manipulates the container settings, giving much improved control because the OS kernel is in the loop. Even so, occasional low-level resource interference (e.g., memory bandwidth or L3 cache pollution) still happens, as in [60, 83].

[5] The anomaly at the end of week 3 is unrelated to this experiment.

To help with overload and overcommitment, Borg tasks have an application class or *appclass*. The most important distinction is between the *latency-sensitive* (LS) appclasses and the rest, which we call *batch* in this paper. LS tasks are used for user-facing applications and shared infrastructure services that require fast response to requests. High-priority LS tasks receive the best treatment, and are capable of temporarily starving batch tasks for several seconds at a time.

A second split is between *compressible* resources (e.g., CPU cycles, disk I/O bandwidth) that are rate-based and can be reclaimed from a task by decreasing its quality of service without killing it; and *non-compressible* resources (e.g., memory, disk space) which generally cannot be reclaimed without killing the task. If a machine runs out of non-compressible resources, the Borglet immediately terminates tasks, from lowest to highest priority, until the remaining reservations can be met. If the machine runs out of compressible resources, the Borglet throttles usage (favoring LS tasks) so that short load spikes can be handled without killing any tasks. If things do not improve, Borgmaster will remove one or more tasks from the machine.

A user-space control loop in the Borglet assigns memory to containers based on predicted future usage (for prod tasks) or on memory pressure (for non-prod ones); handles Out-of-Memory (OOM) events from the kernel; and kills tasks when they try to allocate beyond their memory limits, or when an over-committed machine actually runs out of memory. Linux's eager file-caching significantly complicates the implementation because of the need for accurate memory-accounting.

To improve performance isolation, LS tasks can reserve entire physical CPU cores, which stops other LS tasks from using them. Batch tasks are permitted to run on any core, but they are given tiny scheduler shares relative to the LS tasks. The Borglet dynamically adjusts the resource caps of greedy LS tasks in order to ensure that they do not starve batch tasks for multiple minutes, selectively applying CFS bandwidth control when needed [75]; shares are insufficient because we have multiple priority levels.

Like Leverich [56], we found that the standard Linux CPU scheduler (CFS) required substantial tuning to support both low latency and high utilization. To reduce scheduling delays, our version of CFS uses extended per-cgroup load history [16], allows preemption of batch tasks by LS tasks, and reduces the scheduling quantum when multiple LS tasks are runnable on a CPU. Fortunately, many of our applications use a thread-per-request model, which mitigates the effects of persistent load imbalances. We sparingly use cpusets to allocate CPU cores to applications with particularly tight latency requirements. Some results of these efforts are shown in Figure 13. Work continues in this area, adding thread placement and CPU management that is NUMA-, hyperthreading-, and power-aware (e.g., [81]), and improving the control fidelity of the Borglet.

Tasks are permitted to consume resources up to their limit. Most of them are allowed to go beyond that for compressible resources like CPU, to take advantage of unused (slack) resources. Only 5% of LS tasks disable this, presumably to get better predictability; fewer than 1% of batch tasks do. Using slack memory is disabled by default, because it increases the chance of a task being killed, but even so, 10% of LS tasks override this, and 79% of batch tasks do so because it's a default setting of the MapReduce framework. This complements the results for reclaimed resources (§5.5). Batch tasks are willing to exploit unused as well as reclaimed memory opportunistically: most of the time this works, although the occasional batch task is sacrificed when an LS task needs resources in a hurry.

7. Related work

Resource scheduling has been studied for decades, in contexts as varied as wide-area HPC supercomputing Grids, networks of workstations, and large-scale server clusters. We focus here on only the most relevant work in the context of large-scale server clusters.

Several recent studies have analyzed cluster traces from Yahoo!, Google, and Facebook [20, 52, 63, 68, 70, 80, 82], and illustrate the challenges of scale and heterogeneity inherent in these modern datacenters and workloads. [69] contains a taxonomy of cluster manager architectures.

Apache Mesos [45] splits the resource management and placement functions between a central resource manager (somewhat like Borgmaster minus its scheduler) and multiple "frameworks" such as Hadoop [41] and Spark [73] using an offer-based mechanism. Borg mostly centralizes these functions using a request-based mechanism that scales quite well. DRF [29, 35, 36, 66] was initially developed for Mesos; Borg uses priorities and admission quotas instead. The Mesos developers have announced ambitions to extend Mesos to include speculative resource assignment and reclamation, and to fix some of the issues identified in [69].

YARN [76] is a Hadoop-centric cluster manager. Each application has a manager that negotiates for the resources it needs with a central resource manager; this is much the same scheme that Google MapReduce jobs have used to obtain resources from Borg since about 2008. YARN's resource manager only recently became fault tolerant. A related open-source effort is the *Hadoop Capacity Scheduler* [42] which provides multi-tenant support with capacity guarantees, hierarchical queues, elastic sharing and fairness. YARN has recently been extended to support multiple resource types, priorities, preemptions, and advanced admission control [21]. The Tetris research prototype [40] supports makespan-aware job packing.

Facebook's *Tupperware* [64], is a Borg-like system for scheduling cgroup containers on a cluster; only a few details have been disclosed, although it seems to provide a form of resource reclamation. Twitter has open-sourced *Aurora* [5], a Borg-like scheduler for long running services that runs on top of Mesos, with a configuration language and state machine similar to Borg's.

The *Autopilot* system from Microsoft [48] provides "automating software provisioning and deployment; system monitoring; and carrying out repair actions to deal with faulty software and hardware" for Microsoft clusters. The Borg ecosystem provides similar features, but space precludes a discussion here; Isaard [48] outlines many best practices that we adhere to as well.

Quincy [49] uses a network flow model to provide fairness- and data locality-aware scheduling for data-processing DAGs on clusters of a few hundred nodes. Borg uses quota and priorities to share resources among users and scales to tens of thousands of machines. Quincy handles execution graphs directly while this is built separately on top of Borg.

Cosmos [44] focuses on batch processing, with an emphasis on ensuring that its users get fair access to resources they have donated to the cluster. It uses a per-job manager to acquire resources; few details are publicly available.

Microsoft's *Apollo* system [13] uses per-job schedulers for short-lived batch jobs to achieve high throughput on clusters that seem to be comparably-sized to Borg cells. Apollo uses opportunistic execution of lower-priority background work to boost utilization to high levels at the cost of (sometimes) multi-day queueing delays. Apollo nodes provide a prediction matrix of starting times for tasks as a function of size over two resource dimensions, which the schedulers combine with estimates of startup costs and remote-data-access to make placement decisions, modulated by random delays to reduce collisions. Borg uses a central scheduler for placement decisions based on state about prior allocations, can handle more resource dimensions, and focuses on the needs of high-availability, long-running applications; Apollo can probably handle a higher task arrival rate.

Alibaba's *Fuxi* [84] supports data-analysis workloads; it has been running since 2009. Like Borgmaster, a central FuxiMaster (replicated for failure-tolerance) gathers resource-availability information from nodes, accepts requests from applications, and matches one to the other. The Fuxi incremental scheduling policy is the inverse of Borg's equivalence classes: instead of matching each task to one of a suitable set of machines, Fuxi matches newly-available resources against a backlog of pending work. Like Mesos, Fuxi allows "virtual resource" types to be defined. Only synthetic workload results are publicly available.

Omega [69] supports multiple parallel, specialized "verticals" that are each roughly equivalent to a Borgmaster minus its persistent store and link shards. Omega schedulers use optimistic concurrency control to manipulate a shared representation of desired and observed cell state stored in a central persistent store, which is synced to/from the Borglets by a separate link component. The Omega architecture was designed to support multiple distinct workloads that have their

own application-specific RPC interface, state machines, and scheduling policies (e.g., long-running servers, batch jobs from various frameworks, infrastructure services like cluster storage systems, virtual machines from the Google Cloud Platform). On the other hand, Borg offers a "one size fits all" RPC interface, state machine semantics, and scheduler policy, which have grown in size and complexity over time as a result of needing to support many disparate workloads, and scalability has not yet been a problem (§3.4).

Google's open-source *Kubernetes* system [53] places applications in Docker containers [28] onto multiple host nodes. It runs both on bare metal (like Borg) and on various cloud hosting providers, such as Google Compute Engine. It is under active development by many of the same engineers who built Borg. Google offers a hosted version called Google Container Engine [39]. We discuss how lessons from Borg are being applied to Kubernetes in the next section.

The high-performance computing community has a long tradition of work in this area (e.g., Maui, Moab, Platform LSF [2, 47, 50]); however the requirements of scale, workloads and fault tolerance are different from those of Google's cells. In general, such systems achieve high utilization by having large backlogs (queues) of pending work.

Virtualization providers such as VMware [77] and datacenter solution providers such as HP and IBM [46] provide cluster management solutions that typically scale to O(1000) machines. In addition, several research groups have prototyped systems that improve the quality of scheduling decisions in certain ways (e.g., [25, 40, 72, 74]).

And finally, as we have indicated, another important part of managing large scale clusters is automation and "operator scaleout". [43] describes how planning for failures, multitenancy, health checking, admission control, and restartability are necessary to achieve high numbers of machines per operator. Borg's design philosophy is similar and allows us to support tens of thousands of machines per operator (SRE).

8. Lessons and future work

In this section we recount some of the qualitative lessons we've learned from operating Borg in production for more than a decade, and describe how these observations have been leveraged in designing Kubernetes [53].

8.1 Lessons learned: the bad

We begin with some features of Borg that serve as cautionary tales, and informed alternative designs in Kubernetes.

Jobs are restrictive as the only grouping mechanism for tasks. Borg has no first-class way to manage an entire multi-job service as a single entity, or to refer to related instances of a service (e.g., canary and production tracks). As a hack, users encode their service topology in the job name and build higher-level management tools to parse these names. At the other end of the spectrum, it's not possible to refer to arbitrary subsets of a job, which leads to problems like inflexible semantics for rolling updates and job resizing.

To avoid such difficulties, Kubernetes rejects the job notion and instead organizes its scheduling units (pods) using *labels* – arbitrary key/value pairs that users can attach to any object in the system. The equivalent of a Borg job can be achieved by attaching a job:*jobname* label to a set of pods, but any other useful grouping can be represented too, such as the service, tier, or release-type (e.g., production, staging, test). Operations in Kubernetes identify their targets by means of a label query that selects the objects that the operation should apply to. This approach gives more flexibility than the single fixed grouping of a job.

One IP address per machine complicates things. In Borg, all tasks on a machine use the single IP address of their host, and thus share the host's port space. This causes a number of difficulties: Borg must schedule ports as a resource; tasks must pre-declare how many ports they need, and be willing to be told which ones to use when they start; the Borglet must enforce port isolation; and the naming and RPC systems must handle ports as well as IP addresses.

Thanks to the advent of Linux namespaces, VMs, IPv6, and software-defined networking, Kubernetes can take a more user-friendly approach that eliminates these complications: every pod and service gets its own IP address, allowing developers to choose ports rather than requiring their software to adapt to the ones chosen by the infrastructure, and removes the infrastructure complexity of managing ports.

Optimizing for power users at the expense of casual ones. Borg provides a large set of features aimed at "power users" so they can fine-tune the way their programs are run (the BCL specification lists about 230 parameters): the initial focus was supporting the largest resource consumers at Google, for whom efficiency gains were paramount. Unfortunately the richness of this API makes things harder for the "casual" user, and constrains its evolution. Our solution has been to build automation tools and services that run on top of Borg, and determine appropriate settings from experimentation. These benefit from the freedom to experiment afforded by failure-tolerant applications: if the automation makes a mistake it is a nuisance, not a disaster.

8.2 Lessons learned: the good

On the other hand, a number of Borg's design features have been remarkably beneficial and have stood the test of time.

Allocs are useful. The Borg alloc abstraction spawned the widely-used logsaver pattern (§2.4) and another popular one in which a simple data-loader task periodically updates the data used by a web server. Allocs and packages allow such helper services to be developed by separate teams. The Kubernetes equivalent of an alloc is the *pod*, which is a resource envelope for one or more containers that are always scheduled onto the same machine and can share resources. Kubernetes uses helper containers in the same pod instead of tasks in an alloc, but the idea is the same.

Cluster management is more than task management. Although Borg's primary role is to manage the lifecycles of tasks and machines, the applications that run on Borg benefit from many other cluster services, including naming and load balancing. Kubernetes supports naming and load balancing using the *service* abstraction: a service has a name and a dynamic set of pods defined by a label selector. Any container in the cluster can connect to the service using the service name. Under the covers, Kubernetes automatically load-balances connections to the service among the pods that match the label selector, and keeps track of where the pods are running as they get rescheduled over time due to failures.

Introspection is vital. Although Borg almost always "just works," when something goes wrong, finding the root cause can be challenging. An important design decision in Borg was to surface debugging information to all users rather than hiding it: Borg has thousands of users, so "self-help" has to be the first step in debugging. Although this makes it harder for us to deprecate features and change internal policies that users come to rely on, it is still a win, and we've found no realistic alternative. To handle the enormous volume of data, we provide several levels of UI and debugging tools, so users can quickly identify anomalous events related to their jobs, and then drill down to detailed event and error logs from their applications and the infrastructure itself.

Kubernetes aims to replicate many of Borg's introspection techniques. For example, it ships with tools such as cAdvisor [15] for resource monitoring, and log aggregation based on Elasticsearch/Kibana [30] and Fluentd [32]. The master can be queried for a snapshot of its objects' state. Kubernetes has a unified mechanism that all components can use to record events (e.g., a pod being scheduled, a container failing) that are made available to clients.

The master is the kernel of a distributed system. Borgmaster was originally designed as a monolithic system, but over time, it became more of a kernel sitting at the heart of an ecosystem of services that cooperate to manage user jobs. For example, we split off the scheduler and the primary UI (Sigma) into separate processes, and added services for admission control, vertical and horizontal autoscaling, re-packing tasks, periodic job submission (cron), workflow management, and archiving system actions for off-line querying. Together, these have allowed us to scale up the workload and feature set without sacrificing performance or maintainability.

The Kubernetes architecture goes further: it has an API server at its core that is responsible only for processing requests and manipulating the underlying state objects. The cluster management logic is built as small, composable micro-services that are clients of this API server, such as the replication controller, which maintains the desired number of replicas of a pod in the face of failures, and the node controller, which manages the machine lifecycle.

8.3 Conclusion

Virtually all of Google's cluster workloads have switched to use Borg over the past decade. We continue to evolve it, and have applied the lessons we learned from it to Kubernetes.

Acknowledgments

The authors of this paper performed the evaluations and wrote the paper, but the dozens of engineers who designed, implemented, and maintained Borg's components and ecosystem are the key to its success. We list here just those who participated most directly in the design, implementation, and operation of the Borgmaster and Borglets. Our apologies if we missed anybody.

The initial Borgmaster was primarily designed and implemented by Jeremy Dion and Mark Vandevoorde, with Ben Smith, Ken Ashcraft, Maricia Scott, Ming-Yee Iu, and Monika Henzinger. The initial Borglet was primarily designed and implemented by Paul Menage.

Subsequent contributors include Abhishek Rai, Abhishek Verma, Andy Zheng, Ashwin Kumar, Beng-Hong Lim, Bin Zhang, Bolu Szewczyk, Brian Budge, Brian Grant, Brian Wickman, Chengdu Huang, Cynthia Wong, Daniel Smith, Dave Bort, David Oppenheimer, David Wall, Dawn Chen, Eric Haugen, Eric Tune, Ethan Solomita, Gaurav Dhiman, Geeta Chaudhry, Greg Roelofs, Grzegorz Czajkowski, James Eady, Jarek Kusmierek, Jaroslaw Przybylowicz, Jason Hickey, Javier Kohen, Jeremy Lau, Jerzy Szczepkowski, John Wilkes, Jonathan Wilson, Joso Eterovic, Jutta Degener, Kai Backman, Kamil Yurtsever, Kenji Kaneda, Kevan Miller, Kurt Steinkraus, Leo Landa, Liza Fireman, Madhukar Korupolu, Mark Logan, Markus Gutschke, Matt Sparks, Maya Haridasan, Michael Abd-El-Malek, Michael Kenniston, Mukesh Kumar, Nate Calvin, Onufry Wojtaszczyk, Patrick Johnson, Pedro Valenzuela, Piotr Witusowski, Praveen Kallakuri, Rafal Sokolowski, Richard Gooch, Rishi Gosalia, Rob Radez, Robert Hagmann, Robert Jardine, Robert Kennedy, Rohit Jnagal, Roy Bryant, Rune Dahl, Scott Garriss, Scott Johnson, Sean Howarth, Sheena Madan, Smeeta Jalan, Stan Chesnutt, Temo Arobelidze, Tim Hockin, Todd Wang, Tomasz Blaszczyk, Tomasz Wozniak, Tomek Zielonka, Victor Marmol, Vish Kannan, Vrigo Gokhale, Walfredo Cirne, Walt Drummond, Weiran Liu, Xiaopan Zhang, Xiao Zhang, Ye Zhao, and Zohaib Maya.

The Borg SRE team has also been crucial, and has included Adam Rogoyski, Alex Milivojevic, Anil Das, Cody Smith, Cooper Bethea, Folke Behrens, Matt Liggett, James Sanford, John Millikin, Matt Brown, Miki Habryn, Peter Dahl, Robert van Gent, Seppi Wilhelmi, Seth Hettich, Torsten Marek, and Viraj Alankar. The Borg configuration language (BCL) and borgcfg tool were originally developed by Marcel van Lohuizen and Robert Griesemer.

We thank our reviewers (especially Eric Brewer, Malte Schwarzkopf and Tom Rodeheffer), and our shepherd, Christos Kozyrakis, for their feedback on this paper.

References

[1] O. A. Abdul-Rahman and K. Aida. Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon. In *Proc. IEEE Int'l Conf. on Cloud Computing Technology and Science (CloudCom)*, pages 272–277, Singapore, Dec. 2014.

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

[3] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: fault-tolerant stream processing at internet scale. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 734–746, Riva del Garda, Italy, Aug. 2013.

[4] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Trans. Parallel Distrib. Syst.*, 11(7):760–768, July 2000.

[5] Apache Aurora. `http://aurora.incubator.apache.org/`, 2014.

[6] Aurora Configuration Tutorial. `https://aurora.incubator.apache.org/documentation/latest/configuration-tutorial/`, 2014.

[7] AWS. Amazon Web Services VM Instances. `http://aws.amazon.com/ec2/instance-types/`, 2014.

[8] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, Asilomar, CA, USA, Jan. 2011.

[9] M. Baker and J. Ousterhout. Availability in the Sprite distributed file system. *Operating Systems Review*, 25(2):95–98, Apr. 1991.

[10] L. A. Barroso, J. Clidaras, and U. Hölzle. *The datacenter as a computer: an introduction to the design of warehouse-scale machines*. Morgan Claypool Publishers, 2nd edition, 2013.

[11] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: the Google cluster architecture. In *IEEE Micro*, pages 22–28, 2003.

[12] I. Bokharouss. GCL Viewer: a study in improving the understanding of GCL programs. Technical report, Eindhoven Univ. of Technology, 2008. MS thesis.

[13] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[14] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 335–350, Seattle, WA, USA, 2006.

[15] cAdvisor. `https://github.com/google/cadvisor`, 2014.

[16] CFS per-entity load patches. `http://lwn.net/Articles/531853`, 2013.

[17] cgroups. `http://en.wikipedia.org/wiki/Cgroups`, 2014.

[18] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 363–375, Toronto, Ontario, Canada, 2010.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. *ACM Trans. on Computer Systems*, 26(2):4:1–4:26, June 2008.

[20] Y. Chen, S. Alspaugh, and R. H. Katz. Design insights for MapReduce from diverse production workloads. Technical Report UCB/EECS–2012–17, UC Berkeley, Jan. 2012.

[21] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: if you're late don't blame us! In *Proc. ACM Symp. on Cloud Computing (SoCC)*, pages 2:1–2:14, Seattle, WA, USA, 2014.

[22] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2012.

[23] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[24] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 201.

[25] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–144, Salt Lake City, UT, USA, 2014.

[26] S. Di, D. Kondo, and W. Cirne. Characterization and comparison of cloud versus Grid workloads. In *International Conference on Cluster Computing (IEEE CLUSTER)*, pages 230–238, Beijing, China, Sept. 2012.

[27] S. Di, D. Kondo, and C. Franck. Characterizing cloud applications on a Google data center. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, Lyon, France, Oct. 2013.

[28] Docker Project. `https://www.docker.io/`, 2014.

[29] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial. No justified complaints: on fair sharing of multiple resources. In *Proc. Innovations in Theoretical Computer Science (ITCS)*, pages 68–75, Cambridge, MA, USA, 2012.

[30] ElasticSearch. `http://www.elasticsearch.org`, 2014.

[31] D. G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2014.

[32] Fluentd. `http://www.fluentd.org/`, 2014.

[33] GCE. Google Compute Engine. `http://cloud.google.com/products/compute-engine/`, 2014.

[34] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, pages 29–43, Bolton Landing, NY, USA, 2003. ACM.

[35] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: fair allocation of multiple resource types. In *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 323–326, 2011.

[36] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: max-min fair sharing for datacenter jobs with constraints. In *Proc. European Conf. on Computer Systems (EuroSys)*, pages 365–378, Prague, Czech Republic, 2013.

[37] D. Gmach, J. Rolia, and L. Cherkasova. Selling T-shirts and time shares in the cloud. In *Proc. IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 539–546, Ottawa, Canada, 2012.

[38] Google App Engine. `http://cloud.google.com/AppEngine`, 2014.

[39] Google Container Engine (GKE). `https://cloud.google.com/container-engine/`, 2015.

[40] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proc. ACM SIGCOMM*, Aug. 2014.

[41] Apache Hadoop Project. `http://hadoop.apache.org/`, 2009.

[42] Hadoop MapReduce Next Generation – Capacity Scheduler. `http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`, 2013.

[43] J. Hamilton. On designing and deploying internet-scale services. In *Proc. Large Installation System Administration Conf. (LISA)*, pages 231–242, Dallas, TX, USA, Nov. 2007.

[44] P. Helland. Cosmos: big data and big challenges. `http://research.microsoft.com/en-us/events/fs2011/helland\_cosmos\_big\_data\_and\_big\_challenges.pdf`, 2011.

[45] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2011.

[46] IBM Platform Computing. `http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/clustermanager/index.html`.

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

[48] M. Isaard. Autopilot: Automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2), 2007.

[49] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, 2009.

[50] D. B. Jackson, Q. Snell, and M. J. Clement. Core algorithms of the Maui scheduler. In *Proc. Int'l Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer-Verlag, 2001.

[51] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City, UT, Nov. 2012.

[52] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production MapReduce cluster. In *Proc. IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 94–103, 2010.

[53] Kubernetes. `http://kubernetes.io`, Aug. 2014.

[54] Kernel Based Virtual Machine. `http://www.linux-kvm.org`.

[55] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.

[56] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proc. European Conf. on Computer Systems (EuroSys)*, page 4, 2014.

[57] Z. Liu and S. Cho. Characterizing machines and workloads on a Google cluster. In *Proc. Int'l Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS)*, Pittsburgh, PA, USA, Sept. 2012.

[58] Google LMCTFY project (let me contain that for you). `http://github.com/google/lmctfy`, 2014.

[59] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Conference*, pages 135–146, Indianapolis, IA, USA, 2010.

[60] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. Int'l Symp. on Microarchitecture (Micro)*, Porto Alegre, Brazil, 2011.

[61] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 330–339, Singapore, Sept. 2010.

[62] P. Menage. Linux control groups. `http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, 2007–2014.

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

[64] A. Narayanan. Tupperware: containerized deployment at Facebook. `http://www.slideshare.net/dotCloud/tupperware-containerized-deployment-at-facebook`, June 2014.

[65] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, pages 69–84, Farminton, PA, USA, 2013.

[66] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond Dominant Resource Fairness: extensions, limitations, and indivisibilities. In *Proc. Electronic Commerce*, pages 808–825, Valencia, Spain, 2012.

[67] Protocol buffers. `https://developers.google.com/protocol-buffers/`, and `https://github.com/google/protobuf/`., 2014.

[68] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, San Jose, CA, USA, Oct. 2012.

[69] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. European Conf. on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013.

[70] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, pages 3:1–3:14, Cascais, Portugal, Oct. 2011.

[71] E. Shmueli and D. G. Feitelson. On simulation and design of parallel-systems schedulers: are we doing the right thing? *IEEE Trans. on Parallel and Distributed Systems*, 20(7):983–996, July 2009.

[72] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 53:1–53:12, Austin, TX, USA, 2008.

[73] Apache Spark Project. `http://spark.apache.org/`, 2014.

[74] A. Tumanov, J. Cipar, M. A. Kozuch, and G. R. Ganger. Alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, San Jose, CA, USA, Oct. 2012.

[75] P. Turner, B. Rao, and N. Rao. CPU bandwidth control for CFS. In *Proc. Linux Symposium*, pages 245–254, July 2010.

[76] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, Santa Clara, CA, USA, 2013.

[77] VMware VCloud Suite. `http://www.vmware.com/products/vcloud-suite/`.

[78] A. Verma, M. Korupolu, and J. Wilkes. Evaluating job packing in warehouse-scale computing. In *IEEE Cluster*, pages 48–56, Madrid, Spain, Sept. 2014.

[79] W. Whitt. Open and closed models for networks of queues. *AT&T Bell Labs Technical Journal*, 63(9), Nov. 1984.

[80] J. Wilkes. More Google cluster data. `http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html`, Nov. 2011.

[81] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars. HaPPy: Hyperthread-aware power profiling dynamically. In *Proc. USENIX Annual Technical Conf. (USENIX ATC)*, pages 211–217, Philadelphia, PA, USA, June 2014. USENIX Association.

[82] Q. Zhang, J. Hellerstein, and R. Boutaba. Characterizing task usage shapes in Google's compute clusters. In *Proc. Int'l Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2011.

[83] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proc. European Conf. on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013.

[84] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 1393–1404. VLDB Endowment Inc., Sept. 2014.

Large-scale cluster management at Google with Borg: errata

2015-04-23

After the camera-ready copy was finalized, we noticed a few inadvertent omissions and ambiguities.

The user perspective

SREs do much more than system administration: they are the engineers responsible for Google's production services. They design and implement software, including automation systems, and manage applications, service infrastructure and platforms to ensure high performance and reliability at Google scale.

Related work

Borg leveraged much from its internal predecessor, the Global Work Queue system, which was initially developed by Jeff Dean, Olcan Sercinoglu, and Percy Liang.

Condor [1] has been widely used for aggregating collections of idle resources, and its ClassAds mechanism [2] supports declarative statements and automated attribute/property matching.

Acknowledgements

We accidentally omitted Brad Strand, Chris Colohan, Divyesh Shah, Eric Wilcox, and Pavanish Nirula.

References

[1] Michael Litzkow, Miron Livny, and Matt Mutka. "Condor - A Hunter of Idle Workstations". In *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 104-111, June 1988.

[2] Rajesh Raman, Miron Livny, and Marvin Solomon. "Matchmaking: Distributed Resource Management for High Throughput Computing". In *Proc. Int'l Symp. on High Performance Distributed Computing (HPDC)*, Chicago, IL, USA, July 1998.