使用 Borg 在 Google 管理大规模集群

Abhishek Vermay, Luis Pedrosaz, Madhukar Korupolu
David Oppenheimer, Eric Tune, John Wilkes
Google Inc.

译者：难易（Simpson）
修订：Ying 2017-11，2018-06

https://ai.google/research/pubs/pub43438 或
https://pdos.csail.mit.edu/6.824/papers/borg.pdf
译文 https://ying-zhang.github.io/doc/EuroSys15_Borg_CN_Ying_201806.pdf

摘要

Google 的 Borg 系统是一个集群管理器。它在多个万台机器规模的集群上运行着来自几千个不同应用的几十万个作业。

Borg 通过准入控制、高效的任务装箱、超售、机器共享、以及进程级别的性能隔离，实现了高利用率。它为高可用应用提供了可以减少故障恢复时间的运行时特性，以及降低关联故障概率的调度策略。Borg 提供了声明式的作业描述语言、域名服务集成、实时作业监控、分析和模拟系统行为的工具。这些简化了用户的使用。

本文介绍了 Borg 系统架构和特性，重要的设计决策，对某些策略选择的定量分析，以及十年来的运营经验和教训。

1 简介

我们内部称为 Borg 的集群管理系统，负责接收、调度、启动、重启和监控 Google 所有的应用。本文介绍它是如何实现的。

Borg 提供了三个主要的好处：（1）隐藏资源管理和故障处理细节，使用户可以专注于应用开发；（2）高可靠和高可用的运维，并支持应用程序也能够如此；（3）让我们可以在几万台机器上高效地运行负载。Borg 不是第一个涉及这些问题的系统，但它是少有的运行在如此大规模，具有弹性且完整的系统之一。

本文围绕这些主题来编写，总结了十多年来我们在生产环境运行 Borg 的一些定性观察。

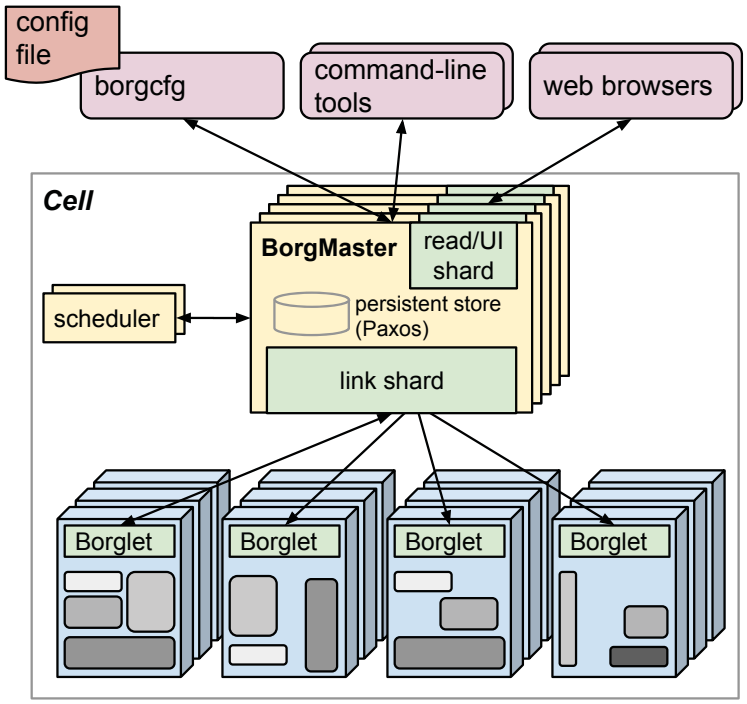


图 1. Borg 的架构。图中只画出了数千个工作节点的很小一部分

2 用户视角

Borg 的用户是 Google 的开发人员以及运行 Google 应用和服务的系统管理员（站点可靠性工程师，SRE）。用户以作业（Job）的方式将他们的工作提交给 Borg。作业由一个或多个任务（Task）组成，每个任务执行相同的二进制程序。每个作业只运行在一个 Borg 单元（Cell）里。Cell 是一组机器的管理单元。下面的小节将介绍用户视角看到的 Borg 系统的主要特性。

> SRE 的职责比系统管理员多得多：他们是负责 Google 生产服务的工程师。他们也设计和实现包括自动化系统等软件，管理应用、服务基础设施和平台，以保证在 Google 如此大的规模下的高性能和高可靠性。

2.1 工作负载

Borg Cell 主要运行 2 种异构的工作负载。第一种是应该“永不”停止的长期运行的服务，处理持续时间较短但对延迟敏感的请求（从几微秒到几百毫秒）。这些服务用于面向最终用户的产品，如 Gmail、Google Docs、网页搜索，以及内部基础设施服务（例如 Bigtable）。第二种是批处理作业，执行时间从几秒到几天，对短期性能波动不敏感。这 2 种负载在不同 Cell 中的比例不同，取决于其主要租户（例如，有些 Cell 就以批处理作业为主）。工作负载也随时间变化：批处理作业不断提交或结束，而很多面向终端用户的服务表现出昼夜周期性的使用模式。Borg 需要都处理好这些情况。

Borg 的代表性负载是一个公开的 2011 年 5 月整月的记录数据集 [80]。这个数据集已经得到了广泛的分析 [1, 26, 27, 57, 68]。

最近几年，以 Borg 为基础构建了很多应用框架，包括我们内部的 MapReduce 系统 [23]、FlumeJava[18]、Millwheel[3] 和 Pregel[59]。这些框架大多有一个控制器来提交 Master Job，还有多个 Worker Job。前两个框架类似于 YARN 的应用管理器 [76]。我们的分布式存储系统，例如 GFS[34] 和它的后继者 CFS、Bigtable[19]、以及 Megastore[8]，都是运行在 Borg 上的。

本文中，我们把高优先级的 Borg 作业称为为生产作业（prod），其它的则是非生产的（non-prod）。大多数长期服务是 prod 的，大部分批处理作业是 non-prod 的。一个典型 Cell 里，prod 作业分配了约 70% 的总 CPU 资源，占总 CPU 使用量约 60%；分配了约 55% 的总内存资源，占总内存使用量约 85%。§5.5节表明分配量和使用量的差异是值得注意的。

2.2 集群（Cluster）和单元（Cell）

一个 Cell 里的机器属于同一个集群。集群由数据中心级的高性能光纤的组网来定义。一个集群位于数据中心的一栋建筑内，而一个数据中心有多栋建筑[1]。一个集群通常包括一个大的 Cell，还可能有一些小规模的测试用或其它特殊用途的 Cell。我们尽力避免任何单点故障（译注：即不搞大型 Cell）。

不计测试用的 Cell，中等规模的 Cell 约有一万台机器；有些 Cell 还要大得多。Cell 中的机器从多个维度看都是异构的：大小（CPU、内存，硬盘，网络）、处理器类型、性能、以及是否有外网 IP 地址或 SSD 等。Borg 负责决定任务在 Cell 中的哪些机器上执行、为其分配资源、安装程序及依赖、监控健康状态并在失败后重启，从而使用户几乎不必关心机器异构性。

2.3 作业（Job）和任务（Task）

一个 Borg 作业的属性有：名称、拥有者和任务个数。作业可以有一些约束来强制其任务运行在有特定属性的机器上，比如处理器架构、操作系统版本、是否有外网 IP 地址等。约束可以是硬性的或者柔性的，柔性约束表示偏好，而非需求。一个作业可以推迟到前一个作业结束后再开始（译注：即作业间的依赖顺序）。一个作业只在一个 Cell 中运行。

每个任务对应着一组 Linux 进程，运行在一台机器上的一个容器内 [62]。绝大部分 Borg 的工作负载没有运行在虚拟机里，因为我们不想付出虚拟化的开销。而且，在 Borg 设计的时候，我们有很多处理器还没有硬件虚拟化功能呢。

任务也有一些属性，如资源需求量，在作业中的序号等。一个作业中的任务大多有相同的属性，但也可以被覆盖——例如特定任务的命令行参数。各维度的资源（CPU 核、内存、硬盘空间、硬盘访问速度、TCP 端口[2]等）。可以互相独立的以细粒度指定。我们不强制使用固定大小的资源桶或槽（见 §5.4）。

[1]这些关系会有少数例外情况

[2]Borg 负责管理一台机器上的可用端口并将其分配给任务

Borg 运行的程序都是静态链接的，以减少对运行环境的依赖，这些程序组织成由二进制文件和数据文件构成的包，由 Borg 负责安装。

用户通过向 Borg 发送 RPC 来控制作业。RPC 大多是从命令行工具、其它作业、或我们的监控系统（§2.6）发出的。大多作业描述文件使用一种声明式配置语言 BCL。BCL 是 GCL[12] 的一个变种，即增加了一些 Borg 专有的关键字，而 GCL 会生成若干 protobuf 文件 [67]。GCL 还提供了匿名函数以支持计算，这样就能让应用根据环境调整自己的配置。有上万个超过一千行的 BCL 配置文件，系统中累计有数千万行 BCL。Aurora 的配置文件与 Borg 的作业配置 [6] 类似。

图 2 展示了作业和任务整个生命周期的状态变化。

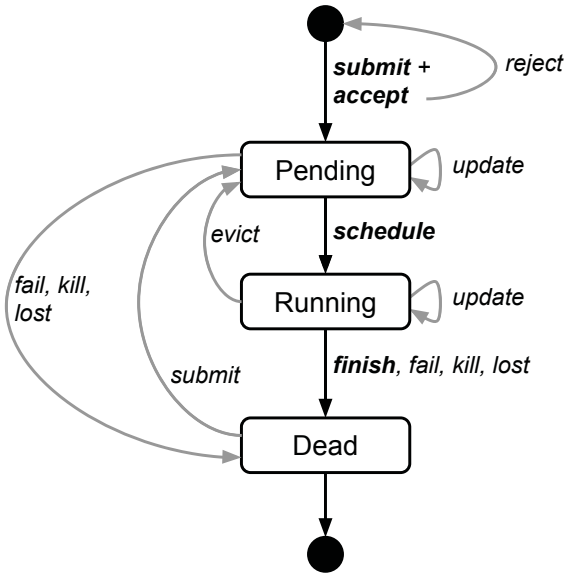


图 2. 作业和任务的状态图。用户可以触发提交，杀死和更新操作

要想在运行时改变一个作业中若干或全部任务的属性，用户可以向 Borg 提交一个新的作业配置，并命令 Borg 将任务更新到新的配置。更新是轻量的，非原子性的事务，在事务结束（提交）之前可以很容易地撤销。更新通常是滚动执行的，而且可以限制由更新导致的任务中断（被重新调度或抢占）的数量；超过限值后，变更将会被跳过。

一些任务更新（如更新二进制程序）需要重启任务；另外一些更新（如增加资源需求或修改约束）可能使该任务不适合运行在当前机器上，导致停止并重新调度该任务；还有一些更新（如修改优先级）总是可以进行的，不需要重启或者移动任务。

任务可以要求在被 Unix 的 `SIGKILL` 信号立即杀死之前获得 `SIGTERM` 信号通知，这样它们还有时间清理资源、保存状态、结束当前请求、拒绝新请求。但如果抢占者设置了延迟限值，就可能来不及发通知。实践中，80% 的情况下能发出通知信号。

2.4 分配（Allocs）

Borg 的 alloc（allocation 的缩写）是一台机器上的预留资源，可以用来执行一个或多个任务；不管有没有被使用，这些资源都算分配出去了。Allocs 可以给将来的任务预留资源，或在任务暂停和重启的间隔保持资源，以及将不同作业的多个任务绑定在同一台机器上——例如一个 Web 服务器实例和附加的将其 URL 日志从本机硬盘拷贝到分布式文件系统的日志转存任务。Alloc 像一台机器那样来管理；运行在同一个 Alloc 内的多个任务共享其资源。如果一个 Alloc 需要迁移到其它机器上，那么它的任务也要跟着重新调度。

一个 Alloc 集合，即一组在多台机器上预留了资源的 Alloc，类似于一个作业。一旦创建了一个 Alloc 集合，就可以向其提交若干作业。简便起见，我们用**任务**表示一个 Alloc 或者一个顶层任务（即运行在 Alloc 之外的任务），用**作业**表示一个普通作业或者 Alloc 集合。

2.5 优先级、配额和准入控制

当出现超过系统容量的工作负载会产生什么情况？我们对此的解决方案是优先级和配额。

每个作业都有一个小的正整数表示的优先级。高优先级的任务可以优先获得资源，甚至抢占（杀死）低优先级的任务。Borg 为不同用途定义了不重叠的优先级区间，包括（优先级降序）：**监控、生产、批处理、尽力（即测试的或免费的）**。本文中，prod 作业的优先级包括监控和生产两个区间。

虽然一个被抢占的任务通常会被重新调度到 Cell 的其它机器上，但级联抢占也可能发生：如果某个任务抢占了一个优先级稍低的任务，而后者又抢占了另一个优先级稍低的，如此继续。为避免这种情况，我们禁止生产区间的任务互相抢占。细粒度（译注：相比于区间的粗粒度）的优先级在其它场景下也很有用——如 MapReduce 的 Master 任务的优先级比其管理的 Worker 高一点，以提高其可靠性。

优先级表示了 Cell 中运行或等待的作业之间的相对重要性。配额（Quota）则用来决定准许哪个作业可以被调度。配额是特定优先级和时间段（典型是几个月）的一个资源向量（CPU，内存，硬盘等）。配额限制了用户的作业一次可以申请资源的最大数量（如：20TB 内存，prod 优先级，从现在到 7 月末，在 xx Cell 内）。配额检查是准入控制的一部分，而不是调度的：配额不足的作业提交时当即就会被拒绝。

高优先级的配额比低优先级的成本要高。生产级的配额限定于一个 Cell 的物理资源量。因此，用户提交了不超过配额的生产级作业时，不考虑资源碎片和约束，可以预期这个作业一定会运行。尽管我们鼓励用户不要购买超过其需求的配额，但很多用户仍然超买了，这样他们就不用担心由于将来应用用户量增长可能导致的配额短缺。我们的应对方案是对低优先级资源配额的超售：所有用户的 0 优先级配额是无限的，尽管这无法实现。低优先级的作业虽然被接收了，但可能由于资源不足而一直等待。

配额分配是 Borg 之外的系统处理的，与我们的物理容量规划紧密相关。容量规划的结果反映在各数据中心的价格和可用配额上。只有在其要求的优先级有足够的配额，用户的作业才能被接收。采用配额使得主导资源公平性（DRF）[29, 35, 36, 66] 这样的策略不是那么必要了。

Borg 的容量系统可以给某些用户一些特殊权限。例如，允许管理员删除或修改 Cell 里的任意作业，或者允许某个用户操作特定的内核特性或 Borg 行为（如对其作业禁用资源估计。§5.5）。

2.6 域名和监控

仅仅创建和部署任务是不够的：一个服务的客户端和其它系统需要能找到它们，即使该服务被重新部署到另一台机器之后。为实现该需求，Borg 为每个任务创建了一个固定的 BNS 域名（BNS，Borg name Service），这个域名包括了 Cell 名，作业名称和任务序号。Borg 把任务的主机名和端口写入 Chubby[14] 的一个持久化高可用文件里，以 BNS 域名为文件名。这个文件被 RPC 用来发现任务的实际地址。BNS 域名也是任务的 DNS 域名的一部分，例如，cc Cell 的 ubar 用户的 jfoo 作业的第 50 个任务可以通过 `50.jfoo.ubar.cc.borg.google.com` 来访问。每当状态改变时，Borg 还会把作业的大小和任务的健康信息写入到 Chubby，这样负载均衡器就知道如何路由请求了。

几乎每个任务都有一个内置的 HTTP 服务器，用来发布任务的健康信息和几千个性能指标（如 RPC 延时）。Borg 监控这些健康检查的 URL，重启那些没有立刻响应或返回 HTTP 错误码的任务。监控工具跟踪其它数据并显示在仪表盘上，当违反服务水平目标（SLO）时报警。

用户可以使用一个称为 Sigma 的 Web 界面来检查他的所有作业的状态，针对某个 Cell，或者深入某个作业及任务，检查其资源使用行为、详细日志、执行历史和最终结果。我们的应用产生大量的日志，它们都会被自动的滚动以避免耗尽硬盘空间。任务退出后，日志会保留一小段时间以帮助调试。如果一个作业没有运行起来，Borg 会提供一个挂起原因的标注，以及建议如何修改作业的资源请求，以使其更适合 Cell。我们发布了如何使资源请求更容易被调度的指南。

Borg 将所有的作业提交、任务事件、以及每个任务的详细资源使用都记录在 Infrastore 里。Infrastore 是一个可扩展的只读数据存储，通过 Dremel[61] 提供了类似 SQL 的交互式接口。这些数据用以支持基于使用量的收费，调试作业和系统故障，以及长期容量规划。公开的 Google 集群负载数据集 [80] 也来自于这些数据。

所有这些特性帮助用户理解和调试 Borg 及其作业的行为，并帮助我们的 SRE 实现每人管理超过上万台机器。

3 Borg 架构

一个 Borg 的 Cell 包括一组机器，一个逻辑上集中的控制器，称为 Borgmaster，以及运行在每台机器上的称为 Borglet 的代理进程（见图 1）。Borg 的组件都是用 C++ 实现的。

3.1 Borgmaster

Cell 的 Borgmaster 由两个进程组成：Borgmaster 主进程和一个单独的调度进程（§3.2）。Borgmaster 主进程处理客户端的 RPC，包括修改状态（如创建作业），或提供只读数据（如查找作业）。它还管理着系统中所有对象（机器、任务、Allocs 等）的状态，与 Borglet 通信，并提供一个 Web UI（作为 Sigma 的备份）。

Borgmaster 在逻辑上是单个进程，但实际上有 5 个副本。每个副本在内存维护着 Cell 状态的拷贝，该状态同时保存在由这些副本的本地硬盘组成的一个基于 Paxos[55] 的高可用、分布式存储上。每个 Cell 中仅有一个选举出来的 Master，它同时作为 Paxos 的 Leader 和状态修改者，处理所有变更 Cell 状态的请求，例如提交作业或者结束某台机器上的一个任务。当 Cell 启动或者上一个 Master 故障时，新的 Master 会通过 Paxos 算法选举出来；新 Master 会获取一个 Chubby 锁，这样其它的系统就可以找到它。选举并转移到新的 Master 通常需要 10 秒，但在大的 Cell 里可能需要长达 1 分钟，因为需要重构一些内存状态。当一个副本从宕机恢复后，它会动态地从其它最新的 Paxos 副本中重新同步自己的状态。

某个时刻的 Borgmaster 状态被称为检查点（Checkpoint），以定期快照加变更日志的形式保存在 Paxos 存储里。检查点有很多用途：如重建过去任意时刻的 Borgmaster 状态（例如，在接收一个触发了 Borg 故障的请求之前的时刻，这样就可以用来调试）；特别情况下可以手工修复检查点；构建一个持久的事件日志供日后查询；或用于离线仿真。

一个高保真的 Borgmaster 模拟器，称为 Fauxmaster，可以读取检查点文件。Fauxmaster 的代码拷贝自线上的 Borgmaster，还有对 Borglet 的存根接口。它接收 RPC 来改变状态，执行操作，例如“调度所有等待的任务”。我们用它来调试故障，像跟在线的 Borgmaster 那样与模拟器交互，用模拟的 Borglet 重放检查点文件里的真实交互。用户可以单步执行并观察系统过去确实发生了的状态变化。Fauxmaster 也用于容量规划（可以接收多少个此类型的作业？），以及在实际更改 Cell 配置前做可行性检查（这个变更会导致关键作业异常退出吗？）

3.2 调度

当提交一个作业后，Borgmaster 会把它保存在持久的 Paxos 存储上，并将这个作业的所有任务加入等待队列中。调度器异步地扫描等待队列，将任务分配到满足作业约束且有足够资源的机器上（调度是针对任务的，而非作业）。队列扫描从高优先级到低优先级，同优先级则以轮转的方式处理，以保证用户间的公平，并避免队首的大型作业阻塞队列。调度算法有两个部分：**可行性检查**，找到一组可以运行任务的机器；**评分**，从中选择一个合适的机器。

在可行性检查阶段，调度器会找到一组满足任务约束且有足够可用资源的机器——可用资源包括已经分配给低优先级任务但可以抢占的资源。在评分阶段，调度器确定每台可行机器的适宜性。评分考虑了用户特定的偏好，但主要取决于内建的标准：例如最小化被抢占任务的个数和优先级，选择已经有该任务安装包的机器，尽可能使任务分散在不同的供电和故障域，以及装箱（Packing）质量（在单台机器上混合高、低优先级的任务，以允许高优先级任务在负载尖峰扩容）等。

Borg 早期使用修改过的 E-PVM[4] 算法来评分。这个算法对异构的资源生成等效的成本值，放置任务的目标是使成本的变化量最小。在实践中，E-PVM 会把负载分散到所有机器，为负载尖峰预留出资源——这样的代价是增加了碎片，特别是对需要大部分机器的大型任务而言；我们有时称其为“最差匹配”。

与之相反的是“最佳匹配”，把机器上的任务塞的越满越好。这就“空”出一些没有用户作业的机器（它们仍运行存储服务），这样放置大型任务就比较直接了。但是，如果用户或 Borg 错误估计了资源需求，紧实的装箱会对此造成（性能上的）惩罚。这种策略不利于有突发负载的应用，而且对申请少量 CPU 的批处理作业特别不友好，这些作业申请少量 CPU 本来是为了更容易被调度执行，并抓住机会使用空闲资源：20% 的 non-prod 任务申请少于 0.1 个 CPU 核。

我们目前的评分模型是混合的，试图减少搁浅（Stranded）的资源（指某些类型资源分配完之后，一台机器上无法分配的其它类型资源）。对我们的负载而言，这个模型比“最佳匹配”提升了 3%-5% 的装箱效率（以 [78] 定义的方式评价）。

如果评分后选中的一台机器仍没有足够的资源来运行新任务，Borg 会抢占低优先级的任务，从最低优先级向上逐级抢占，直到资源足够运行该任务。被抢占的任务放回到调度器的等待队列里，而不

是被迁移或休眠[3]。

任务的启动延迟（从提交作业到任务开始运行之间的时间段）是我们持续重点关注的。这个时间差别很大，中位数约 25 秒。安装软件包耗费了其中 80% 的时间：一个已知的瓶颈就是软件包写入时对本地硬盘的竞争。为了减少任务启动时间，调度器偏好将任务分配到已经有必需的软件包（程序及数据）的机器：大部分包是只读的，所以可以被共享和缓存（这是 Borg 调度器唯一的一种数据局部性支持）。另外，Borg 通过树形和类似 BT 的协议并发地将软件包分发到多个机器上。

此外，调度器采用多种技术使其能够扩展到数万台机器的 Cell（§3.4）。

3.3 Borglet

Borglet 是部署在 Cell 每台机器上的本机 Borg 代理。它负责启动和停止任务；重启失败的任务；通过 OS 内核设置来管理本地资源；滚动调试日志；把本机的状态上报给 Borgmaster 和其它监控系统。

Borgmaster 每过几秒就会轮询每个 Borglet 来获取机器的当前状态，并向其发送请求。这让 Borgmaster 能控制通信频率，省去了显式的流量控制机制，而且防止了恢复风暴 [9]。

选举出来的 Master 负责准备发送给 Borglet 的消息，并根据 Borglet 的响应更新 Cell 的状态。为使性能可扩展，每个 Borgmaster 副本会负责一个无状态的链接分片（Link Shard）来处理部分 Borglet 的通信；Borgmaster 选举后重新计算链接的分片。为了保证容错（Resiliency），Borglet 总是汇报全部状态，但是 Link Shard 只汇报变化值，从而聚合、压缩这些信息，减少 Master 更新的负担。

如果某个 Borglet 几次没有响应轮询请求，该机器会被标记为宕机，其上运行的所有任务会被重新调度到其它机器。如果通讯恢复了，Borgmaster 会让这个 Borglet 杀掉已经被重新调度出去的任务，以避免重复。即便无法与 Borgmaster 通信，Borglet 仍会继续正常运行。所以即使所有的 Borgmaster 都出故障了，正在运行的任务和服务还会保持运行。

3.4 扩展性

我们还没有遇到 Borg 这种集中式架构的终极扩展上限。我们顺利地突破了遇到的每个限制。一个单独的 Borgmaster 可以管理有数千台机器的 Cell，有些 Cell 每分钟有 10000 多个任务到达。一个繁忙的 Borgmaster 使用 10 ～14 个 CPU 核以及 50GB 内存。我们用了几项技术来实现这种扩展性。

早期版本的 Borgmaster 使用一个简单的，同步的循环来处理请求、调度任务，并与 Borglet 通信。为了处理更大的 Cell，我们把调度器分离为一个单独的进程，这样它就可以与其它的 Borgmaster 功能并行执行，而这些其它的功能有多副本以便容错。调度器使用一份缓存的 Cell 状态拷贝，重复执行下面的操作：从选举出来的 Master 获取状态变更（包括已分配的和等待中的工作）；更新自己的本地拷贝；执行一轮调度来分配任务；将分配信息发送给 Master。Master 会接受并应用这些分配，但如果分配不适合（例如，是基于过时的状态做出的），就会等待调度器的下一轮调度。这与 Omega[69] 使用的乐观并发控制思路很相似，而且我们最近还给 Borg 添加了对**不同负载类型使用不同调度器**的功能。

为了改进响应时间，Borglet 使用独立的线程分别进行通信和响应只读 RPC。为了更好的性能，我们将这些请求划分给 5 个 Borgmaster 副本（§3.3）。总的效果是，UI 响应时间的 99% 分位数小于 1 秒，而 Borglet 轮询间隔的 95% 分位数小于 10 秒。

一些提高 Borg 调度器扩展性的方法如下：

缓存评分：计算一台机器的可行性和评分是比较昂贵的，所以 Borg 会一直缓存评分，直到这台机器或者任务的属性发生了变化——例如，这台机器上的某个任务结束了，一些属性修改了，或者任务的需求改变了。忽略小额的资源变化可以减少缓存失效。

任务等效类（Equivalence classes）：一般来说，同一个 Borg 作业的任务有相同的请求和约束。任务等效类即一组有相同需求的任务。Borg 只对等效类中的一个任务进行可行性检查和评分，而不是对等待的每个任务去检查一遍所有机器的可行性并对可行的机器评分。

适度随机：在一个大的 Cell 中，对所有机器都去计算一遍可行性和评分是很浪费的。调度器会随机地检查机器，直到找到足够多的可用机器来评分，然后从中挑选出最好的一个。这减少了任务启动和退出所需的评分次数及导致的缓存失效，加快了任务分配过程。适度随机有点类似 Sparrow[65] 的批量采样技术（译注：Sparrow 的批量采样考虑的是机器上的任务队列长度），但 Borg 还处理了优先级、抢占、异构性和安装软件包的成本。

在我们的实验中（§5），从零开始调度整个 Cell 的工作负载只要几百秒，但禁用上面几项技术的话，3 天都不够。正常情况下，半秒之内就能完成一遍等待队列的在线调度。

[3] 例外情况是，为 Google Compute Engine 提供虚拟机的任务会被迁移

4 可用性

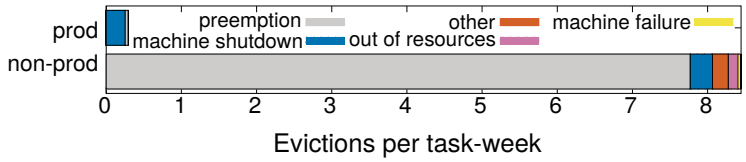


图 3. 不同类型任务的异常退出率及原因

包括抢占、资源不足、机器故障、机器关机、其它。数据从 2013-08-01 开始

大型系统里故障是很常见的 [10, 11, 12]。图 3 展示了在 15 个样本 Cell 里任务异常退出的原因分类。在 Borg 上运行的应用需要能处理这种事件，可采用的技术有多副本、保存持久状态到分布式存储，或定期快照（如果可行的话）等。当然，我们也尽可能的缓解异常事件的影响。例如，Borg 提供了：

- 自动重新调度异常退出的任务，如果必要，可以放置到另一台机器上去运行
- 把一个作业的任务分散到不同的可用域，例如机器、机架、供电域层次，以减少关联失效
- 在机器/OS 升级等维护活动期间，限制任务受影响的速率，以及同一作业中同时中止的任务的个数
- 使用声明式的预期状态表示，及幂等的变更操作，这样故障的客户端可以无损地重复提交故障期间漏掉的请求
- 对于失联的机器上的任务，限制重新调度的速率，因为大规模的机器故障和网络分区是很难区分的
- 回避造成崩溃的 < 任务，机器 > 组合
- 通过不断重新执行日志保存任务（§2.4），恢复已写入本地硬盘的关键中间数据，就算这个日志关联的 Alloc 已经终止或调度到其它机器上了。用户可以设置系统持续重复尝试多久，通常是几天时间。

Borg 的一个关键设计特性是：就算 Borgmaster 或者 Borglet 退出了，已经运行的任务还会继续运行下去。不过，保持 Master 正常运行仍然重要，因为在它退出后就无法提交新的作业，无法更新运行作业的状态，也不能重新调度故障机器上的任务。

Borgmaster 使用多项的技术支持其获得 99.99% 的实际可用性：多副本应对机器故障；准入控制应对过载；使用简单、底层的工具部署实例，以减少外部依赖。Cell 彼此是独立的，减少了关联误操作和故障传播的机会。同时这也是我们不扩大 Cell 规模的主要考虑，而并非是受限于扩展性。

5 利用率

Borg 的一个主要目标就是有效地利用 Google 的大量机器（这是一大笔财务投资）：让效率提升几个百分点就能省下几百万美元。这一节讨论和评估了一些 Borg 使用的策略和技术。

5.1 评估方法

作业有部署约束，而且需要处理负载尖峰（尽管比较少见）；机器是异构的；我们回收服务型作业的资源来运行批处理作业。因此，我们需要一个比“平均利用率”更高级的指标来评估我们的策略选择。大量实验后，我们选择了 Cell 压缩量（Compaction）：给定一个负载，我们不断地移除机器，直到无法容纳该负载，从而得知所需最小的 Cell 规模。从空集群开始部署该负载并重复多次，以减少特殊情况的影响。终止条件是明确的，对比可以自动化，避免了生成和建模合成负载的陷阱 [31]。[78] 提供了评估技术的定量比较，其中的细节非常微妙。

我们不可能在线上 Cell 进行实验，但是我们用了 Fauxmaster 来获得高保真的模拟效果，它使用了真实生产 Cell 和负载的数据，包括所有约束、实际限制、预留和使用量数据（§5.5）。实验数据提取自 2014-10-01 14:00 PDT 的 Borg 快照（其它快照也有类似的结论）。我们首先排除了特殊用途的、测试用的、小型的（少于 5000 台机器）的 Cell，然后从剩下的 Cell 中选取了 15 个样本，抽样尽量关于 Cell 的大小均匀分布。

为了保持机器异构性，在 Cell 压缩实验中，我们随机地移除机器。为了保持工作负载的异构性，我们保留了所有负载（除了那些绑定到特定机器的服务和存储任务，如 Borglet）。我们把那些需要超过原 Cell 大小一半的作业的硬性限制改成柔性的，允许不超过 0.2% 的任务一直等待，这是针对一些特别“挑剔”的，只能放置在很少的特定机器上的任务；充分的实验表明结果是可复现的，波动很小。如果需要一个大型的 Cell，就把原 Cell 复制几倍；如果需要更多的 Cell，也是复制原 Cell。

每个实验都用不同的随机数种子对每个 Cell 重复了 11 次。图中，我们用误差线线来表示所需机器数量的最大和最小值，选择 90% 分位数作为结果——平均值或中位数不能反映系统管理员所期望的充分把握。我们认为 Cell 压缩率是一个公平一致的比较调度策略的方法，而且可以直接转化为成本/收益的结果：更好的策略只需要更少的机器来运行相同的负载。

我们的实验关注于即时的调度（装箱），而不是重放一段长时间的负载记录。部分原因是避免处理开放或闭合的队列模型 [71, 79] 的困难；部分是传统的完成时间不适用于长时间运行的服务；部分是这样可以提供明确的比较结果；部分是因为我们认为不会对结果产生显著影响；还有部分现实原因，我们发现一次实验使用了 20 万个 Borg CPU 核——即便对 Google 而言，这个成本也不是个小数目。

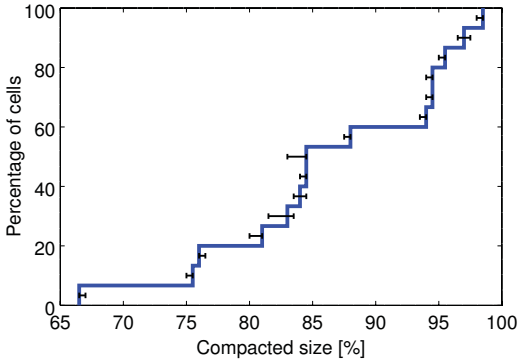
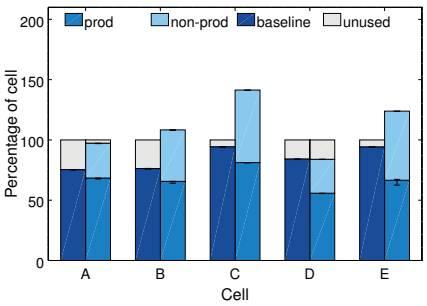


图 4. 压缩的效果。15 个 Cell 在压缩后相比原规模的百分比累积分布（CDF）

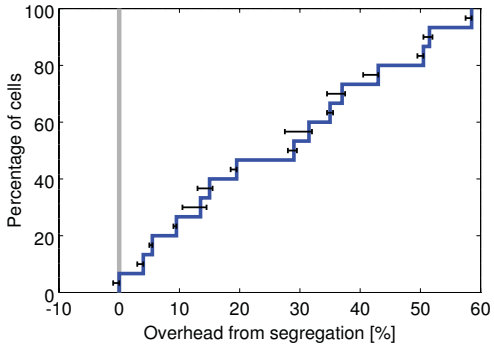
生产环境中，我们特意保留了一些裕度（Headroom），以应对负载增长、偶然的“黑天鹅”事件、负载尖峰、机器故障、硬件升级、以及大范围的局部故障（如供电母线短路）。图 4 显示了如果应用 Cell 压缩，实际的 Cell 可以压缩到多小。下文的图使用这些压缩后的大小作为基准值。

5.2 Cell 共享

几乎所有的机器都同时运行 prod 和 non-prod 的任务：在共享的 Cell 里是 98% 的机器，在所有 Borg 管理的机器里是 83%（有一些是 Cell 专用的）。



(a) 每个Cell中，左侧的条柱是初始大大小及负载组合；右侧是分开运行的情况



(b) 若将15个代表性Cell的负载分开运行，需要额外增加机器的累积分布（CDF）

图 5. 将 prod 和 non-prod 工作划分到不同的集群将需要更多的机器。
两幅图中的百分比都是相对于单个集群所需机器的最少数量而言的

鉴于很多外部组织将面向用户的作业和批处理作业分别运行在不同的集群上，我们检查一下如果我们也这么做会怎样。图 5 表明，在一个中等大小的 Cell 上，分开运行 prod 和 non-prod 的工作负载将需要增加 20-30% 的机器。这是因为 prod 的作业通常会保留一些资源来应对极少发生的负载尖峰，但

大多情况下用不到这些资源。Borg 回收了这些用不到的资源（§5.5），来运行 non-prod 的工作，所以总体我们只需要更少的机器。

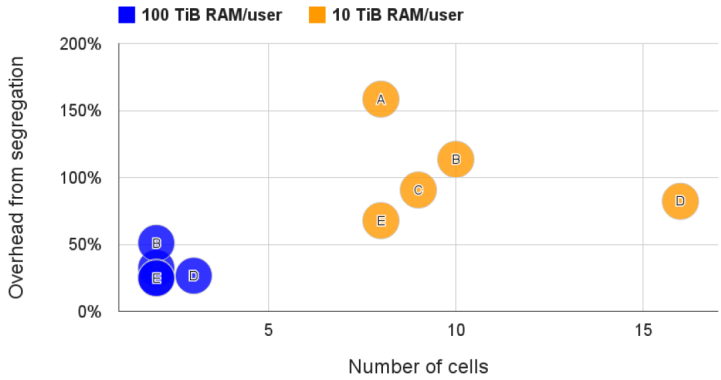


图 6. 将用户分开到不同的集群也会需要更多的机器

大部分 Borg Cell 被数千个用户共享使用。图 6 展示了为什么要共享。测试中，如果一个用户消费了超过 10TiB（或 100TiB）的内存，我们就把这个用户的工作负载分离到另一个 Cell 中。我们目前的共享策略是有效的：即使 100TiB 的阈值，也需要 2-16 倍的 Cell，增加 20-150% 的机器。将资源池化再次显著地节省了成本。

但是，把很多不相关的用户和作业类型放置到同一台机器上可能会造成 CPU 冲突，我们是否需要更多的机器来补偿？为评估这一点，我们看一下固定机器类型和时钟频率，任务的 CPI（Cycles per Instruction，执行每条指令平均所需时钟数，越大则程序执行越慢）在其它环境条件不同的影响下是如何变化的。在这种实验条件下，CPI 是一个可比较的指标，而且可以表征性能冲突，因为 2 倍的 CPI 意味着一个 CPU 密集型程序需要 2 倍的执行时间。数据是在一周内从约 12000 个随机选择的 prod 任务获取的，使用了 [83] 中介绍的硬件剖析工具记录 5 分钟内的时钟数和指令数，并调整了采样的权重，使 CPU 时间的每秒都均等处理。结果并非直截了当的：

（1）我们发现 CPI 在同一个时间段内和下面两个量正相关：这台机器上总的 CPU 使用量，以及这个机器上同时运行的任务个数（基本上独立）；每向一台机器上增加一个任务，就会使其它任务的 CPI 增加 0.3%（从数据拟合的线性模型给出的预测值）；将一台机器的 CPU 使用量增加 10%，就会增加 2% 弱的 CPI。尽管相关性在统计意义上是显著的，也只是解释了 CPI 变化的 5%。还有其它的因素，支配着 CPI 的变化，例如，应用程序固有的差别，以及特殊的干扰模式 [24, 83]。

（2）比较从共享 Cell 和只运行几种应用的少数专用 Cell 获取的 CPI 采样，我们看到共享 Cell 里的 CPI 平均值为 1.58（σ=0.35，标准差），专用 Cell 的 CPI 平均值是 1.53（σ=0.32）——也就是说，共享 Cell 的性能差 3%。

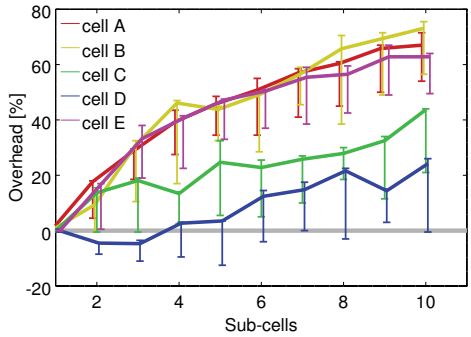
（3）为了搞定不同 Cell 的应用会有不同的工作负载，或者会有幸存者偏差（或许对冲突更敏感的程序会被挪到专用 Cell 里面去），我们观察了 Borglet 的 CPI。所有 Cell 的所有机器上都运行着 Borglet。我们发现专用 Cell 里 Borlet 的 CPI 平均值是 1.20（σ=0.29），而共享 Cell 里的 CPI 平均值为 1.43（σ=0.45），表明在专用 Cell 上比在共享 Cell 上快 1.19 倍，不过这个结果忽略了专用 Cell 中的机器负载较轻的因素，即稍偏向专用 Cell。

这些实验表明了仓库级别的性能比较是复杂的，强化了 [51] 中的观察，但也说明共享并没有显著增加运行程序的开销。

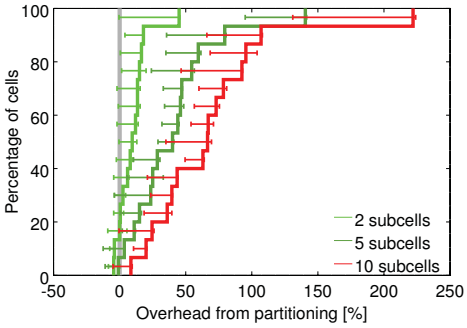
不过，就算从结果中最差的数据来看，共享还是有益的：比起 CPU 的降速，共享比各个划分方案都减少了机器，这一点更重要，而且共享的收益适用于包括内存和硬盘等各种资源，不仅仅是 CPU。

5.3 大型 Cell

Google 建立了大型 Cell，一是为了允许运行大型任务，二是为了减少资源碎片。为测试减少碎片的效果，我们把负载从一个 Cell 分散多个较小的 Cell 中——首先将作业随机排列，然后轮流分配到各小的 Cell 中。图 7 确认了使用小型 Cell 需要增加相当多的机器。



(a) 对5个不同的初始Cell，改成小型Cell后，随个数变化需要增加的机器



(b) 对15个不同的Cell，分别将其改成2，5 或 10个小Cell后需增加机器的累积分布（CDF）

图 7. 将 Cell 分成更小的规模将需要更多的机器

5.4 细粒度资源请求

Borg 用户请求的 CPU 单位是 0.001 个核，内存和硬盘的单位是字节。（一个核实际上是一个 CPU 的超线程，对不同机器类型的性能进行了标准化）。图 8 表明用户充分利用了细粒度：请求的 CPU 核和内存数量的“特别偏好值”是很少的，这些资源也没有明显的相关性。这与 [68] 里的分布非常相似，除了我们在 90% 分位数及以上的内存请求多一点之外。

尽管 IaaS 普遍只提供一组固定尺寸的容器或虚拟机 [7, 33]，但不符合我们的需求。为说明这一点，我们对 prod 的作业和 Alloc（§2.4）申请的 CPU 核和内存分别向上取整到最接近的 2 的幂，形成固定大小的“桶”，最小的桶有 0.5 个核和 1GiB 内存。图 9 显示一般情况下这样需要增加 30-50% 的资源。上限的情形是，有的大型任务即便将 Cell 扩大为未压缩尺寸的四倍也无法容纳，只好为其分配一整台机器。下限是允许这些任务一直等待。（这比 [37] 给出的将近 100% 的额外开销要小一些，因为我们支持不止 4 种尺寸的桶，而且允许 CPU 和内存分别改变）。

5.5 资源回收

作业可以声明一个资源**限额（Limit）**，是每个任务能获得的资源上限。Borg 会用它来检查用户是否有足够的配额来接受该作业，并检查某个机器是否有足够的可用资源来运行任务。因为 Borg 通常会杀死那些试图使用超出内存和硬盘申请值的任务，或者限制其 CPU 使用量不超过申请值，所以有的用户会为任务申请超过实际需要的资源，就像有的用户会购买超过实际需要的配额一样。另外，一些任务只是偶尔需要使用它们申请的所有资源（例如，在一天中的高峰期或者受到了拒绝服务攻击），但大多时候用不了。

与其把那些分配出来但暂时没有被用到的资源浪费掉，我们估计了一个任务会用多少资源，然后把剩余的资源回收给那些可以忍受低质量资源的任务，例如批处理作业。这整个过程称为**资源再利用**。这个估值称为任务的资源**预留（Reservation）**。Borgmaster 每隔几秒就会根据 Borglet 获取的细粒度资源使用量信息来计算一次预留值。最初的预留资源被设置为资源限额；在 300 秒之后，也就过了启动阶段，预留资源会缓慢的下降到实际使用量加上一个安全值。在实际使用量超过它时，预留值会迅速增加。

Borg 调度器使用资源限额来计算 prod 级任务[4]是否可以执行（§3.2），所以这些任务不依赖于回收的资源，也与资源超售无关；对于 non-prod 的任务，运行任务使用的资源在预留值之内，这样新任务就可以使用回收的资源。

一台机器有可能因为预留（预测）错误而导致运行时资源不足——即使所有的任务都在资源限额之内。如果这种情况发生了，我们会杀掉或者限制 non-prod 任务，但从来不对 prod 任务下手。

图 10 表明，如果没有资源回收，将需要更多的机器。在一个中等规模的 Cell 中大概有 20% 的工作负载（§6.2）使用了回收的资源。

图 11 可以看到更多的细节，其中有预留值、使用量与限额的比例。当资源紧张时，超出内存限额的任务首先会被抢占，不论优先级有多高，所以很少有任务超过内存限额。另一方面，CPU 使用量是可以被轻易限制住的，所以短时的毛刺虽然会导致使用量超过预留值，但这没什么损害。

[4]准确的说，是高优先级的、延迟敏感的任务，见 §6.2

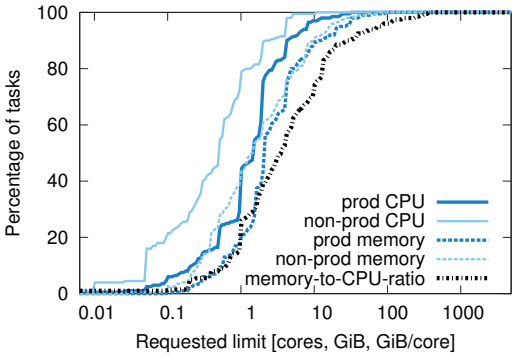


图8：没有一个桶的尺寸能适合所有任务。请求CPU和内存的CDF。没有突出值，尽管CPU核有几个整数更常见

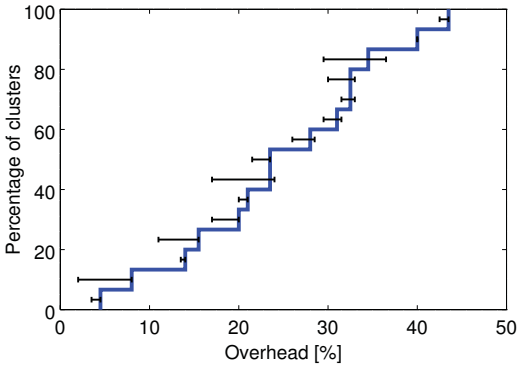


图10：资源回收非常有效。若禁用了，需要额外增加的机器的CDF

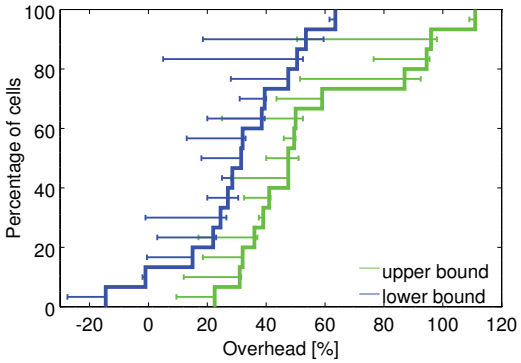


图9：将资源请求限制为特定的桶将需要更多的机器。若将CPU和内存请求向上取整到最接近的2的幂，需额外增加的负担的CDF。上下限超过了实际值（详见正文）

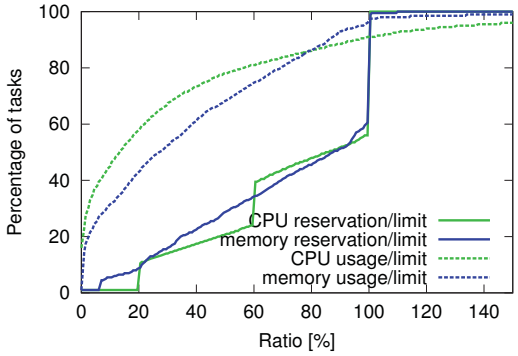


图11：资源估计可以成功识别空闲资源。虚线是CPU和内存的使用量与申请值（限额）之比的CDF。大部分任务的使用量比申请值少得多，尽管有少数使用的CPU比申请值多。实线是CPU和内存的预留值与申请值（限额）之比的CDF，该值更接近100%。直线段是资源估计过程

图 11 表明了资源回收可能还过于保守：在预留值和实际使用量中间还有一大段差距。为了测试这一点，我们选择了一个线上 Cell，（第一周作为参照基准，）第二周将其估计算法的参数调整为比较**激进**的设置，即把安全裕度留小一点；第三周采取的是介于激进和基准之间的**适度**策略，最后一周恢复到基准策略。

图 12 展示了结果。第二周的预留值明显更接近实际使用量，第三周稍大一点，最大的是第一周和第四周。和预期的一样，第二周和第三周的 OOM 比率轻微地增加了[5]。在评估了这个结果后，我们认为利大于弊，于是在其它 Cell 上也采用了**适度**策略的资源回收参数。

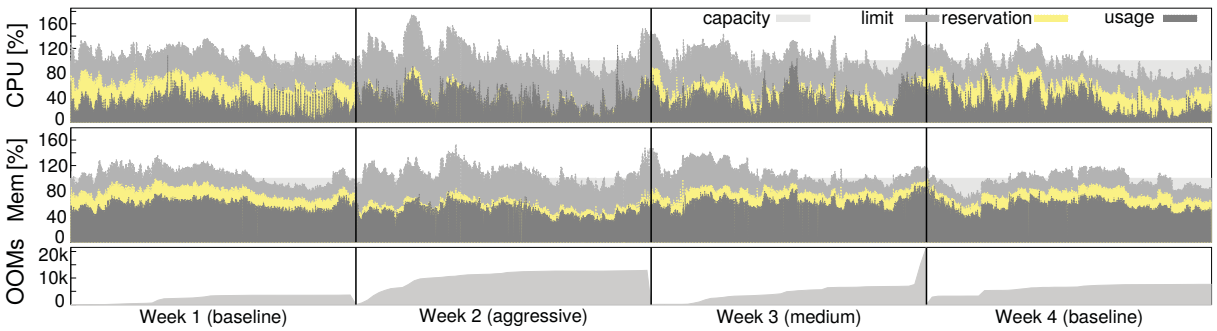


图 12. 更激进的资源估计可以回收更多资源，但会稍增加 OOM 事件

6 隔离

50% 的机器运行了 9 个以上的任务；处于 90% 分位数的机器则有大约 25 个任务，4500 个线程 [83]。虽然在应用之间共享机器会增加利用率，但也需要一个比较好的机制来保证任务之间不产生干

[5]第 3 周后期的异常情况与本次实验无关

扰。这同时适用于安全和性能两个方面。

6.1 安全隔离

我们使用 Linux 的 chroot 作为同一台机器上不同任务之间的主要安全隔离机制。仅当某台机器有用户运行的任务时，为了允许远程调试，我们以前会自动分发（或废除）SSH 秘钥，使用户可以访问这台机器。对大多数用户来说，现在被替换为 `borgssh` 命令，这个程序和 Borglet 协同构建一个 SSH 通道，连接到与任务运行在同一个 chroot 和 cgroup 下的 Shell，这样限制就更加严格了。

Google 的 AppEngine（GAE）[38] 和 Google Compute Engine（GCE）使用 VM 和安全沙箱技术运行外部的软件。我们把每个运行在 KVM 进程中的 VM 作为一个 Borg 任务来运行。

6.2 性能隔离

早期的 Borglet 使用了一种相对原始的资源隔离措施：事后检查内存、硬盘和 CPU 使用量，终止使用过多内存和硬盘的任务，或者降低使用过多 CPU 的任务的 Linux CPU 优先级。不过，一些粗暴的任务还是能很容易地影响到同台机器上其它任务的性能，于是有的用户就会多申请资源来让 Borg 减少与其共存的任务数量，降低了资源利用率。资源回收可以弥补一些损失，但不是全部，因为涉及到安全裕度。在极端情况下，用户会要求使用专属的机器或者 Cell。

目前，所有 Borg 任务都运行在基于 Linux cgroup 的资源容器 [17, 58, 62] 里。Borglet 控制着这些容器的设置。有了 OS 内核的帮助，控制能力得到了改善。即使这样，偶尔还是有底层的资源发生冲突（例如内存带宽或 L3 缓存污染），见 [60, 83]。

为了应对过载和超售，Borg 任务有一个应用类别（appclass）属性。最重要的区分是延迟敏感（LS）的应用和本文中称为批处理（batch）的其它类别。LS 任务包括面向用户的应用和需要快速响应的共享基础设施。高优先级的 LS 任务得到最高优待，可以暂时让批处理任务等待几秒种。

第二个区分是：**可压缩**资源（例如 CPU，硬盘 I/O 带宽），都是基于速率的，可以通过降低一个任务的服务质量而不是杀死它来回收；**不可压缩**资源（例如内存、硬盘空间），这些一般来说不杀掉任务是没办法回收的。如果一个机器用光了不可压缩资源，Borglet 马上就会开始杀死任务，从低优先级开始，直到能满足剩下的资源预留。如果机器用完了可压缩资源，Borglet 会限制使用量（偏好 LS 任务），这样不用杀死任何任务也能处理短期负载尖峰。如果情况没有改善，Borgmaster 会从这个机器上移除一个或多个任务。

Borglet 有一个用户态的控制循环，负责以下操作：为容器确定内存量，prod 任务基于预测值，而 non-prod 任务则基于内存压力；处理来自内核的 OOM 事件；当任务试图分配超过其自身限额的内存时，或者超售的机器上确实耗尽内存时，都会杀死任务。Linux 激进的文件缓存让我们的实现复杂得多，因为需要精确计算内存使用量。

为了增强性能隔离，LS 任务可以预留整个物理 CPU 核，以阻止别的 LS 任务来使用它们。批处理任务被允许运行在任何核上，但是相比 LS 任务，批处理任务只分配了很少的调度份额。Borglet 动态地调整贪婪的 LS 任务的资源上限，以保证它们不会把批处理任务饿上几分钟，必要时有选择的使用 CFS 带宽控制 [75]；仅用份额来表示是不够的，因为我们有多个优先级。

同 Leverich[56] 一样，我们发现标准的 Linux CPU 调度器（CFS）需要大幅调整才能同时支持低延迟和高利用率。为了减少调度延迟：我们内部版本的 CFS 对每个 `cgroup` 都有单独的负载历史 [16]；允许 LS 任务抢占批处理任务；当一个 CPU 有多个就绪的 LS 任务时，会减少其调度量（Quantum）。幸运的是，我们的大多应用使用一个线程处理一个请求的模型，这样就缓解了持续的负载失衡。我们节俭地使用 `cpusets` 为有特别严格的延迟需求的应用分配 CPU 核。这些努力的一些效果展示在图 13 中。我们持续在这方面投入，增加感知 NUMA、超线程、能耗（如 [81]）的线程放置和 CPU 管理，改进 Borglet 的控制精确度。

任务被允许在其上限之内消费资源。大部分任务还允许去使用超出上限的可压缩资源，例如 CPU，以利用空闲资源。只有 5% 的 LS 任务禁止这么做，主要是为了改善可预测性；小于 1% 的批处理任务也禁止了。使用超量内存默认是被禁止的，因为这会增加任务被杀掉的概率，不过即使这样，10% 的 LS 任务解除了这个限制，79% 的批处理任务也解除了，因为这是 MapReduce 框架的默认设置。这补偿了资源回收（§5.5）的后果。批处理任务很乐意使用空闲的或回收的内存：大多情况下这样运作得很好，即使偶尔批处理任务会被急需资源的 LS 任务杀掉。

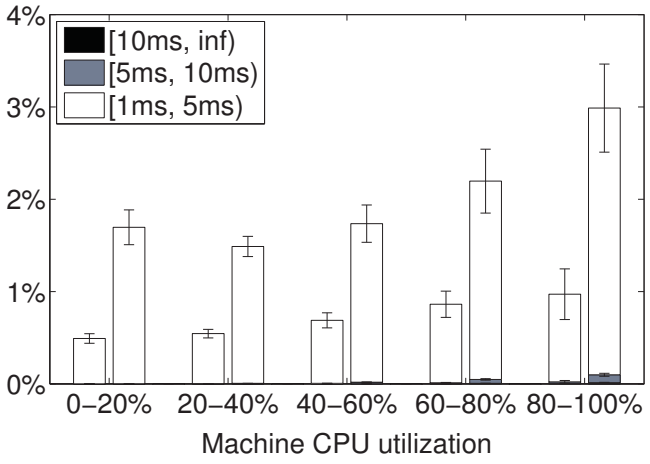


图 13. 调度延迟与负载的关系。即一个就绪线程需要等待超过 1 ms 才能运行的比率，与机器繁忙程度的关系。每组数据条中，左侧是延迟敏感的任务，右侧是批处理任务。只有很少的比率需要等 5 ms 以上，超过 10 ms 就极少了。这是 2013 年 12 月从一个代表性的 Cell 中获取的一个月的数据；误差线是每天的波动

7 相关工作

数十年来，资源调度已经在多种场景得到了研究，如广域高性能计算网格、工作站网络、和大规模服务器集群等。我们这里只关注最相关的大规模服务器集群这个场景。

最近的一些研究分析了来自于 Yahoo!、Google 和 Facebook 的集群记录数据 [20, 52, 63, 68, 70, 80, 82]，展现了这些现代的数据中心和工作负载固有的异构性和大规模带来的挑战。[69] 包含了对集群管理器架构的分类。

Apache Mesos[45] 将资源管理和任务放置功能拆分到一个集中资源管理器（类似于去掉调度器的 Bormaster）和多种“框架”（比如 Hadoop[41] 和 Spark[73]）之间，两者基于供应（Offer）机制交互。Borg 则把这些功能集中在一起，使用基于请求的机制，而且扩展性相当好。DRF[29, 35, 36, 66] 最初是为 Mesos 开发的；Borg 则使用优先级和准入配额来替代。Mesos 开发者已经宣布了他们扩展 Mesos 的雄心壮志：预测性资源分配和回收，以及解决 [69] 中发现的一些问题。

YARN[76] 是一个针对 Hadoop 的集群管理器。每个应用都有一个另外的管理器，与中央资源管理器谈判所需资源；这跟大约 2008 年开始 Google 的 MapReduce 作业已经使用的向 Borg 获取资源的模式如出一辙。YARN 的资源管理器最近才支持容错。一个相关的开源项目是 Hadoop Capacity Scheduler（基于容量的调度器）[42]，提供了多租户下的容量保证、多层队列、弹性共享和公平调度。YARN 最近扩展支持了多种资源类型、优先级、抢占和高级准入控制 [21]。Tetris（俄罗斯方块）研究原型 [40] 支持完成时间感知的作业装箱。

Facebook 的 Tupperware[64]，是一个在集群中调度 `cgroup` 容器的类 Borg 系统；只有少量细节披露出来了，看起来它也提供了某种形式的资源回收功能。Twitter 开源的 Aurora[5] 是一个类似 Borg 的，用于长期运行服务的调度器，运行与 Mesos 之上，其配置语言和状态迁移与 Borg 类似。

微软的 Autopilot[48] 为其集群提供了“自动化的软件供应和部署；系统监控；以及采取修复行为处理软硬件故障”的功能。Borg 生态系统提供了相似的特性，不过篇幅所限，不再深入讨论；作者 Isaard 概括了很多我们也赞成的最佳实践。

Quincy[49] 使用了一个网络流模型来提供公平性和数据局部性感知的调度，应用在几百个节点的集群的数据处理 DAG 上。Borg 使用配额和优先级在用户间共享数据，可以扩展到上万台机器。Quincy 可以直接处理执行图，而 Borg 需要在其上层另外构建。

Cosmos[44] 聚焦在批处理上，强调了用户可以公平获取他们已经捐献给集群的资源。每个作业分别有一个管理器来获取资源；只有很少公开的细节。

微软的 Apollo 系统 [13] 为每个短期批处理作业分别使用单独的调度器，以获得高吞吐量，其集群规模看起来与 Borg 的 Cell 相当。Apollo 投机地执行低优先级后台任务来提升资源利用率，代价是有时有长达多日的队列延迟。Apollo 的各节点都一个关于开始时间的预测矩阵，其行列分别为 CPU 和内存两个资源维度。调度器会综合开始时间、估计的启动开销、获取远程数据的开销来决定部署位置，

并用一个随机延时来减少冲突。Borg 使用的是中央调度器，基于之前的分配来决定部署位置，可以处理更多的资源维度，而且更关注高可用、长期运行的应用；Apollo 也许能处理比 Borg 更高的任务到达率。

阿里巴巴的伏羲（Fuxi）[84] 支持数据分析的应用，从 2009 年就开始运行了。类似 Borgmaster，一个集中的 FuxiMaster（也做了多副本容错）从节点上获取可用资源的信息、接受应用的资源请求，然后匹配两者。伏羲的增量调度策略与 Borg 的任务等价类是相反的：伏羲用最新的可用资源匹配等待队列里的任务（译注：Borg 是用任务匹配资源）。类似 Mesos，伏羲允许定义“虚拟资源”类型。只有对合成工作负载的实验结果是公开的。

Omega[69] 支持多个并发的调度器，粗略相当于没有持久存储和链接分片的 Borgmaster。Omega 调度器使用乐观并发控制的方式去操作一个共享的集群预期的和观察的状态表示。集群状态存储在一个集中持久存储中，用单独的连接组件与 Borglet 同步。Omage 架构设计为支持多种不同的工作负载，它们有自己特定的 RPC 接口、状态迁移和调度策略（例如长期运行的服务、多个框架批处理作业、如集群存储这样的基础服务、Google 云平台上的虚拟机）。相反，Borg 提供了一种通用方案，同样的 RPC 接口、状态迁移、调度策略，为支持多种不同的负载，其规模和复杂度逐渐增加，但目前来说可扩展性还不算一个问题（§3.4）。

Google 的开源项目 Kubernetes 系统 [53] 把应用放在 Docker 容器内 [28]，再分发到多个机器上。它既可以运行在物理机上（像 Borg 那样），也可以运行在多个云供应商（比如 Google Compute Engine，GCE）的主机上。Kubernetes 正在快速开发中，它的很多开发者也参与开发了 Borg。Google 提供了一个托管的版本，称为 Google Container Engine（GKE）[39]。我们会在下一节里面讨论 Kubernetes 从 Borg 中学到了哪些东西。

在高性能计算社区对这个领域有长期的研究传统（如 Maui, Moab, Platform LSF[2, 47, 50]）；但是这和 Google Cell 所面对的规模、工作负载、容错性是不同的。总体而言，为达到高用率，这些系统需要让任务在一个很长的队列中等待。

虚拟化供应商，例如 VMware[77]，和数据中心方案供应商，例如 HP 和 IBM[46] 提供了典型情况下可以扩展到一千台机器规模的集群管理解决方案。另外，一些研究小组的原型系统以多种方式提升了调度质量（如 [25, 40, 72, 74]）。

最后，正如我们所指出的，大规模集群管理的另外一个重要部分是自动化和无人化。[43] 指出，失效预案、多租户、健康检查、准入控制，以及可重启对实现单个运维人员管理更多的机器的目标是必要的。Borg 的设计哲学也是这样的，而且支撑了我们的每个 SRE 管理数万台机器。

> Borg 从它的前任继承了很多东西，即我们内部的全局工作队列（Global Work Queue）系统，它最初是由 Jeff Dean，Olcan Sercinoglu, 和 Percy Liang 开发的。
>
> Conder[85] 曾被广泛用于收集空闲资源，其 ClassAds 机制 [86] 支持声明式的语句和自动属性匹配。

8 经验教训和未来工作

在这一节中我们介绍了十多年来我们在生产环境运行 Borg 得到的定性的经验教训，然后介绍设计 Kubernetes[53] 是如何吸收这些经验的。

8.1 教训

我们从一些 Borg 作为反面警示的特性开始，然后介绍 Kubernetes 的替代方案。

将作业作为唯一的任务分组机制比较受限

Borg 没有内置的方法将多个作业组成单个实体来管理，或将相关的服务实例关联起来（例如，测试通道和生产通道）。作为一个技巧，用户把他们的服务拓扑编码到作业的名称中，然后构建了更高层的管理工具来解析这些名称。这个问题的另外一面是，没办法指向服务的任意子集，这就导致了僵硬的语义，以至于无法滚动升级或改变作业的实例数。

为了避免这些困难，Kubernetes 不再使用作业这个概念，而是用标签（Label）来组织它的调度单元（Pod）。标签是任意的键值对，用户可以对系统的任何对象打上标签。Borg 作业可以等效地通过对一组 Pod 打上 < 作业：作业名 > 这样的标签来实现。其它有用的分组方式也可以用标签来表示，例如服务、层级、发布类型（如，生产、就绪、测试）。Kubernetes 用标签查询的方式来选取待操作的目标对象。这样就比固定的作业分组更加灵活。

同一台机器的任务共用一个 IP 太复杂了

Borg 中，同一台机器上的所有任务都使用主机的同一个 IP 地址，共享端口空间。这就带来几个麻烦：Borg 必须把端口当做资源来调度；任务必须先声明它需要多少端口，而且需要支持启动时传入可用端口号；Borglet 必须强制端口隔离；域名和 RPC 系统必须像 IP 一样处理端口（译注：最后这一点我认为是必要的）。

多亏了 Linux 的 namespace、虚拟机、IPv6 和软件定义网络 SDN 的出现，Kubernetes 可以用一种更用户友好的方式来消除这些复杂性：每个 Pod 和 Service 都自己的 IP 地址，允许开发者选择端口，而不是让他们的软件支持从基础设施获得分配，这也消除了基础设施管理端口的复杂性。

给资深用户优化而忽略了初级用户

Borg 提供了一大堆针对“资深用户”的特性，这样他们就可以仔细地调节他们程序的运行方式（BCL 规范约有 230 个参数）：开始的目的是为了支持 Google 的大型资源用户，提升他们的效率会带来显著的效益。但不幸的是，这么复杂的 API 让初级用户用起来很复杂，而且限制了 API 的演进。我们的解决方案是在 Borg 上又做了一些自动化的工具和服务，从实验中决定合理的配置。由于应用支持容错，实验可以自由进行：即使自动化出了问题，也只是小麻烦，不会导致灾难。

8.2 经验

另一方面，有不少 Borg 的设计特性是非常有益的，而且经历了时间考验。

Alloc 是有用的

Borg 的 Alloc 抽象适用于广泛使用的保存日志模式（§2.4），另一个流行的模式是：一个简单的数据加载任务定期更新 Web 服务器使用的数据。Alloc 和软件包机制允许这些辅助服务由不同的小组开发。Kubernetes 对应于 Alloc 的概念是 Pod，它是对一个或多个容器的资源封装，其中的容器共享 Pod 的资源，而且总是被调度到同一台机器上。Kubernetes 使用 Pod 里的辅助容器来替代 Alloc 里面的任务，不过思路是一样的。

集群管理不只是任务管理

虽然 Borg 的主要角色是管理任务和机器的生命周期，但 Borg 上的应用还从其它的集群服务中收益良多，例如域名服务和负载均衡。Kubernetes 用 Service 这个抽象概念来支持域名服务和负载均衡：Service 有一个域名和用标签选出的多个 Pod 的动态集合。集群中的任何容器都可以通过 Service 域名连接到该服务。幕后，Kubernetes 自动将连接到该 Service 的负载分散到与其标签匹配的 Pod 之间，由于 Pod 挂掉后会被重新调度到其它机器上，Kubernetes 还会跟踪这些 Pod 的位置。

自省是至关重要的

虽然 Borg 总体上是工作良好的，但出了问题后，定位根本原因是非常有挑战性的。Borg 的一个关键设计选择是把所有的调试信息暴露给用户而不是隐藏起来：Borg 有几千个用户，所以“自助”是调试的第一步。虽然一些用户的依赖项让我们难以废弃一些特性或修改内部策略，但这还是成功的，我们还没找到其它实际的替代方式。为管理大量的数据，我们提供了多个层次的 UI 和调试工具，这样用户就可以快速定位与其作业相关的异常事件，深入挖掘来自其应用和基础设施本身的详细事件和错误日志。

Kubernetes 也计划引入 Borg 的大部分自省技术。和 Kubernetes 一起发布了很多工具，比如用于资源监控的 cAdvisor[15]，它基于 Elasticsearch/Kibana[30] 和 Fluentd[32] 聚合日志。Master 可以用来查询某个对象的状态快照。Kubernetes 提供了一致机制，所有可以记录事件的组件（例如，被调度的 Pod、出错的容器）都可以被客户端访问。

Master 是分布式系统的核心

Borgmaster 最初设计为一个单体的系统，随着时间发展，它演变成了一组服务生态系统的核心。用户作业管理的管理是由这些服务协同完成的。比如，我们把调度器和主要的 UI（Sigma）分离成单独的进程，增加了一组服务，包括准入控制、纵向和横向扩展、任务重新装箱、周期性作业提交（cron）、工作流管理，用于离线查询的系统活动归档等。总体而言，这让我们能扩展工作负载和特性集合，但无需牺牲性能和可维护性。

Kubernetes 的架构走的更远一些：它的核心是一个仅处理请求和操作底层状态目标的 API 服务。集群管理逻辑构建为一个小型的、可组合的微服务，作为 API 服务的客户端，如故障后仍维持 Pod 副本个数在期望值的副本管理器，以及管理机器生命周期的节点管理器。

8.3 总结

在过去十年间，所有几乎所有的 Google 集群负载都迁移到了 Borg 上。我们仍在持续改进它，并把经验应用到了 Kubernetes 上。

致谢

这篇文章的作者负责撰写文章，并完成了评估实验。几十位设计、实现和维护 Borg 组件和生态系统的工程师才是它成功的关键。我们在这里列出直接参与设计、实现和维护 Borgmaster 及 Borglet 的人员。如果有遗漏，我们深表歉意。

早期版本的 Borgmaster 设计和实现人员有：Jeremy Dion 和 Mark Vandevoorde，以及 Ben Smith, Ken Ashcraft, Maricia Scott, Ming-Yee Iu 和 Monika Henzinger。早期版本的 Borglet 主要是由 Paul Menage 设计和实现的（译注：见 [62]）。

后续的参与者包括：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, Zohaib Maya.

Borg SRE 团队也是非常重要的，包括：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, 和 Viraj Alankar。Borg 配置语言（BCL）和 `borgcfg` 工具最初是 Marcel van Lohuizen 和 Robert Griesemer 开发的。

我们不小心漏掉了 Brad Strand, Chris Colohan, Divyesh Shah, Eric Wilcox, 和 Pavanish Nirula。

谢谢我们的审稿人（尤其是 Eric Brewer, Malte Schwarzkopf 和 Tom Rodeheffer），以及我们的导师 Christos Kozyrakis，对这篇论文的反馈。

勘误

2015-04-23

定稿后，我们发现了若干无意的疏漏和歧义。（译注：译文已将勘误内容放置到对应章节。补充的 2 条参考文献与已有序号冲突，故放在了列表之后，并继续编号为 85，86）

参考文献

[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. **Flume−Java: 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. **CPI2: 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.

[85] 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.

[86] 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.