Google的大规模集群管理系统Borg

摘要：Google的Borg系统是一个运行着成千上万项作业的集群管理器，它同时管理着很多个应用集群，每个集群都有成千上万台机器，这些集群之上运行着Google的很多不同的应用。Borg通过准入控制，高效的任务打包，超额的资源分配和进程级隔离的机器共享，来实现超高的资源利用率。它通过最小化故障恢复时间的运行时特性和减少相关运行时故障的调度策略来支持高可用的应用程序Borg通过提供一个作业声明的标准语言，命名服务的集成机制，实时的作业监控，以及一套分析和模拟系统行为的工具来简化用户的使用。

我们将通过此论文对Borg系统的架构和主要特性进行总结，包括重要的设计决定，一些调度管理策略的定量分析，以及对十年的使用经验中汲取的教训的定性分析。

1.简介

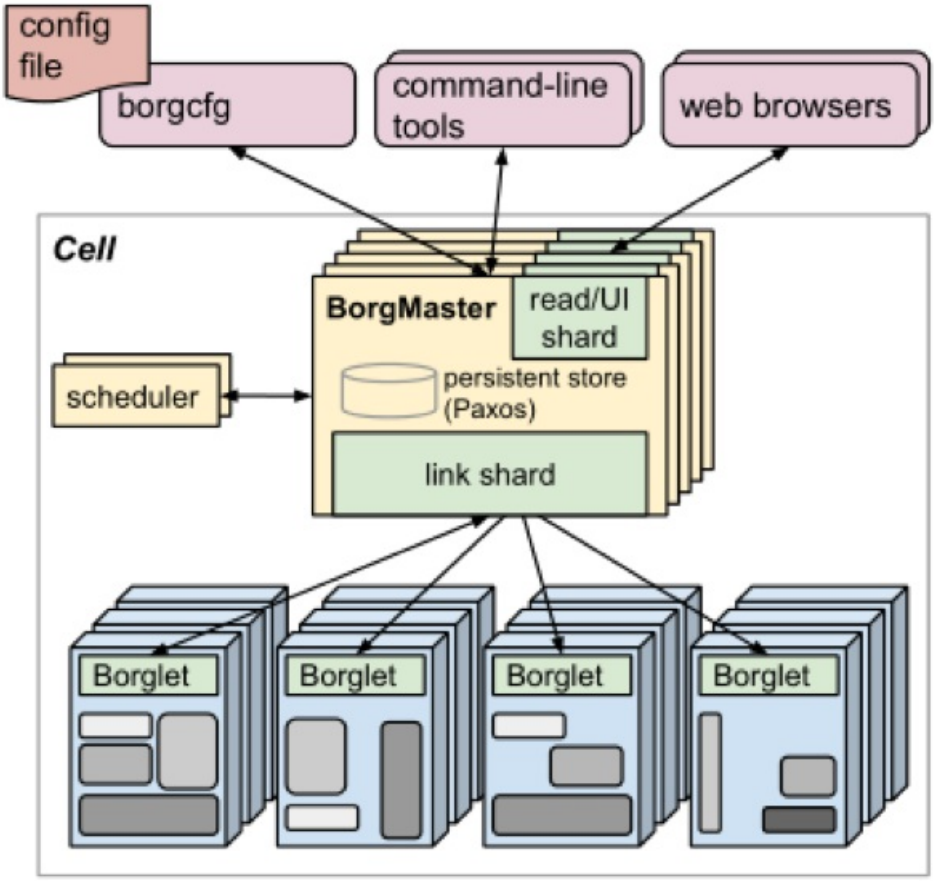


图1 Borg的高级架构。仅显示了成千上万工作节点中的一小部分。

这个在我们内部称为Borg的集群管理系统，它负责权限控制、调度、启动、重新启动和监视全部的Google中运行的应用程序。本文将解释它是如何做到的。

总的来说，Brog主要提供了三个主要的好处：（1）隐藏资源管理和故障处理的细节，因此其用户可以专注于应用程序开发;（2）提供高可靠性和高可用性操作，支持的应用也是如此;（3）使我们能够有效地在数万台机器上运行工作负载。 Borg不是解决这些问题的第一个系统，但它是在能够保证最大弹性和完整性情况下，以大规模运行的少数几个系统之一。 本文将主要围绕这些主题进行组织，并从Borg投入生产，这十多年来的使用经验作为总结 。

2.用户视图

Borg的用户是运行Google应用和服务的Google开发人员和系统管理员（网站可靠性工程师或SRE）。 用户以作业的形式将他们的工作提交给Borg，每个作业包括一个或多个任务，它们都运行相同的程序（二进制）。 每个作业在一个Borg单元中运行，一组机器组织为一个单元。 本节的剩余部分描述了Borg用户视图中展现的主要功能。

2.1 工作负载

Borg的所有单元都同时运行着两种类型的异质工作负载。第一个是“永远运行下去”的长服务，他们对延迟和性能波动敏感， 此类服务用于面向终端用户的产品，例如Gmail，Google文档，web搜索和内部基础设施服务（例如，BigTable）。 第二个是批处理作业，需要花费从几秒到几天完成，这些任务对短期性能波动的敏感性要小得多。这些工作负载混合运行在Borg的各个运行单元中，其根据其主要租户（例如，一些单元是专门用来运行批量密集任务的）运行不同的混合应用，并且也随时间变化：批处理作业完成和重新运行，许多面向终端用户的服务作业看到日常使用模式。 Borg同样需要处理好所有这些情况。

Borg的代表性工作负载情况可以从2011年5月的一个公开的月份跟踪中找到[80]，已经进行了广泛分析（例如[68]和[1,26,27,57]）。

在过去几年中，许多应用程序框架已经建立在Borg之上，包括我们内部的MapReduce系统[23]，FlumeJava [18]，Millwheel [3]和Pregel [59]。 大多数都有一个控制器提交一个主作业和一个或多个工作作业; 前两者对YARN的应用程序管理器[76]起类似的作用。 我们的分布式存储系统如GFS [34]及其后继CFS，Bigtable [19]和Megastore [8]都运行在Borg上。

对于本文，我们将优先级较高的Borg作业分为“生产”（prod）作业，其余作为“非生产”（non-prod）作业。 大多数长期运行的服务器作业是prod;大多数批处理作业是非prod的。在代表性单元中，分配给prod作业大约总CPU资源的70％，大约占总CPU使用量的60％; 分配给它们约总内存的55％，约占总内存使用的85％。在§5.5节，将看到分配和使用之间的差异将是很重要的。

2.2 集群和单元

单元中的机器属于单个集群，由连接它们的高性能数据中心规模的网络架构定义。

一个集群位于单个数据中心大楼内，大厦集合构成一个站点。一个集群通常承载一个大型单元，可能有一些较小规模的测试或特殊用途单元。 我们努力避免任何单点故障。

中央单元大小是排除测试单元后约10k机器; 有些会更大。一个单元中的机器在许多维度上是异构的：大小（CPU，RAM，磁盘，网络），处理器类型，性能和功能（比如外部IP地址或闪存存储器）。Borg通过确定单元中的运行任务，为任务分配资源，安装程序和其他的依赖，监控任务状态并在失败时重启，将用户从大多数差异中隔离出来。

2.3 作业和任务

Borg作业的属性包括名称，所有者及其拥有的任务数量。作业可能具有限制，使其任务在具有特定属性（例如处理器体系结构，操作系统版本或外部IP地址）的计算机上运行。限制可以是硬的或软的; 软限制就像是偏好而不是要求。作业的开始能被推迟到直到前一个作业完成。 一个作业仅在一个单元中运行。

每个任务映射到在机器上的容器中运行的一组Linux进程[62]。 大多数Borg工作负载不在虚拟机（VM）内运行，因为我们不想支付虚拟化的成本。此外，该系统是在我们对没有硬件的虚拟化支持的处理器进行大量投资的时候设计的。

任务也具有属性，例如资源需求和任务在作业中的索引。 大多数任务属性对作业中的所有任务是相同的，但是可以被重写 - 例如，以提供指定任务的命令行标志。每个资源维度（CPU核，RAM，磁盘空间，磁盘访问速率，TCP端口，等）以细粒度独立指定; 我们不强加固定大小的桶或槽（§5.4）。静态链接Borg程序以减少对其运行时环境的依赖，并且Brog程序被打包为二进制文件和数据文件，由Borg负责安装。

用户通过向Borg发出远程过程调用（RPC）来操作作业，最常见的是通过命令行工具，其他Borg作业或监视系统（§2.6）。大多数作业描述都是用声明性配置语言BCL编写的。BCL是GCL的一个变体[12]，它生成protobuf文件[67]，并扩展了一些Borg特定的关键字。GCL提供lambda函数以允许计算，应用程序可以使用它们来调整环境配置; 成千上万的BCL文件超过1k行长，我们已经积累了数千万行的BCL。Borg作业配置与Aurora配置文件相似[6]。

图2说明了作业和任务在其生命周期中经历的状态。

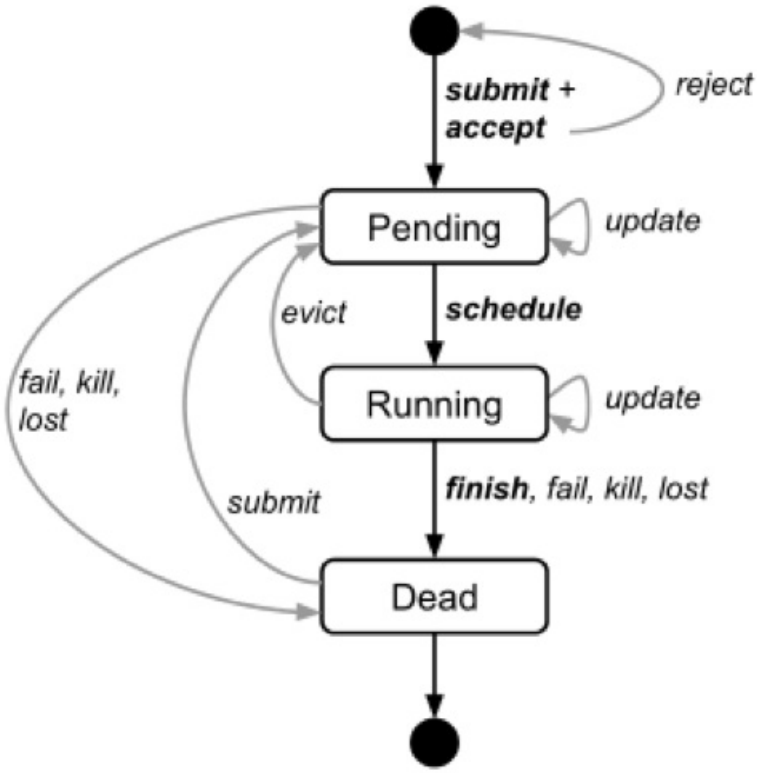


图2：作业和任务的状态图。 用户可以触发提交，终止和更新转换。

用户可以通过推送新的作业配置到Borg，再指示Borg将任务更新到新配置，来更改正在运行的作业中的某些任务或所有任务的属性。 这是一个轻量级的非原子事务，可以很容易地被撤销，直到它被关闭（提交）。 更新通常以滚动方式完成，并且可以对更新导致的任务中断（重新计划或抢占）的数量加以限制; 跳过会导致更多中断的任何更改。

某些任务更新（例如，推送新的二进制）总是需要重启任务; 某些更新（例如，增加的资源需求增加或约束改变）可能使得任务不再适合于这台机器，将导致任务停止并重新调度; 而某些更新（例如，改变优先级）却可在不重新启动或移动任务的情况下进行。

任务可以要求在被SIGKILL抢占之前通过Unix SIGTERM信号获取通知，这样任务就有时间进行清理，保存状态，完成当前正在执行的请求并拒绝新的请求。 如果抢占者设置延迟界限，则实际通知可能更少。 在实践中，通知传递约80％的时间。

2.4 分配

Borg alloc（分配的简称）是可以运行一个或多个任务的机器上的一组保留资源;无论资源是否被使用仍然被分配。 Alloc可以用于为将来的任务设置资源，在停止和重启任务之间保留资源，以及将不同作业中的任务收集到同一台机器上 - 例如，Web服务实例和相关的日志保存任务， 这个任务将服务的URL日志从本地磁盘复制到分布式文件系统。 alloc的资源以类似于机器资源的方式处理; 多个任务运行在一个alloc中，共享其资源。如果一个alloc必须重定位到另一台机器，它的任务将被重新调度。

一个alloc集合就像一个作业：它是一组在多个机器上预留资源的alloc。 一旦创建了一个alloc集，可以提交一个或多个作业在其中运行。 简单期间，我们一般会使用“task”来引用alloc或顶层任务（在alloc之外的）和“job”来引用一个作业或alloc集。

2.5 优先级，配额和接纳控制

当更多的工作出现而超过可容纳的限度时会发生什么？我们的解决方案是优先级和配额。

每个作业都有一个优先级，它是一个小的正整数。高优先级任务可以以牺牲低优先级任务为代价而获得资源，即使这导致抢占（杀死）后者。Borg将不同种类的作业分为不同的领域，并给每个领域定义了不重叠的优先权重，这些作业组包括：监视作业，生产作业，批处理作业和尽力而为的作业（已知的包括测试程序），他们的优先级依次递减。对于本文，prod作业是监测和生产领域的工作。

虽然被抢占的任务通常将被重新安排在单元中的其他地方，抢占级联可能发生，如果高优先级的任务碰到一个略低优先级的任务，而这个略低优先级任务又引发另一个略低优先级的任务，等等。为了消除大部分这种情况，我们不允许生产领域中的任务相互抢占。 细粒度优先级在其他情况下仍然有用 - 例如，MapReduce主任务以比他们控制的workers更高的优先级运行，来提高其可靠性。

优先级表示单元中正在运行或正等待运行的作业的相对重要性。 配额用于决定允许进行调度的作业。 配额表示为在给定优先级上的一段时间（通常为几个月）内的资源量（CPU，RAM，磁盘等）的向量。 数量指定用户的作业请求可以一次请求的资源的最大量（例如，“从现在直到7月底在单元xx中的prod优先级的20TiBRAM“）。 配额检查是许可控制

的一部分，而不是调度：配额不足的作业立即拒绝提交。

较高优先级配额的成本高于较低优先级配额。生产优先级配额仅限于单元中可用的实际资源，因此，提交符合配额的生产优先级作业的用户可以预期运行。 即使我们鼓励用户购买的配额不超过他们的需求，但是许多用户仍然过度购买，因为这帮助他们在应用程序的用户群增长时克服不足。我们通过在较低优先级别上过度销售配额来响应这一点：每个用户具有在优先级零的无限配额，尽管这常常难以执行，因为资源被过度订阅。一个低优先级作业可能被允许了，但是由于资源不足而保持等待（未调度）。

在Borg以外进行配额分配，并且与我们的物理容量规划密切相关，其结果反映在不同数据中心的配额的价格和可用性上。 仅当用户作业具有所需优先级的足够配额时，才允许用户作业。 配额的使用减少了对优势资源公平（DRF）[29,35,36,66]等策略的需要。

Borg有一个能力系统，能给予一些用户特殊的权限; 例如，允许管理员删除或修改单元中的任何作业，或允许用户访问受限内核功能或Borg行为（例如禁用其作业的资源估计（§5.5））。

2.6 命名和监控

仅创建和放置任务是不够的：服务的客户端和其他系统需要能够找到它们，即使它们被重定位到新机器上了。要启用此功能，Borg将为每个任务创建一个稳定的“Borg name service”（BNS）名称，其中包含单元名称，作业名称和任务编号。Borg将任务的主机名和端口写入一个以BNS命名的一致的高可用的Chubby [14]文件中，由我们的RPC系统使用该文件来查找任务端点。BNS名称还形成任务的DNS名称的基础，所以在cc单元中的用户ubar拥有的作业 jfoo中的第五十个任务将通过50.jfoo.ubar.cc.borg.google.com访问到。Borg还会在Chubby发生变化时将作业大小和任务健康信息写入Chubby，因此负载平衡器可以查看将请求路由到哪里。

几乎在Borg下运行的每个任务都包含一个内置的HTTP服务器，它发布有关任务运行状况的信息和成千上万个性能指标（例如RPC延迟）。 Borg监控health-check URL，并重新启动不会及时响应或返回HTTP错误代码的任务。 其他数据由仪表盘监视工具和违反服务级别目标（SLO）的警报进行跟踪。

称为Sigma的服务提供了基于Web的用户界面（UI），通过该UI用户可以检查所有作业，特定单元的状态，或向下钻取到单个作业和任务，以检查其资源行为，详细日志，执行历史 ，和最终的结果。 我们的应用产生大量日志; 这些被自动轮转以避免用完磁盘空间，并在任务退出后保存一段时间以协助调试。 如果作业未运行，Borg提供了“为什么待处理？”注释，以及如何修改作业的资源请求以更好地适应单元的指导。 我们发布了“切合”更可能容易调度的资源形式的规则。

Borg记录所有作业提交事件和任务事件，以及每个任务在Infrastore中详细的资源使用信息，这是一个可扩展的只读数据存储，通过Dremel [61]具有一个交互式的类似SQL的界面。此数据用于基于使用的计费，作业调试和系统故障以及长期容量规划。 它还为Google群集工作负载跟踪提供数据[80]。

所有这些功能都有助于用户理解和调试Borg的行为及用户的作业，并帮助我们的SREs为每个人管理几万台机器。

3.Borg体系结构

Borg单元由一组机器，一个称为Borgmaster的逻辑中央控制器和单元中每台机器上运行的称为Borglet的代理进程构成（参见图1）。 Borg的所有组件都用C ++编写。

3.1 Borgmaster

每个单元的Borgmaster包括两个进程：主进程Borgmaster和独立的调度程序（§3.2）。 主Borgmaster进程处理客户端RPC，状态变化（例如，创建作业）或提供对数据的只读访问（例如，查找作业）。它还管理系统中所有对象（机器，任务，分配等）的状态机，与Borglets进行通信，并提供Web UI作为Sigma的备份。

Borgmaster在逻辑上是一个单一的进程，但实际上被复制了五次。 每个副本维护了一份该单元大部分状态的内存副本，并且该状态也记录在该副本的本地磁盘上的高可用性，分布式，基于Paxos的存储[55]中。每个单元的单个选定的master既用作Paxos的领导者又用作状态mutator，处理改变单元状态的所有操作（例如提交作业或在机器上终止任务）。当cell建立时或只要当选择的master出现故障时，就会选择一个master（使用Paxos）; 它获取一个Chubby锁，以便其他系统可以找到它。选择一个master和故障转移到新的master通常需要大约10s，但是在大单元中可能需要一分钟，因为一些内存中的状态必须重建。 当副本从中断恢复时，它将自动重新同步来自最新的其它Paxos副本的状态。

Borgmaster在某个时间点的状态称为检查点，并采用定期快照的形式增加一条更改日志（保存在Paxos存储中）。检查点有许多用途，包括将Borgmaster的状态恢复到过去的任意一个点（例如，在接受触发Borg中的软件缺陷的请求之前，以便可以对其进行调试）; 构建用于未来查询的事件的持久日志; 以及离线模拟。

高保真的Borgmaster模拟器Faokemaster可用于读取检查点文件，并包含产生Borgmaster代码的完整副本，其中包含与Borglets的无存根接口。它接受RPC进行状态机更改和执行操作，如“调度所有挂起的任务”，通过与它进行交互（它就像是一个活的Borgmaster，带有模拟的Borglets可从检查点文件重放真实的交互），可以使用它来调试故障。用户可以逐步观察在过去实际发生的系统状态的改变。 Fauxmaster对于容量规划（“符合多少这种类型的新作业？”）以及在更改单元配置之前进行完整性检查（“这种更改是否会驱逐重要的工作？”）也很有用。

3.2 调度

提交作业时，Borgmaster会将其持久化在Paxos存储中，并将作业的任务添加到等待队列。 这是由调度程序异步扫描

的，如果有足够的可用资源满足作业的要求，则会将任务分配给机器。（调度程序主要操作任务，而不是作业。）扫描从高到低优先级，由优先级循环方案调度，以确保用户之间的公平性，并避免大型作业后面的队头阻塞。 调度算法有两个部分：可行性检查（用于找到任务可以运行的机器），以及评分（用于挑选一个可行的机器）。

在可行性检查中，调度器找到满足任务需求的一组机器，这组机器具有足够的“可用”资源 - 这些资源中包括已经分配给可以被抢占的较低优先级任务的资源。 在评分中，调度器确定每个可行机器的“良好性”。该分数考虑了用户指定的偏好，但主要是由内置标准决定，如最大限度地减少抢占任务的数量和优先级，选择已经有任务包副本的机器，跨越电源和故障域传播任务，以及打包质量（包括将高优先级任务和低优先级任务混合到单个机器上，以允许高优先级任务在负载高峰中扩展）。

Borg最初使用E-PVM [4]的变体进行评分，其在不同资源上生成单一成本值，并且在放置任务时最小化成本的变化。在实践中，E-PVM最终在所有机器上扩展负载，为负载高峰留下余量 - 但是以增加碎片为代价，特别是对于需要大部分机器的大型任务; 我们有时称之为“刚好合适”。

调度的另一端是“最佳合适”，它试图尽可能紧密地填充机器。 这使一些机器没有用户作业（它们仍然运行存储服务器），因此放置大任务是简单直接的，但是严格的封装不利于用户或Borg对资源需求的任何错误估计。 这会伤害突发负载的应用程序，对于指定低CPU需求的批处理作业尤其糟糕，以便他们可以轻松安排并尝试在未使用的资源中伺机运行：20％的非生产任务请求少于0.1个CPU内核。

我们当前的评分模型是一种混合式的，它试图减少搁置资源的数量 - 由于机器上的另一个资源被完全分配而无法使用的资源。 它提供比最适合我们工作负载约3-5％的更好的包装效率（在[78]中定义）。

如果计分阶段选择的机器没有足够的可用资源来满足任务，则Borg会抢占（杀死）较低优先级任务，从最低优先级到最高优先级，直到满足为止。 我们将被抢占的任务添加到调度程序的挂起队列，而不是迁移或休眠它们。

任务启动延迟（从作业提交到任务运行的时间）是一个已经并继续受到极大关注的领域。它是高度可变的，中值通常约25s。 软件包安装大约占全部的80％：其中一个已知的瓶颈是软件包要写入的本地磁盘的争用。为了减少任务启动时间，调度程序更倾向将任务分配给已经安装了必要的软件包（程序和数据）的机器：大多数软件包是不可变的，因此可以共享和缓存。（这是Borg调度程序支持数据本地化的唯一形式。）此外，Borg使用类似树和torrent的协议并行地将软件包分发到机器。

此外，调度程序使用几种技术来扩展具有成千上万台机器的单元（§3.4）。

3.3 Borglet

Borglet是一个本地Borg代理，存在于单元中的每一台机器中。它启动和停止任务; 如果故障就重启任务; 通过操纵操作系统内核设置来管理本地资源; 翻转调试日志; 并向Borgmaster等监控系统报告机器的状态。

Borgmaster每隔几秒钟轮询一次Borglet以检索机器的当前状态，并将所有未完成的请求发送给它。 这使Borgmaster控制通信速率，避免了显式流控制机制的需要，并防止恢复风暴[9]。

选定的master负责准备要发送到Borglets的消息，并负责根据cell的响应更新cell的状态。为了性能可扩展性，每个Borgmaster副本运行无状态链接分片来处理与一些Borglets的通信;每当发生Borgmaster选择时重新计算分区。对于弹性，Borglet始终报告其完整状态，但链接分片通过仅报告状态机间的差异来收集和压缩此信息，以减少选定master的更新负载。

如果Borglet没有响应几个轮询消息，它的机器被标记为关闭，并且其运行的任何任务被重新安排在其他机器上。如果通信恢复，Borgmaster会通知Borglet要停止这些已经重新安排的任务，以避免重复。即使与Borgmaster失去联系，Borglet也继续正常运行，因此即使所有Borgmaster副本故障了，当前运行的任务和服务也会保持。

3.4可扩展性

我们不确定Borg的集中式架构的最终可扩展性限制将出现在何处; 到目前为止，每次我们接近一个极限，我们已经设法消除它。一个Borgmaster可以管理一个cell中的数千台机器，并且几个cell具有每分钟超过10000个任务的到达速率。繁忙的Borgmaster使用10-14个 CPU内核和高达50GiB 的RAM。我们使用几种技术来实现这种规模。

早期版本的Borgmaster有一个简单的，同步的循环：接受请求，计划任务，并与Borglets通信。为了处理更大的cell，我们将调度程序分离出来作为一个单独的进程，这样它可以与其他Borgmaster函数并行操作故障容限。 调度器副本对单元状态的高速缓存副本进行操作。它反复：从选定的主机检索状态更改（包括已分配和挂起的工作）; 更新其本地副本;执行调度传递以分配任务; 并将这些分配通知选定的主机。master将接受并采用这些分配，除非它们是不适当的（例如，基于过期状态），这将导致它们在调度程序的下一次传递中被重新考虑。这在灵魂上与在Omega [69]中使用的乐观并发控制非常相似，事实上，我们最近为Borg添加了针对不同工作负载类型使用不同调度程序（schedulers）的能力。

为了提高响应时间，我们添加了单独的线程来与Borglets进行通信并响应只读RPC。为了更好的性能，我们在五个Borgmaster副本（§3.3）中分割（分区）这些功能。同时，这保持了UI上99％ile的响应时间低于1s和95％ile 的Borglet轮询间隔低于10s。

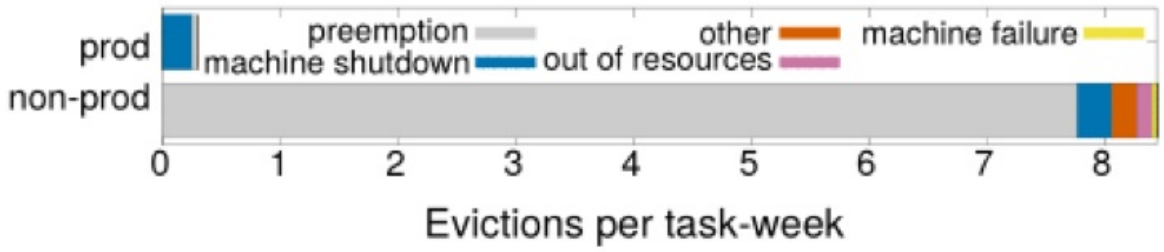


图3：针对生产和非生产工作负载的任务驱逐率及原因。 数据自2013年8月1日起。

有几点使Borg调度器更具可扩展性：

分数缓存：评估可行性和评价机器是昂贵的，因此Borg缓存分数直到机器或任务的属性改变 - 例如，机器上的任务终止，属性改变或任务的需求改变。 忽略资源数量的小变化可减少高速缓存失效。

等价类：Borg作业中的任务通常具有相同的需求和约束，因此并不是确定每个机器上的每个挂起任务的可行性，并对所有可行的机器进行评分，Borg只对每个等价类的一个任务进行可行性分析和评分 - 一组具有相同需求的任务。

轻松随机化：计算大cell中所有机器的可行性和分数是浪费的，因此调度程序以随机顺序检查机器，直到找到“足够”可行的机器进行评分，然后选择该集合中的最佳机器。 这减少了任务进入和离开系统时所需的评分和高速缓存失效的数量，并加快了任务到机器的分配。放松随机化有时类似于Sparrow [65]的批量采样，同时还处理优先级，抢占，异质性和软件包安装的成本。

在我们的实验（§5）中，从头开始安排单元的整个工作负载通常需要几百秒，但是在禁用上述技术后超过3天后还没有完成。 通常，在等待队列上的在线调度传递在不到半秒内完成。

4.可用性

故障是大规模系统中的常态[10,11,22]。图3提供了15个样本cell中任务驱逐原因的分解。运行在Borg上的应用程序应能使用诸如复制，在分布式文件系统中存储持久状态并（如果适当的话）捕捉临时检查点等技术来处理此类事件。即使如此，我们也试图减轻这些事件的影响。例如，Borg：

如有必要，在新机器上自动重新安排逐出的任务;

通过在诸如机器，机架和电源域之类的故障域中扩展作业的任务，减少相关故障;

限制任务中断的允许速率和任务数量，这些任务可以在维护活动（例如操作系统或机器更新）期间同时关闭；

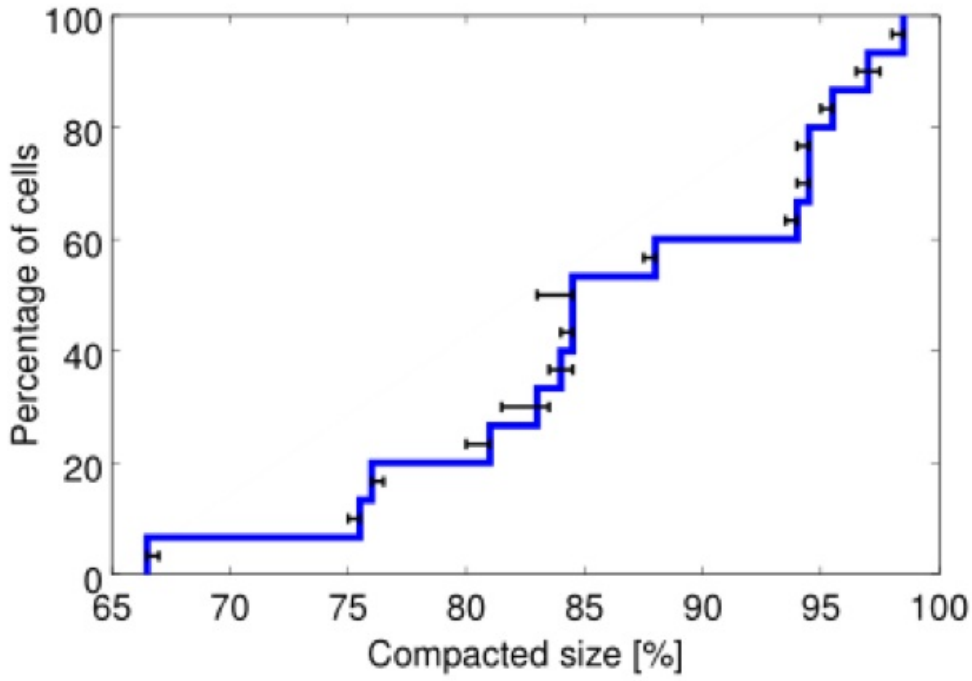


图4：压缩的效果。 对15个cell，在压缩后获得的原始cell大小的百分比的CDF。

使用声明性期望状态表示和幂等变换操作，使得失败的客户端可以无损地重新提交任何被遗忘的请求;

rate-limits找到无法访问的机器的任务的新位置，因为它无法区分大型机器故障和网络分区;

避免重复任务::导致任务或机器崩溃的机器配对;

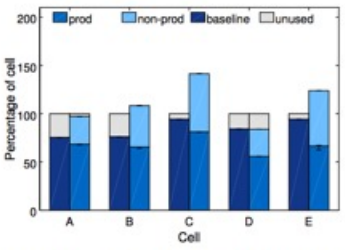
通过不断重新运行日志记录器任务（§2.4）来恢复写入本地磁盘的关键中间数据，即使所连接的alloc已终止或移动到了另一台机器。 用户可以设置系统持续尝试的时间;一般是几天。

Borg的一个关键设计特点是，即使Borgmaster或任务的Borglet关闭，已经运行的任务也会继续运行。但是保持master仍然很重要，因为当它关闭时，无法提交新作业或更新现有的作业，并且无法重新计划故障的计算机上的任务。

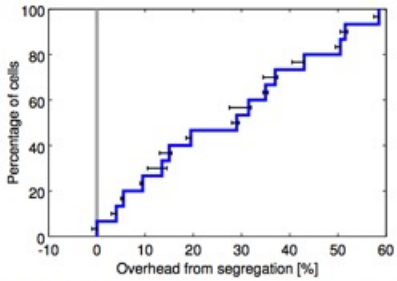
Borgmaster使用的技术组合，使其在实践中达到了99.99％的可用性：机器故障复制; 准入控制避免过载; 并使用简单的低级工具部署实例以最小化外部依赖性。 每个单元独立于其他单元，以最小化关联的操作者错误和故障传播的机会。这些目标，不是可扩展性限制，而是反对较大cell的主要论证。

5.利用

Borg的主要目标之一是高效利用Google的机器，这意味着巨大的财务投资：将利用率提高几个百分点可以节省数百万美元。 本节讨论和评估Borg使用的一些策略和技术。



(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.*

图5：将prod和non-prod工作分离到不同的单元将需要更多的机器。这两个图表显示如果prod和non-prod工作负载发送到单独的单元，需要多少额外的机器，使用百分比表示在一个单元中运行工作负载所需的最小机器数量。在当前和随后的CDF图中，每个单元显示的值推导自我们的实验尝试产生的不同cell大小的90％，误差条显示了尝试值的完整范围。

5.1 评估方法

我们的工作存在布局限制，需要处理稀少的工作负载峰值，机器是异构的，在从服务作业中回收的资源中运行批处理作业。 所以，为了评估策略选择，需要一个比“平均利用率”更复杂的度量标准。 经过多次实验，我们选择了cell compaction：给定一个工作负载，通过删除cell中的机器直到不再适应，可以发现它能适应的cell有多小，，从头开始重新包装工作负载以确保没有挂在一个糟糕的配置。 这提供了干净的最终条件，促进了自动化比较（没有合成工作生成和建模的陷阱 [31]）。评价技术的定量比较可以在[78]中找到：细节是令人惊讶的微妙。

不可能对真实生产单元进行实验，但可以使用Fauxmaster获得高保真的模拟结果，使用来自实际生产单元和工作负载的数据，包括其所有约束、实际限制、保留和使用数据（§5.5 ）。 这些数据来自2014-10-01 14：00 PDT的Borg检查点。 （其他检查点产生了类似结果）。选择15个Borgcell进行报告，首先消除特殊用途、测试和小（<5000台机器）cell，然后对剩余的cell进行取样，以获得大小均匀的范围。

为了在压缩的cell中维持机器异构性，随机选择机器进行删除。 为了保持工作负载的异构性，保留除了与特定机器（例如，Borglets）绑定的服务器和存储任务外的一切。为大于原cell大小一半的作业更改硬约束为软约束，并允许多达0.2％的任务等待，如果他们非常“挑剔”并只能放置在少数几台机器上; 广泛的实验表明，这产生了低方差的可重复结果。 如果我们需要比原cell更大的cell，可以在压缩之前克隆原始cell几次; 如果我们需要更多cell，只需要克隆原来的cell即可。

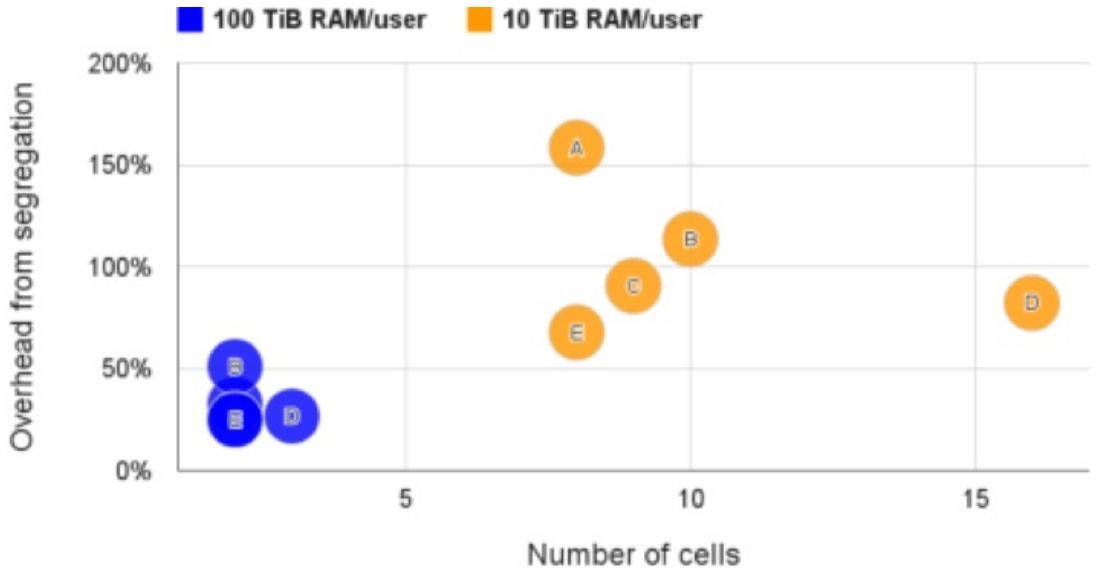
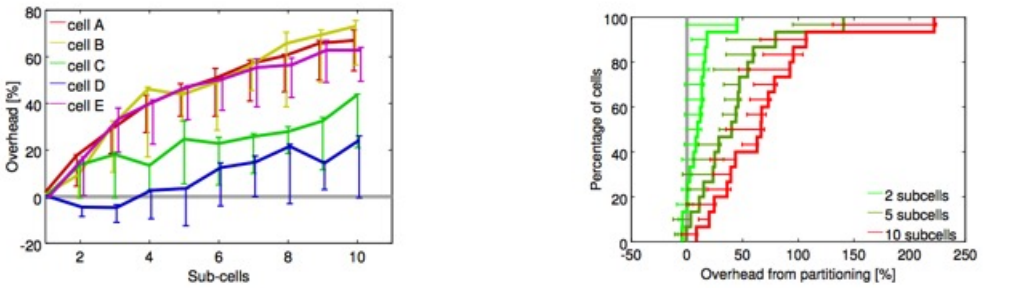


图6：分离用户将需要更多的机器。如果用户多于所示的阈值，针对5个不同的cell，给出了各私有单元的 Cell总数和需要的额外机器。

对于具有不同随机数种子的每个cell，每个实验重复11次。 在图表中，我们使用错误栏显示所需机器数量的最小值和最大值，并选择90%ile值作为“结果”-平均值或中位数并不反映系统管理员如果希望合理地确保工作负载适合时将做什么。 我们认为cell压缩提供了一种公平、一致的方法来比较调度策略，它直接转化为成本/效益结果：更好的策略需要更少的机器来运行相同的工作负载。

我们的实验集中在从一个时间点调度（打包）工作负载，而不是重放长期工作负载跟踪。 这部分是为了避免应对开放和封闭排队模型的困难[71,79]，部分原因是传统的完成时间指标不适用于长期运行服务的环境，部分是为了提供清晰的信号 进行比较，部分是因为我们不相信结果会有明显的不同，部分是一个实际问题：我们发现自己消耗了200000个Borg CPU核用于实验 - 即使按谷歌的规模，这也是一个非平凡的投资 。

在产品中，针对工作负载的增长留出有效的空间，偶然的“black swan”事件，负载高峰，机器故障，硬件更新，以及大规模局部故障（e.g., a power supply bus duct）。图4显示了如果应用了cell压缩，真实的cell有多小。图中跟随的基线使用了压缩的大小。



(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.*

图7：将cell细分成更小的cell将需要更多的机器。 如果将这些特定cell划分为不同数量的较小cell，则需要额外的机器（作为单个cell情况的百分比）。

5.2 cell共享

几乎所有机器同时运行着prod 和 non-prod任务：在共享的Borg cell中98%的机器，有83%跨越Borg管理的整个的机器集合。（有一些特殊用途的专用cell）。

由于很多其它组织在单独的集群中运行面向用户或批量作业，我们对如果我们也这样做会发生什么做了调查。图5显示了分离开proc和non-proc工作，在中值的cell中将需要20-30%的更多的机器来运行工作负载。这是因为proc作业总是会预留资源来处理稀有的负载高峰，但大部分时间并不使用这些资源。Borg回收利用这些未使用的资源来运行大部分non-proc工作，这样总体上就需要更少的机器了。

大部分Borg cell被数以千计的用户共享。图6显示了原因。对这个测试，如果用户的工作负载消耗了至少10TiB（或者100 TiB）的内存，就分离用户的工作负载到一个新的cell。现存的策略看起来是好的：即使是更大的极限，将需要2–16×作为一些cell，以及20–150%的额外的机器。再次，合并资源显著减少了成本。

但是可能将不相关的用户和作业类型打包到同一台机器上会导致CPU干扰，因此需要更多的机器来弥补？ 为了评估这一点，我们研究了在具有相同时钟速度的相同机器类型上运行的不同环境中的任务的CPI（每个指令的周期数）如何改变。 在这些条件下，CPI值是可比的，并且可以用作针对性能干扰的代理，因为CPI的翻倍加倍了CPU绑定程序的运行时间。在一个星期内从约12000个随机选择的prod任务收集数据，使用[83]中描述的硬件配置架构，在5分钟的间隔内

对周期和指令进行计数，并对样本进行加权，使得每秒钟的CPU时间被公平计数。 结果并不清晰。

（1）我们发现CPI与在相同时间间隔内的两个测量结果正相关：机器上的总体CPU使用率，（很大程度上独立地）机器上的任务数量; 向机器添加任务使得其他任务的CPI提高0.3%（使用适合这些数据的线性模型）; 将机器CPU使用率提高10%使得CPI提高小于2%。 但是即使相关性在统计上是重要的，也只显示了在CPI测量中看到的方差的5%; 其他因素占主导地位，例如应用的固有差异和特定的干扰模式[24,83]。

（2）将我们从共享cell中采样的CPI与来自具有较少不同应用的几个专用cell的CPI进行比较，我们看到共享cell的平均CPI为1.58（σ= 0.35），专用cell的平均CPI为1.53（σ= 0.32）- 即，CPU在共享cell中性能降低约3%。

（3）为了解决不同cell中的应用可能具有不同工作负载或还遇到选择偏差（可能将干扰更敏感的程序已经移动到专用cell）的担忧，考察了Borglet的CPI，它在两种类型的所有机器上运行。它在专用cell中的CPI为1.20（σ= 0.29），在共享cell中的CPI为1.43（σ= 0.45），表明它在专用cell中的运行速度为1.19倍，与在共享单元中一样快，虽然这加重了轻负载机器的影响，但轻微偏差结果有利于专用cell。

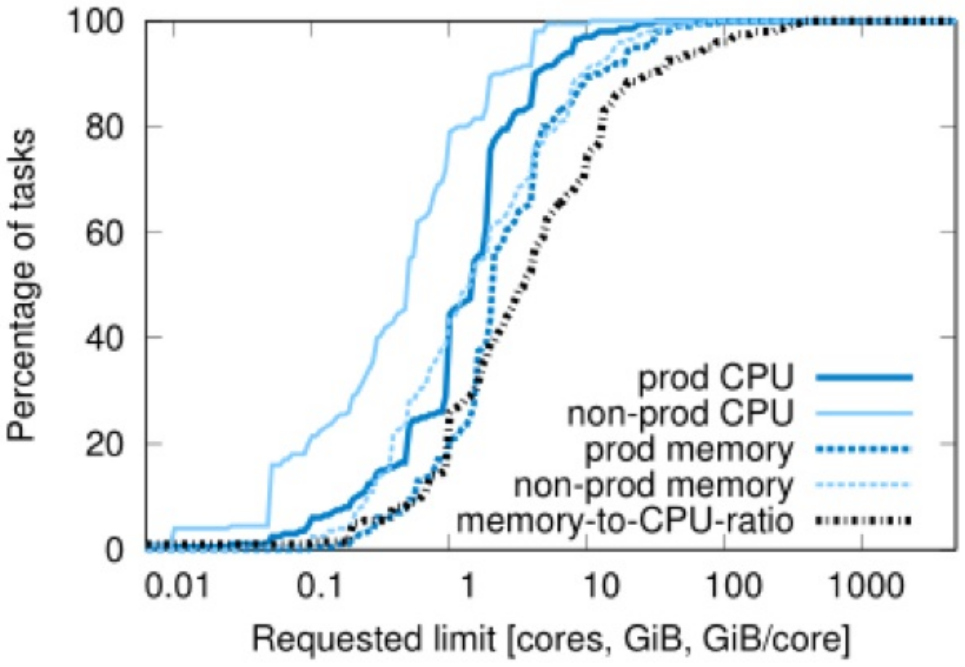


图8：没有适合大多数任务的桶大小。请求的CPU和内存的CDF要求跨越样本单元。

没有一个值突出，虽然几个整数CPU核大小有点更受欢迎。

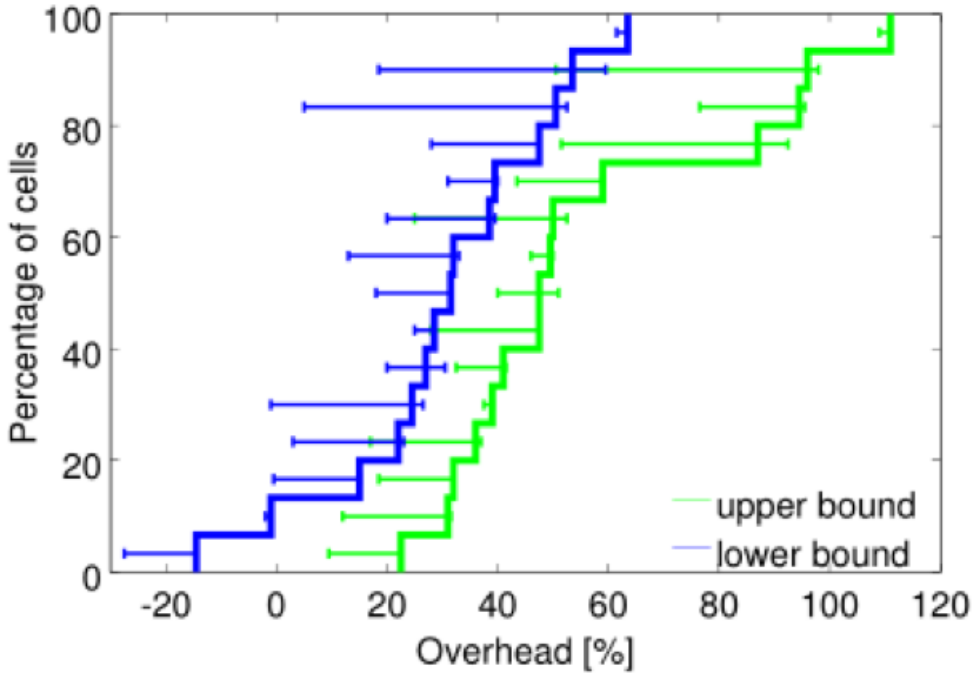


图9：“Bucketing”资源需求将需要更多的机器。

将15个单元中的CPU和存储器请求四舍五入为下一个最接近2的幂所产生的额外开销的CDF。 下限和上限跨越实际值

（见文本）。

这些实验证实，仓库规模的性能比较是棘手的，在[51]中加强了观察，并且还表明共享不会大大增加运行程序的成本。

但即使假设最不利的结果，共享仍然是一个胜利：CPU减速超过了几种不同划分方案所需机器的减少，共享优点适用于所有资源，包括内存和磁盘，而不仅仅是CPU 。

5.3 大细胞

Google构建了大cell，以允许运行大型计算，并减少资源碎片。通过将cell的工作负载分到多个较小的cell来测试后者的效果 - 首先随机排列作业，然后在分区之间以循环方式分配作业。 图7证实使用较小cell将明显需要更多的机器。

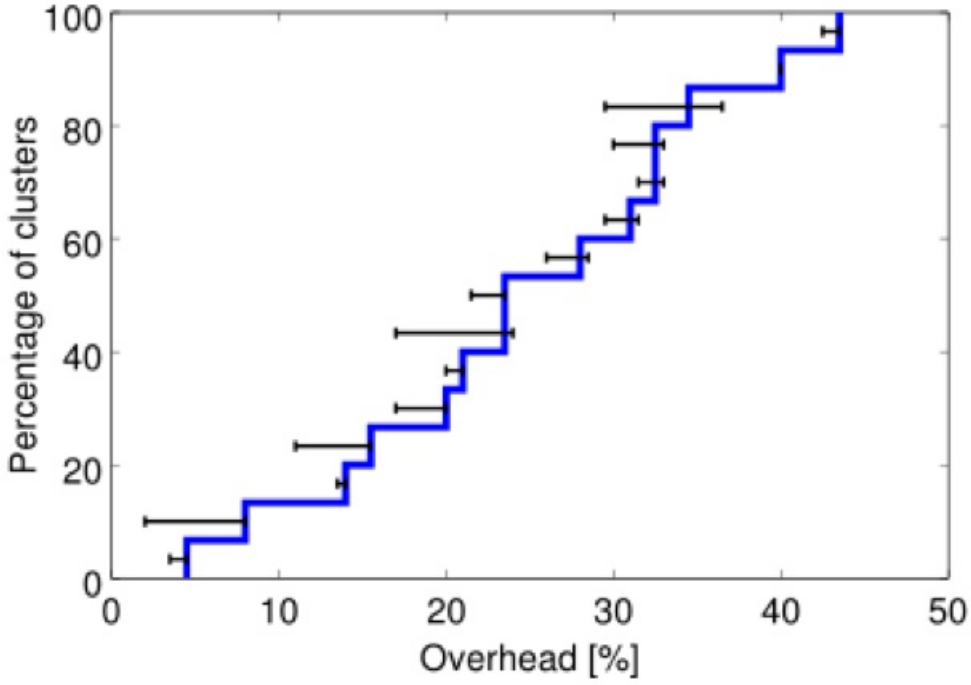


图10：资源回收是相当有效的。 如果禁用15个代表性cell，则需要额外机器的CDF。

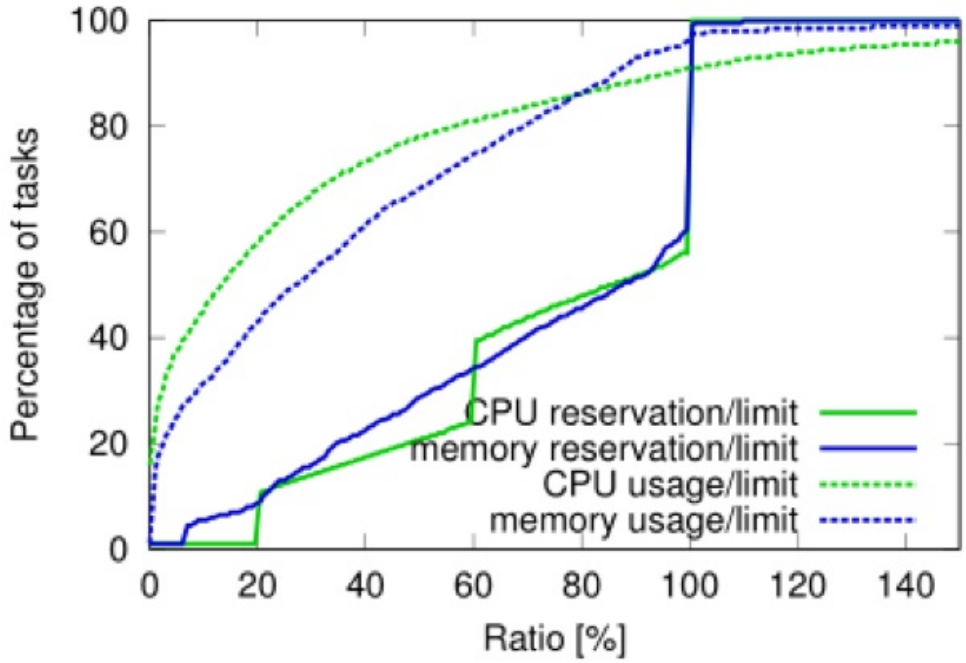


图11：资源估计在识别未使用的资源方面是成功的。虚线显示了15个cell中的任务请求（极限）的CPU和内存使用率的CDF。大多数任务使用远远低于其极限，虽然少数比请求的使用了更多的CPU。实线显示出了CPU和内存保留限制率的CDF; 这些都更接近100%。直线是资源估计处理的伪像。

5.4 细粒度资源请求

Borg用户请求CPU以milli-cores为单位，内存和磁盘空间以字节为单位。 （core是处理器超线程，针对机器类型的性能进行标准化）。图8显示了利用这种粒度：在所请求的内存或CPU核的数量上几乎没有明显的“sweet spots”，并且这些资源之间几乎没有明显的相关性。除了在90%及以上的内存请求稍大之外，这些分布与[68]中提出的分布非常相

似。

提供一组固定大小的容器或虚拟机虽然在IaaS（基础设施即服务）提供商[7,33]中很常见，但不能很好地满足我们的需求。为了说明这一点，通过在每个资源维度上将它们四舍五入到下一个最接近的2的幂，从CPU的0.5内核和RAM的1GiB开始，对prod作业和分配（§2.4）“bucketed”CPU核和内存资源限制。图9显示这样做将需要比平均情况下多30-50%的资源。上限来自于将整个机器分配给大型任务（在压缩前将原始细胞翻两番后不适合）；下限来自于允许这些任务进入待定状态。（这小于[37]中报告的大约100%的开销，因为我们支持超过4个buckets，并允许CPU和RAM容量独立扩展。）

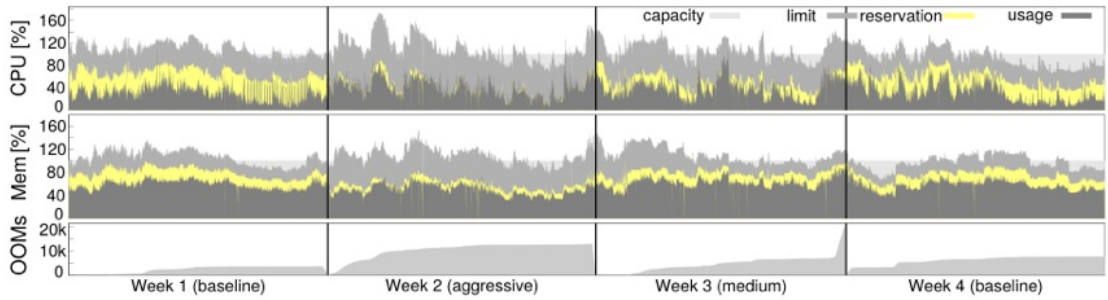


图12：更积极的资源估计可以回收更多的资源，对内存不足事件（OOM）几乎没有影响。一个生产cell使用的时间线（从2013-11-11开始），5分钟窗口平均的预留和限制以及累积的内存不足事件; 后者的斜率是OOM的总速率。竖栏使用不同的资源估计设置按周分开。

5.5 资源回收

作业可以指定资源限制 - 每个任务应被授予的资源的上限。 Borg使用该限制来确定用户是否有足够的配额来接受作业，并确定特定机器是否有足够的免费资源来安排任务。 正如有用户购买比所需要的更多的配额，有用户请求比任务将使用的更多的资源，因为Borg通常会杀死一个试图使用比它需要的更多的RAM或磁盘空间的任务，或节流CPU到 任务所要求的。 此外，某些任务偶尔需要使用其所有资源（例如，在一天的高峰时间或在应对拒绝服务攻击时），但大多数时间不会。

不是浪费当前未被消耗的已分配资源，而是估计任务将使用多少资源，并回收可以容忍低质量资源（例如批处理作业）的工作的剩余资源。这整个过程称为资源回收。 该估计称为任务的预留，并且由Borgmaster每几秒钟使用由Borglet捕获的细粒度使用（资源消耗）信息来计算。初始预留被设置为等于资源请求（限制）;在300s后，为了允许瞬间启动，缓慢向实际使用加上安全余量衰减。如果使用超过，预订会迅速增加。

Borg调度程序使用极限来计算prod任务的可行性（§3.2），因此调度程序从不依赖于已回收的资源，也没有暴露给资源超额订阅; 对于non-proc任务，使用现有任务的预留，以便将新任务安排到已回收的资源中。

机器在运行时可能会耗尽资源，如果这个保留（预测）是错误的 - 即使所有任务使用的资源小于其极限。 如果发生这种情况，杀死或限制non-proc任务（永不处理proc任务）。

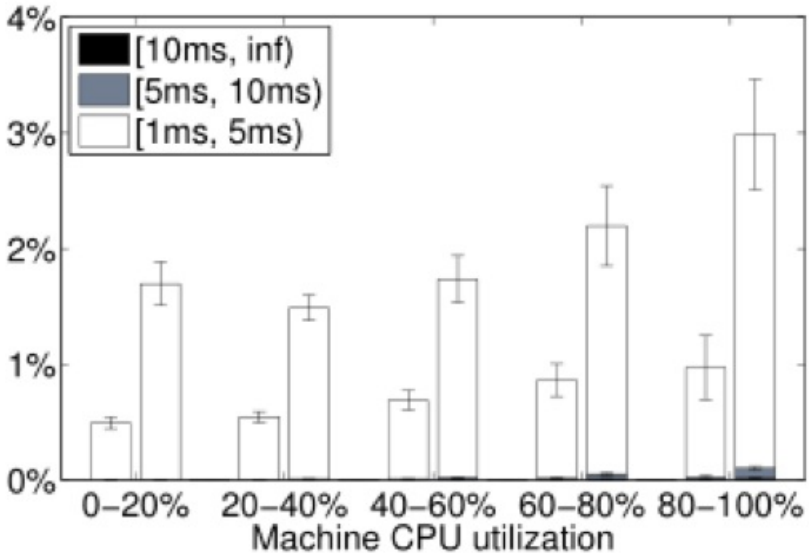


图13：调度延迟作为负载的函数。 描绘了一个可运行的线程必须等待多于1ms的时间来访问CPU的频率，并作为机器有多忙的函数。 在每对竖栏中，延迟敏感的任务在左侧，批处理任务在右侧。仅在几个时间的百分比中，线程必须等待超过5ms才能访问CPU（白色栏）;其它几乎不必等待更长的时间（更黑的栏）。来自从 2013年12月以来代表性单元格的数据；错误栏显示每日差异。

图10显示了更多的机器无需资源回收。大约20%的工作负载（§6.2）在中值cell的回收资源中运行。

可以在图11中看到更多细节，其中显示了预留和使用限制的比率。 如果需要资源，超过其内存限制的任务将首先被抢

占，而不管其优先级如何，因此任务超过其内存限制是很少见的。 另一方面，CPU可以容易地被节流，因此短期峰值可以相当无害地推动使用高于预留（push usage above reservation fairly harmlessly）。

图11表明资源回收可能是无必要保守的：在预留和使用线之间存在明显的区域。 为了测试这一点，选择了一个活的生产cell，并通过减少安全边界，在第一周调整资源估计算法的参数到一个积极的设置，然后在下一周调整到基线和积极设置之间的中间设置，然后恢复到基线。图12显示发生了什么。在第二周 保留明显更接近使用，第三周稍差，基准周（第一和第四周）显示了最大差距。如预期的那样，内存（OOM）事件的发生率在第2周和第3周轻微增加。在评估结果后，我们决定净收益超过了消极面，并将中等资源回收参数部署到其他 cell。

6.隔离

50％的机器运行9个或更多的任务; 一个90%ile的机器大约有25个任务，将运行大约4500个线程[83]。虽然在应用之间共享机器增加了利用率，但还需要良好的机制来防止任务彼此干扰。这都适用于安全性和性能。

6.1 安全隔离

使用Linux chroot jail作为同一台机器上多个任务之间的主要安全隔离机制。为了允许远程调试，通过使用自动分发（和撤销）ssh密钥，使得用户只有当机器在为该用户运行任务时才能访问该机器。 对于大多数用户，这已被替换为borgssh命令，它与Borglet协作构建一个ssh连接到一个shell，该shell在与任务相同的chroot和cgroup中运行，从而更紧密地锁定访问。

VM和安全沙盒技术用于通过Google的AppEngine（GAE）[38]和Google Compute Engine（GCE）运行外部软件。在作为Borg任务运行的KVM进程[54]中运行每个托管的VM。

6.2 性能隔离

Borglet的早期版本具有相对原始的资源隔离实施：内存，磁盘空间和CPU周期的事后使用检查，结合使用过多内存或磁盘的任务，以及积极应用Linux的CPU优先级来控制使用太多CPU的任务。 但是欺骗任务对机器上其他任务的性能影响仍然太容易，因此一些用户膨胀了他们的资源请求，以减少Borg可以与他们共同调度的任务数量，这降低了利用率。 资源回收可以收回一些剩余的，但不是所有，因为涉及安全边际。在最极端的情况下，用户请求使用专用机器或cell。

现在，所有Borg任务都在基于Linux cgroup的资源容器中运行[17,58,62]，Borglet操作容器设置，提供更好的控制(因为操作系统内核在循环中)。 即使如此，偶尔的低级资源干扰（例如，存储器带宽或L3高速缓存污染）仍然发生，如在[60,83]中。

为了帮助过载和过量使用，Borg任务有一个应用类或appclass。 最重要的区别存在于延迟敏感（LS）应用类和其余的应用类（在本文中称为批处理）中。LS任务用于面向用户的应用程序和需要快速响应请求的共享基础结构服务。高优先级LS任务得到最佳处理，并且能够一次暂时使批量任务挨饿几秒钟。

二次分割存在于可压缩资源（例如，CPU周期，磁盘I / O带宽）和不可压缩资源（例如，存储器，磁盘空间）中，可压缩资源是基于速率的并且可以通过在不杀死它的情况下降低其服务质量而从任务中回收；不可压缩资源通常不能在不杀死任务的情况下被回收。 如果机器耗尽了不可压缩资源，则Borglet立即终止任务，从最低优先级到最高优先级，直到可以满足剩余保留为止。 如果机器耗尽了可压缩资源，Borglet会限制使用（有利于LS任务），使得可以处理短负载高峰而不杀死任何任务。 如果情况没有改善，Borgmaster将从机器中删除一个或多个任务。

Borglet中的用户空间控制循环基于预测的未来使用情况（针对prod任务）或内存压力（针对非prod的情况）给容器分配内存；处理来自内核的Out-of-Memory（OOM）事件; 并且当尝试分配超出其内存限制时，或者当过度提交的机器实际上耗尽内存时，杀死任务。 Linux的渴望文件缓存由于需要准确的内存核算，显着地使实现复杂化。

为了提高性能隔离，LS任务可以保留整个物理CPU核，从而阻止其他LS任务使用。 批处理任务允许在任何核上运行，但被赋予相对于LS任务的小调度程序共享。Borglet动态调整贪婪LS任务的资源上限，以确保不会使批处理任务挨饿多分钟，在需要时选择性地应用CFS带宽控制[75]； 共享是不足的，因为我们有多个优先级。

像Leverich [56]，我们发现标准的Linux CPU调度器（CFS）需要大量调整，以支持低延迟和高利用率。 为了减少调度， CFS版本使用扩展的每个群组的负载历史[16]，允许由LS任务抢占批任务，并且当多个LS任务在一个CPU上是可运行时减少调度量。 幸运的是，许多应用程序使用线程请求模型，这减轻了持续负载不平衡的影响。谨慎使用cpusets将CPU核分配给具有特别紧迫延迟要求的应用程序。 这些努力的一些结果如图13所示。这一领域的工作继续，添加线程布局和CPU管理，即NUMA-，超线程和功率感知（例如，[81]），并提高Borglet的控制保真度。

允许任务使用达到其限制的资源。大多数被允许超过用于诸如CPU的可压缩资源，以利用未使用（松弛）资源。 只有5％的LS任务禁用此功能，可能获得更好的可预测性; 少于1％的批量任务使用此功能。默认情况下，禁止使用闲置内存，因为这增加了任务被终止的机会，但即使如此，10％的LS任务会覆盖此功能，而且79％的批处理任务这样做，因为这是MapReduce框架的默认设置。这补充了回收资源的结果（§5.5）。批处理任务会随机利用未使用的以及回收的内存：大多数时间这是可行的，虽然偶尔当一个LS任务急需资源时，批处理任务会被牺牲。

7.相关工作

已经研究了几十年的资源调度，在不同的上下文中，如广域HPC超级计算网格，工作站网络和大规模服务器集群。 在这里只关注大型服务器集群环境中最有意义的工作。

最近的一些研究分析了来自Yahoo!，Google和Facebook的集群跟踪[20,52,63,68,70,80,82]，并说明了这些现代数据中心和工作负载中固有的规模和异构性的挑战。 [69]包含集群管理器架构的分类。

Apache Mesos [45]使用基于供应的机制在中央资源管理器（有点像Borgmaster减去其调度程序）和多个“框架”（如Hadoop [41]和Spark [73]）之间划分资源管理和放置功能。 Borg主要使用基于请求的机制来集中化这些功能，这种机制可以很好地扩展。 DRF [29,35,36,66]最初是为Mesos开发的; Borg使用优先级和许可配额来代替。Mesos开发商已经宣布扩展Mesos以包括投机的资源分配和回收，并解决[69]中确定的一些问题。

YARN [76]是一个以Hadoop为中心的集群管理器。 每个应用程序都有一个管理器，它与中央资源管理器协商所需的资源; 这与Google MapReduce作业自2008年以来用于从Borg获取资源的方案大致相同.YARN的资源管理器最近才变得容错。 相关的开源工作是Hadoop容量调度器[42]，它为容量保证，分层队列，弹性共享和公平性提供多租户支持。

YARN最近已经扩展到支持多种资源类型，优先级，抢占和高级准入控制[21]。俄罗斯方块研究原型[40]支持工时感知的（makespan-aware）作业打包。

Facebook的Tupperware [64]是一个类似Borg的系统，用于在群集上调度cgroup容器; 只有一些细节已经被公开，尽管它似乎提供了一种资源回收的形式。 Twitter有开源的Aurora [5]，一个类似Borg的调度器，用于在Mesos之上运行的长时间运行的服务，配置语言和状态机类似于Borg。

Microsoft的Autopilot系统提供了“自动化软件配置和部署; 系统监控; 执行修复动作来处理软件和硬件故障“。Borg生态系统提供了类似的功能，但限于篇幅这里不做讨论； Isaard [48]概述了我们坚持的许多最佳实践。

Quincy [49]使用网络流模型为几百个节点的集群上的数据处理DAG提供公平性和数据位置感知调度。Borg使用配额和优先级在用户之间共享资源，并扩展到数万台机器。Quincy直接处理执行图，而这是单独构建在Borg的顶部。

Cosmos [44]专注于批处理，重点是确保其用户能够公平地访问他们捐赠给集群的资源。 它使用每个工作管理器（per-job manager）来获取资源; 几个细节是公开的。

微软的Apollo系统[13]使用每个作业调度器进行短期批处理作业，以在看来与Borg cell大小相同的集群上实现高吞吐量。 Apollo使用机会主义执行低优先级后台工作，以多日排队延迟为代价（有时）来提高利用率。Apollo节点提供任务的开始时间的预测矩阵作为两个资源维度上大小的函数，其中调度器结合启动成本的估计及远程数据访问以进行布置决定，由随机延迟调制以减少冲突。Borg使用中央调度器基于先前分配的状态来布置决定，能处理更多的资源维度，并专注于高可用性、长期运行的应用程序的需求; Apollo可以处理更高的任务到达率。

阿里巴巴的Fuxi [84]支持数据分析工作负载；它从2009年开始运行。像Borgmaster一样，中央FuxiMaster（复制用于容错）从节点收集资源可用性信息，接受来自应用程序的请求，并将一个匹配到另一个。Fuxi增量调度策略与Borg的等价类相反：Fuxi不是将每个任务与一组合适的机器相匹配，而是将新可用资源与积压的待处理工作进行匹配。 像Mesos一样，Fuxi允许定义“虚拟资源”类型。 只有合成工作负载结果是公开的。

Omega [69]支持多个平行的，专门的“垂直”，每个大致相当于Borgmaster减去其持久存储和链接分片。 Omega调度器使用乐观并发控制来操作存储在中央持久存储器中的期望和观察到的cell状态的共享表示，其通过单独的链路组件被同步到Borglet。Omega架构旨在支持多个不同的工作负载，这些工作负载具有特定于应用程序的RPC接口，状态机和调度策略（例如，长期运行的服务器，来自各种框架的批处理作业，基础架构服务（如集群存储系统），虚拟机Google Cloud Platform）。另一方面，Borg提供了“一个适合所有”的RPC接口，状态机语义和调度器策略，随着时间的推移，由于需要支持许多不同的工作负载，它们的规模和复杂性都有所增长，可扩展性尚未成为问题（§3.4）。

Google的开源Kubernetes系统[53]将应用程序放置在Docker容器[28]中（在多个主机节点上）。它运行在裸机（如Borg）和各种云托管提供者中，如Google Compute Engine。它是由许多建立Borg的工程师积极开发的。 Google提供了一个名为Google Container Engine的托管版本[39]。将在下一节讨论如何将Borg的教训应用于Kubernetes。

高性能计算团体在这一领域有着悠久的工作传统（例如，Maui，Moab，Platform LSF [2，47，50]）; 然而，规模、工作负载和容错的要求不同于谷歌的cell。通常，这样的系统通过具有等待工作的大积压（队列）来实现高利用率。

虚拟化提供商如VMware [77]和数据中心解决方案提供商（如HP和IBM [46]）提供集群管理解决方案，通常扩展到O（1000）机器。此外，几个研究团体具有以某些方式改进调度决策质量的原型系统（例如，[25,40,72,74]）。

最后，正如已经指出的，管理大规模集群的另一个重要部分是自动化和“运算符scaleout”。 [43]描述了如何计划故障，多租户，健康检查，准入控制和可重新启动性对于每个操作者完成大量机器是必要的。Borg的设计理念是类似的，能够为每个操作员（SRE）支持数万台机器。

8.经验教训和未来工作

本节讲述了十多年来在生产中操作Borg所学到的一些定性教训，并描述了在设计Kubernetes时如何利用这些观察结果[53]。

8.1经验教训：坏的方面

从Borg的一些特性开始，作为告诫和在Kubernetes中知情的替代设计。

作为任务的唯一分组机制，作业是限制性的。 Borg没有一流的方式来管理整个多作业服务作为一个单一的实体，或指的是服务的相关实例（例如，canary和生产轨迹）。作为一个黑客，用户在作业名称中编码服务拓扑，并构建更高级

别的管理工具来解析这些名称。在范围的另一端，不可能引用作业的任意子集，这导致诸如滚动更新和作业调整大小的不灵活语义问题。

为了避免这种困难，Kubernetes拒绝了作业概念，而是使用标签组织调度单元（pods）- 用户可以附加到系统中任何对象的任意键/值对。同等的，可以通过附加到一个作业上实现Borg作业：将作业名标签附加到一组pod，但是也可以表示任何其他有用的分组，例如服务，层或发行类型（例如，生产、分段、测试）。 Kubernetes中的操作通过标签查询来识别其目标，该查询选择了将应用操作的对象。 这种方法比作业的单个固定分组提供更多的灵活性。

每个机器一个IP地址使事情复杂化。 在Borg中，机器上的所有任务都使用其主机的单个IP地址，从而共享主机的端口空间。这导致了一些困难：Borg必须调度作为资源的端口; 任务必须预先声明需要多少个端口，并且愿意在启动时被告知使用哪些端口；Borglet必须强制端口隔离；并且命名和RPC系统必须处理端口和IP地址。

由于Linux命名空间，虚拟机，IPv6和软件定义网络的出现，Kubernetes可以采取对用户更加友好的方法，消除这些复杂性：每个pod和服务都有自己的IP地址，允许开发人员选择端口，而不是要求软件适应选择 ，并消除了管理端口的基础架构复杂性。

针对高级用户进行优化，牺牲了休闲用户。Borg提供了一大套针对“超级用户”的功能，以便用户可以微调程序运行方式（BCL规范列出约230个参数）：最初的重点是支持谷歌上最大的资源消费者，他们的效率增益是至关重要的。不幸的是，这种API的丰富性使事情变得更难针对“休闲”用户，而限制了其发展。解决方案是构建在Borg之上运行的自动化工具和服务，并通过实验确定合适的设置。这些都受益于容错应用程序提供的实验自由：如果自动化出现错误，这是一个麻烦事，而不是灾难。

8.2经验教训：好的方面

另一方面，一些Borg的设计特性已经非常优越，并经受住了时间的考验。

Allocs是有用的。 Borg alloc抽象概念产生了广泛使用的日志存储模式（§2.4），另一个流行的模式是简单的数据加载器任务定期更新Web服务器使用的数据。Allocs和包允许这种帮助服务由不同的团队开发。

Kubernetes中同alloc等价的是pod，它是一个或多个容器的资源封装，这些容器总是被调度到同一机器上并且可以共享资源。Kubernetes在相同的pod中使用辅助容器代替alloc中的任务，但是想法是一样的。

集群管理不仅仅是任务管理。尽管Borg的主要作用是管理任务和机器的生命周期，但是运行在Borg上的应用程序可以从许多其他集群服务中受益，包括命名和负载平衡。 Kubernetes使用服务抽象支持命名和负载平衡：服务具有名称和由标签选择器定义的动态pod集。群集中的任何容器都可以使用服务名称连接到服务。在封面下，Kubernetes自动负载平衡与标签选择器匹配的pod中的服务连接，并且跟踪pod在由于故障而随着时间重新安排时运行的位置。

内省是至关重要的。 虽然Borg几乎总是“只是工作”，当出了问题，找到根本原因可能是挑战性的。Borg中一个重要的设计决策是要向所有用户显示调试信息，而不是隐藏：Borg有成千上万的用户，所以“自助”必须是调试的第一步。 虽然这使得更难以轻视特性和改变用户依赖的内部策略，但它仍然是一个赢家，还没有找到任何现实的替代品。为了处理大量数据，提供了多个级别的UI和调试工具，因此用户可以快速识别与其作业相关的异常事件，然后从其应用程序和基础架构本身深入查看详细的事件和错误日志。

Kubernetes旨在复制Borg的许多内省技术。 例如，它附带了诸如cAdvisor [15]等用于资源监视和基于Elasticsearch / Kibana [30]和Fluentd [32]的日志聚合的工具。可以查询主机的对象状态的快照。Kubernetes具有统一的机制，所有组件都可以用来记录事件（例如，被调度的pod，容器失败），这些事件对客户端也是可用的。

主机是分布式系统的内核。Borgmaster最初设计为一个单片系统，但随着时间的推移，它变得更像一个内核，位于服务生态系统的核心，协作管理用户作业。例如，将调度程序和主UI（Sigma）拆分为单独的进程，并增加了服务用于准入控制、垂直和水平自动缩放，重新打包任务，定期作业提交（cron），工作流管理以及用于离线查询的归档系统操作。 总之，这使得能够在不牺牲性能或可维护性的情况下扩展工作负载和功能集。

Kubernetes架构更进一步：它的核心有一个API服务器，只负责处理请求和操作底层状态对象。集群管理逻辑被构建为小的可组合的微服务（该API服务器的客户端），例如复制控制器，用于在面对故障时保持pod的期望数量的副本，以及节点控制器，用于管理 机器生命周期。

8.3结论

在过去十年里，几乎所有的Google集群工作负载都转而使用Borg。我们继续发展它，并将从中学到的教训应用到Kubernetes。

摘要：Google的Borg系统是一个运行着成千上万项作业的集群管理器，它同时管理着很多个应用集群，每个集群都有成千上万台机器，这些集群之上运行着Google的很多不同的应用。Borg通过准入控制，高效的任务打包，超额的资源分配和进程级隔离的机器共享，来实现超高的资源利用率。它通过最小化故障恢复时间的运行时特性和减少相关运行时故障的调度策略来支持高可用的应用程序Borg通过提供一个作业声明的标准语言，命名服务的集成机制，实时的作业监控，以及一套分析和模拟系统行为的工具来简化用户的使用。

我们将通过此论文对Borg系统的架构和主要特性进行总结，包括重要的设计决定，一些调度管理策略的定量分析，以及对十年的使用经验中汲取的教训的定性分析。

1.简介

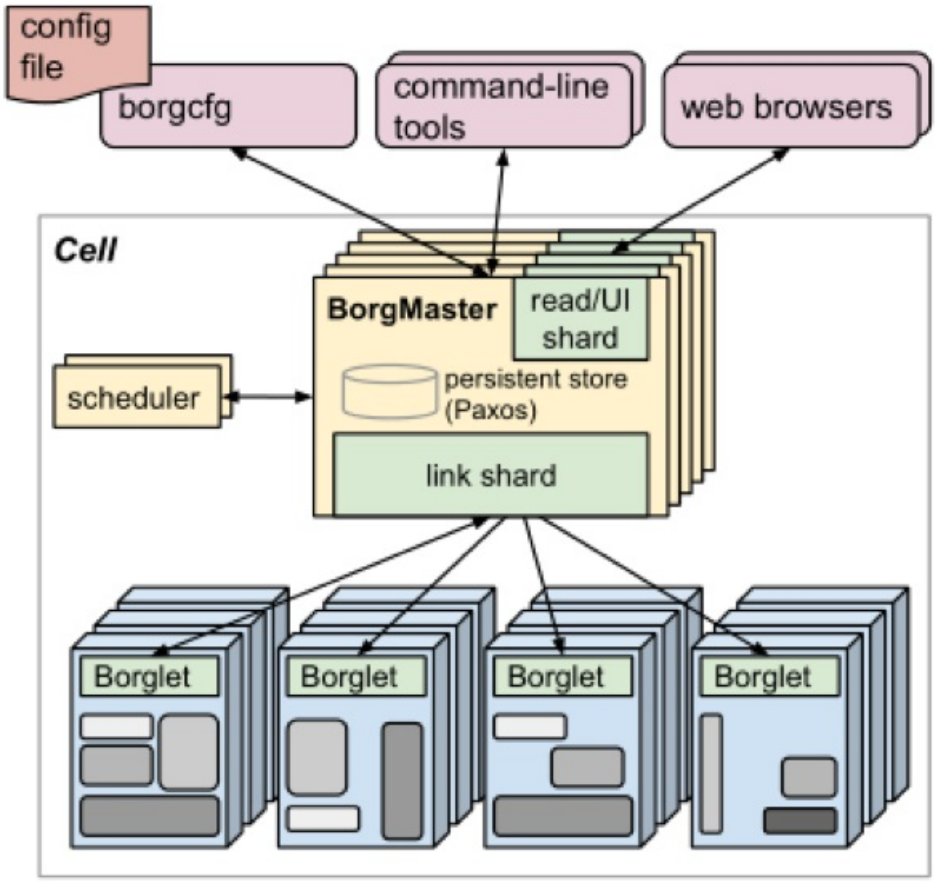


图1 Borg的高级架构。仅显示了成千上万工作节点中的一小部分。

这个在我们内部称为Borg的集群管理系统，它负责权限控制、调度、启动、重新启动和监视全部的Google中运行的应用程序。本文将解释它是如何做到的。

总的来说，Brog主要提供了三个主要的好处：（1）隐藏资源管理和故障处理的细节，因此其用户可以专注于应用程序开发;（2）提供高可靠性和高可用性操作，支持的应用也是如此;（3）使我们能够有效地在数万台机器上运行工作负载。 Borg不是解决这些问题的第一个系统，但它是在能够保证最大弹性和完整性情况下，以大规模运行的少数几个系统之一。 本文将主要围绕这些主题进行组织，并从Borg投入生产，这十多年来的使用经验作为总结 。

2.用户视图

Borg的用户是运行Google应用和服务的Google开发人员和系统管理员（网站可靠性工程师或SRE）。 用户以作业的形式将他们的工作提交给Borg，每个作业包括一个或多个任务，它们都运行相同的程序（二进制）。 每个作业在一个Borg单元中运行，一组机器组织为一个单元。 本节的剩余部分描述了Borg用户视图中展现的主要功能。

2.1 工作负载

Borg的所有单元都同时运行着两种类型的异质工作负载。第一个是“永远运行下去”的长服务，他们对延迟和性能波动敏感， 此类服务用于面向终端用户的产品，例如Gmail，Google文档，web搜索和内部基础设施服务（例如，BigTable）。 第二个是批处理作业，需要花费从几秒到几天完成，这些任务对短期性能波动的敏感性要小得多。 这些工作负载混合运行在Borg的各个运行单元中，其根据其主要租户（例如，一些单元是专门用来运行批量密集任务的）运行不同的混合应用，并且也随时间变化：批处理作业完成和重新运行，许多面向终端用户的服务作业看到日常使用模式。 Borg同样需要处理好所有这些情况。

Borg的代表性工作负载情况可以从2011年5月的一个公开的月份跟踪中找到[80]，已经进行了广泛分析（例如[68]和[1,26,27,57]）。

在过去几年中，许多应用程序框架已经建立在Borg之上，包括我们内部的MapReduce系统[23]，FlumeJava [18]，Millwheel [3]和Pregel [59]。 大多数都有一个控制器提交一个主作业和一个或多个工作作业; 前两者对YARN的应用程序管理器[76]起类似的作用。 我们的分布式存储系统如GFS [34]及其后继CFS，Bigtable [19]和Megastore [8]都运行在Borg上。

对于本文，我们将优先级较高的Borg作业分为“生产”（prod）作业，其余作为“非生产”（non-prod）作业。 大多数长期运行的服务器作业是prod;大多数批处理作业是非prod的。在代表性单元中，分配给prod作业大约总CPU资源的70％，大约占总CPU使用量的60％; 分配给它们约总内存的55％，约占总内存使用的85％。在§5.5节，将看到分配和使用之间的差异将是很重要的。

2.2 集群和单元

单元中的机器属于单个集群，由连接它们的高性能数据中心规模的网络架构定义。

一个集群位于单个数据中心大楼内，大厦集合构成一个站点。一个集群通常承载一个大型单元，可能有一些较小规模的测试或特殊用途单元。 我们努力避免任何单点故障。

中央单元大小是排除测试单元后约10k机器; 有些会更大。一个单元中的机器在许多维度上是异构的：大小（CPU，RAM，磁盘，网络），处理器类型，性能和功能（比如外部IP地址或闪存存储器）。Borg通过确定单元中的运行任务，为任务分配资源，安装程序和其他的依赖，监控任务状态并在失败时重启，将用户从大多数差异中隔离出来。

2.3 作业和任务

Borg作业的属性包括名称，所有者及其拥有的任务数量。作业可能具有限制，使其任务在具有特定属性（例如处理器体系结构，操作系统版本或外部IP地址）的计算机上运行。限制可以是硬的或软的; 软限制就像是偏好而不是要求。作业的开始能被推迟到直到前一个作业完成。 一个作业仅在一个单元中运行。

每个任务映射到在机器上的容器中运行的一组Linux进程[62]。 大多数Borg工作负载不在虚拟机（VM）内运行，因为我们不想支付虚拟化的成本。此外，该系统是在我们对没有硬件的虚拟化支持的处理器进行大量投资的时候设计的。

任务也具有属性，例如资源需求和任务在作业中的索引。 大多数任务属性对作业中的所有任务是相同的，但是可以被重写 - 例如，以提供指定任务的命令行标志。每个资源维度（CPU核，RAM，磁盘空间，磁盘访问速率，TCP端口，等）以细粒度独立指定; 我们不强加固定大小的桶或槽（§5.4）。静态链接Borg程序以减少对其运行时环境的依赖，并且Brog程序被打包为二进制文件和数据文件，由Borg负责安装。

用户通过向Borg发出远程过程调用（RPC）来操作作业，最常见的是通过命令行工具，其他Borg作业或监视系统（§2.6）。大多数作业描述都是用声明性配置语言BCL编写的。BCL是GCL的一个变体[12]，它生成protobuf文件[67]，并扩展了一些Borg特定的关键字。GCL提供lambda函数以允许计算，应用程序可以使用它们来调整环境配置; 成千上万的BCL文件超过1k行长，我们已经积累了数千万行的BCL。Borg作业配置与Aurora配置文件相似[6]。

图2说明了作业和任务在其生命周期中经历的状态。

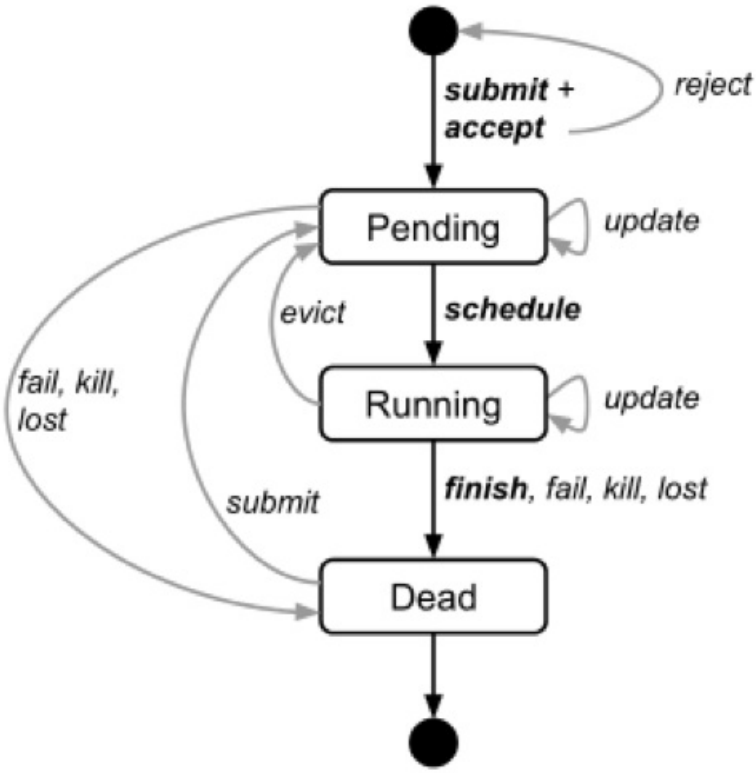


图2：作业和任务的状态图。 用户可以触发提交，终止和更新转换。

用户可以通过推送新的作业配置到Borg，再指示Borg将任务更新到新配置，来更改正在运行的作业中的某些任务或所有任务的属性。 这是一个轻量级的非原子事务，可以很容易地被撤销，直到它被关闭（提交）。 更新通常以滚动方式完成，并且可以对更新导致的任务中断（重新计划或抢占）的数量加以限制; 跳过会导致更多中断的任何更改。

某些任务更新（例如，推送新的二进制）总是需要重启任务; 某些更新（例如，增加的资源需求增加或约束改变）可能使得任务不再适合于这台机器，将导致任务停止并重新调度; 而某些更新（例如，改变优先级）却可在不重新启动或移动任务的情况下进行。

任务可以要求在被SIGKILL抢占之前通过Unix SIGTERM信号获取通知，这样任务就有时间进行清理，保存状态，完成当前正在执行的请求并拒绝新的请求。 如果抢占者设置延迟界限，则实际通知可能更少。 在实践中，通知传递约80％的时间。

2.4 分配

Borg alloc（分配的简称）是可以运行一个或多个任务的机器上的一组保留资源;无论资源是否被使用仍然被分配。Alloc可以用于为将来的任务设置资源，在停止和重启任务之间保留资源，以及将不同作业中的任务收集到同一台机器上 - 例如，Web服务实例和相关的日志保存任务， 这个任务将服务的URL日志从本地磁盘复制到分布式文件系统。 alloc的资源以类似于机器资源的方式处理; 多个任务运行在一个alloc中，共享其资源。如果一个alloc必须重定位到另一台机器，它的任务将被重新调度。

一个alloc集合就像一个作业：它是一组在多个机器上预留资源的alloc。 一旦创建了一个alloc集，可以提交一个或多个作业在其中运行。 简单期间，我们一般会使用“task”来引用alloc或顶层任务（在alloc之外的）和“job”来引用一个作业或alloc集。

2.5 优先级，配额和接纳控制

当更多的工作出现而超过可容纳的限度时会发生什么？我们的解决方案是优先级和配额。

每个作业都有一个优先级，它是一个小的正整数。高优先级任务可以以牺牲低优先级任务为代价而获得资源，即使这导致抢占（杀死）后者。Borg将不同种类的作业分为不同的领域，并给每个领域定义了不重叠的优先权重，这些作业组包括：监视作业，生产作业，批处理作业和尽力而为的作业（已知的包括测试程序），他们的优先级依次递减。对于本文，prod作业是监测和生产领域的工作。

虽然被抢占的任务通常将被重新安排在单元中的其他地方，抢占级联可能发生，如果高优先级的任务碰到一个略低优先级的任务，而这个略低优先级任务又引发另一个略低优先级的任务，等等。为了消除大部分这种情况，我们不允许生产领域中的任务相互抢占。 细粒度优先级在其他情况下仍然有用 - 例如，MapReduce主任务以比他们控制的workers更高的优先级运行，来提高其可靠性。

优先级表示单元中正在运行或正等待运行的作业的相对重要性。 配额用于决定允许进行调度的作业。 配额表示为在给定优先级上的一段时间（通常为几个月）内的资源量（CPU，RAM，磁盘等）的向量。 数量指定用户的作业请求可以一次请求的资源的最大量（例如，“从现在直到7月底在单元xx中的prod优先级的20TiBRAM“）。 配额检查是许可控制的一部分，而不是调度：配额不足的作业立即拒绝提交。

较高优先级配额的成本高于较低优先级配额。生产优先级配额仅限于单元中可用的实际资源，因此，提交符合配额的生产优先级作业的用户可以预期运行。 即使我们鼓励用户购买的配额不超过他们的需求，但是许多用户仍然过度购买，因为这帮助他们在应用程序的用户群增长时克服不足。我们通过在较低优先级别上过度销售配额来响应这一点：每个用户具有在优先级零的无限配额，尽管这常常难以执行，因为资源被过度订阅。一个低优先级作业可能被允许了，但是由于资源不足而保持等待（未调度）。

在Borg以外进行配额分配，并且与我们的物理容量规划密切相关，其结果反映在不同数据中心的配额的价格和可用性上。 仅当用户作业具有所需优先级的足够配额时，才允许用户作业。 配额的使用减少了对优势资源公平（DRF）[29,35,36,66]等策略的需要。

Borg有一个能力系统，能给予一些用户特殊的权限; 例如，允许管理员删除或修改单元中的任何作业，或允许用户访问受限内核功能或Borg行为（例如禁用其作业的资源估计（§5.5））。

2.6 命名和监控

仅创建和放置任务是不够的：服务的客户端和其他系统需要能够找到它们，即使它们被重定位到新机器上了。要启用此功能，Borg将为每个任务创建一个稳定的“Borg name service”（BNS）名称，其中包含单元名称，作业名称和任务编号。Borg将任务的主机名和端口写入一个以BNS命名的一致的高可用的Chubby [14]文件中，由我们的RPC系统使用该文件来查找任务端点。BNS名称还形成任务的DNS名称的基础，所以在cc单元中的用户ubar拥有的作业 jfoo中的第五十个任务将通过50.jfoo.ubar.cc.borg.google.com访问到。Borg还会在Chubby发生变化时将作业大小和任务健康信息写入Chubby，因此负载平衡器可以查看将请求路由到哪里。

几乎在Borg下运行的每个任务都包含一个内置的HTTP服务器，它发布有关任务运行状况的信息和成千上万个性能指标（例如RPC延迟）。 Borg监控health-check URL，并重新启动不会及时响应或返回HTTP错误代码的任务。 其他数据由仪表盘监视工具和违反服务级别目标（SLO）的警报进行跟踪。

称为Sigma的服务提供了基于Web的用户界面（UI），通过该UI用户可以检查所有作业，特定单元的状态，或向下钻取到单个作业和任务，以检查其资源行为，详细日志，执行历史 ，和最终的结果。 我们的应用产生大量日志; 这些被自动轮转以避免用完磁盘空间，并在任务退出后保存一段时间以协助调试。 如果作业未运行，Borg提供了“为什么待处理？”注释，以及如何修改作业的资源请求以更好地适应单元的指导。 我们发布了“切合”更可能容易调度的资源形式的规则。

Borg记录所有作业提交事件和任务事件，以及每个任务在Infrastore中详细的资源使用信息，这是一个可扩展的只读数据存储，通过Dremel [61]具有一个交互式的类似SQL的界面。此数据用于基于使用的计费，作业调试和系统故障以及长期容量规划。 它还为Google群集工作负载跟踪提供数据[80]。

所有这些功能都有助于用户理解和调试Borg的行为及用户的作业，并帮助我们的SREs为每个人管理几万台机器。

3.Borg体系结构

Borg单元由一组机器，一个称为Borgmaster的逻辑中央控制器和单元中每台机器上运行的称为Borglet的代理进程构成（参见图1）。 Borg的所有组件都用C ++编写。

3.1 Borgmaster

每个单元的Borgmaster包括两个进程：主进程Borgmaster和独立的调度程序（§3.2）。 主Borgmaster进程处理客户端RPC，状态变化（例如，创建作业）或提供对数据的只读访问（例如，查找作业）。它还管理系统中所有对象（机器，任务，分配等）的状态机，与Borglets进行通信，并提供Web UI作为Sigma的备份。

Borgmaster在逻辑上是一个单一的进程，但实际上被复制了五次。 每个副本维护了一份该单元大部分状态的内存副本，并且该状态也记录在该副本的本地磁盘上的高可用性，分布式，基于Paxos的存储[55]中。每个单元的单个选定的master既用作Paxos的领导者又用作状态mutator，处理改变单元状态的所有操作（例如提交作业或在机器上终止任务）。当cell建立时或只要当选择的master出现故障时，就会选择一个master（使用Paxos）; 它获取一个Chubby锁，以便其他系统可以找到它。选择一个master和故障转移到新的master通常需要大约10s，但是在大单元中可能需要一分钟，因为一些内存中的状态必须重建。当副本从中断恢复时，它将自动重新同步来自最新的其它Paxos副本的状态。

Borgmaster在某个时间点的状态称为检查点，并采用定期快照的形式增加一条更改日志（保存在Paxos存储中）。检查点有许多用途，包括将Borgmaster的状态恢复到过去的任意一个点（例如，在接受触发Borg中的软件缺陷的请求之前，以便可以对其进行调试）; 构建用于未来查询的事件的持久日志; 以及离线模拟。

高保真的Borgmaster模拟器Faokemaster可用于读取检查点文件，并包含产生Borgmaster代码的完整副本，其中包含与Borglets的无存根接口。它接受RPC进行状态机更改和执行操作，如“调度所有挂起的任务”，通过与它进行交互（它就像是一个活的Borgmaster，带有模拟的Borglets可从检查点文件重放真实的交互），可以使用它来调试故障。用户可以逐步观察在过去实际发生的系统状态的改变。 Fauxmaster对于容量规划（“符合多少这种类型的新作业？”）以及在更改单元配置之前进行完整性检查（“这种更改是否会驱逐重要的工作？”）也很有用。

3.2 调度

提交作业时，Borgmaster会将其持久化在Paxos存储中，并将作业的任务添加到等待队列。 这是由调度程序异步扫描的，如果有足够的可用资源满足作业的要求，则会将任务分配给机器。（调度程序主要操作任务，而不是作业。）扫描从高到低优先级，由优先级循环方案调度，以确保用户之间的公平性，并避免大型作业后面的队头阻塞。 调度算法有两个部分：可行性检查（用于找到任务可以运行的机器），以及评分（用于挑选一个可行的机器）。

在可行性检查中，调度器找到满足任务需求的一组机器，这组机器具有足够的“可用”资源 - 这些资源中包括已经分配给可以被抢占的较低优先级任务的资源。 在评分中，调度器确定每个可行机器的“良好性”。该分数考虑了用户指定的偏好，但主要是由内置标准决定，如最大限度地减少抢占任务的数量和优先级，选择已经有任务包副本的机器，跨越电源和故障域传播任务，以及打包质量（包括将高优先级任务和低优先级任务混合到单个机器上，以允许高优先级任务在负载高峰中扩展）。

Borg最初使用E-PVM [4]的变体进行评分，其在不同资源上生成单一成本值，并且在放置任务时最小化成本的变化。在实践中，E-PVM最终在所有机器上扩展负载，为负载高峰留下余量 - 但是以增加碎片为代价，特别是对于需要大部分机器的大型任务; 我们有时称之为“刚好合适”。

调度的另一端是“最佳合适”，它试图尽可能紧密地填充机器。 这使一些机器没有用户作业（它们仍然运行存储服务器），因此放置大任务是简单直接的，但是严格的封装不利于用户或Borg对资源需求的任何错误估计。 这会伤害突发负载的应用程序，对于指定低CPU需求的批处理作业尤其糟糕，以便他们可以轻松安排并尝试在未使用的资源中伺机运行：20％的非生产任务请求少于0.1个CPU内核。

我们当前的评分模型是一种混合式的，它试图减少搁置资源的数量 - 由于机器上的另一个资源被完全分配而无法使用的资源。它提供比最适合我们工作负载约3-5％的更好的包装效率（在[78]中定义）。

如果计分阶段选择的机器没有足够的可用资源来满足任务，则Borg会抢占（杀死）较低优先级任务，从最低优先级到最高优先级，直到满足为止。 我们将被抢占的任务添加到调度程序的挂起队列，而不是迁移或休眠它们。

任务启动延迟（从作业提交到任务运行的时间）是一个已经并继续受到极大关注的领域。它是高度可变的，中值通常约25s。软件包安装大约占全部的80％：其中一个已知的瓶颈是软件包要写入的本地磁盘的争用。为了减少任务启动时间，调度程序更倾向将任务分配给已经安装了必要的软件包（程序和数据）的机器：大多数软件包是不可变的，因此可以共享和缓存。（这是Borg调度程序支持数据本地化的唯一形式。）此外，Borg使用类似树和torrent的协议并行地将软件包分发到机器。

此外，调度程序使用几种技术来扩展具有成千上万台机器的单元（§3.4）。

3.3 Borglet

Borglet是一个本地Borg代理，存在于单元中的每一台机器中。它启动和停止任务; 如果故障就重启任务; 通过操纵操作系统内核设置来管理本地资源; 翻转调试日志; 并向Borgmaster等监控系统报告机器的状态。

Borgmaster每隔几秒钟轮询一次Borglet以检索机器的当前状态，并将所有未完成的请求发送给它。 这使Borgmaster控制通信速率，避免了显式流控制机制的需要，并防止恢复风暴[9]。

选定的master负责准备要发送到Borglets的消息，并负责根据cell的响应更新cell的状态。为了性能可扩展性，每个Borgmaster副本运行无状态链接分片来处理与一些Borglets的通信;每当发生Borgmaster选择时重新计算分区。对于弹性，Borglet始终报告其完整状态，但链接分片通过仅报告状态机间的差异来收集和压缩此信息，以减少选定master的更新负载。

如果Borglet没有响应几个轮询消息，它的机器被标记为关闭，并且其运行的任何任务被重新安排在其他机器上。如果通信恢复，Borgmaster会通知Borglet要停止这些已经重新安排的任务，以避免重复。即使与Borgmaster失去联系，Borglet也继续正

常运行，因此即使所有Borgmaster副本故障了，当前运行的任务和服务也会保持。

3.4可扩展性

我们不确定Borg的集中式架构的最终可扩展性限制将出现在何处; 到目前为止，每次我们接近一个极限，我们已经设法消除它。一个Borgmaster可以管理一个cell中的数千台机器，并且几个cell具有每分钟超过10000个任务的到达速率。繁忙的Borgmaster使用10-14个 CPU内核和高达50GiB 的RAM。我们使用几种技术来实现这种规模。

早期版本的Borgmaster有一个简单的，同步的循环：接受请求，计划任务，并与Borglets通信。为了处理更大的cell，我们将调度程序分离出来作为一个单独的进程，这样它可以与其他Borgmaster函数并行操作故障容限。 调度器副本对单元状态的高速缓存副本进行操作。它反复：从选定的主机检索状态更改（包括已分配和挂起的工作）; 更新其本地副本;执行调度传递以分配任务; 并将这些分配通知选定的主机。master将接受并采用这些分配，除非它们是不适当的（例如，基于过期状态），这将导致它们在调度程序的下一次传递中被重新考虑。这在灵魂上与在Omega [69]中使用的乐观并发控制非常相似，事实上，我们最近为Borg添加了针对不同工作负载类型使用不同调度程序（schedulers）的能力。

为了提高响应时间，我们添加了单独的线程来与Borglets进行通信并响应只读RPC。为了更好的性能，我们在五个Borgmaster副本（§3.3）中分割（分区）这些功能。同时，这保持了UI上99%ile的响应时间低于1s和95%ile 的Borglet轮询间隔低于10s。

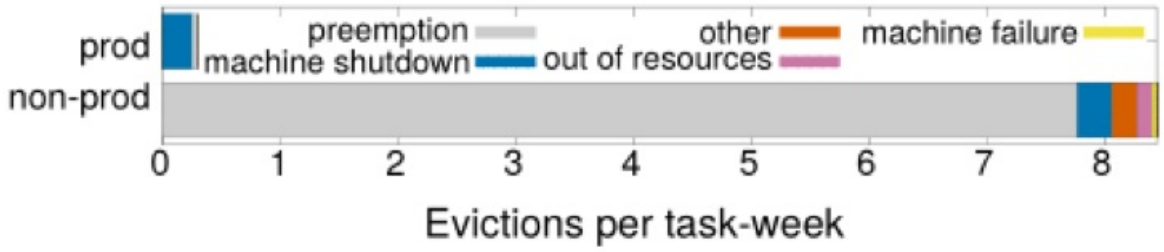


图3：针对生产和非生产工作负载的任务驱逐率及原因。 数据自2013年8月1日起。

有几点使Borg调度器更具可扩展性：

分数缓存：评估可行性和评价机器是昂贵的，因此Borg缓存分数直到机器或任务的属性改变 - 例如，机器上的任务终止，属性改变或任务的需求改变。 忽略资源数量的小变化可减少高速缓存失效。

等价类：Borg作业中的任务通常具有相同的需求和约束，因此并不是确定每个机器上的每个挂起任务的可行性，并对所有可行的机器进行评分，Borg只对每个等价类的一个任务进行可行性分析和评分 - 一组具有相同需求的任务。

轻松随机化：计算大cell中所有机器的可行性和分数是浪费的，因此调度程序以随机顺序检查机器，直到找到“足够”可行的机器进行评分，然后选择该集合中的最佳机器。 这减少了任务进入和离开系统时所需的评分和高速缓存失效的数量，并加快了任务到机器的分配。放松随机化有时类似于Sparrow [65]的批量采样，同时还处理优先级，抢占，异质性和软件包安装的成本。

在我们的实验（§5）中，从头开始安排单元的整个工作负载通常需要几百秒，但是在禁用上述技术后超过3天后还没有完成。通常，在等待队列上的在线调度传递在不到半秒内完成。

4.可用性

故障是大规模系统中的常态[10,11,22]。图3提供了15个样本cell中任务驱逐原因的分解。运行在Borg上的应用程序应能使用诸如复制，在分布式文件系统中存储持久状态并（如果适当的话）捕捉临时检查点等技术来处理此类事件。即使如此，我们也试图减轻这些事件的影响。例如，Borg：

如有必要，在新机器上自动重新安排逐出的任务;

通过在诸如机器，机架和电源域之类的故障域中扩展作业的任务，减少相关故障;

限制任务中断的允许速率和任务数量，这些任务可以在维护活动（例如操作系统或机器更新）期间同时关闭；

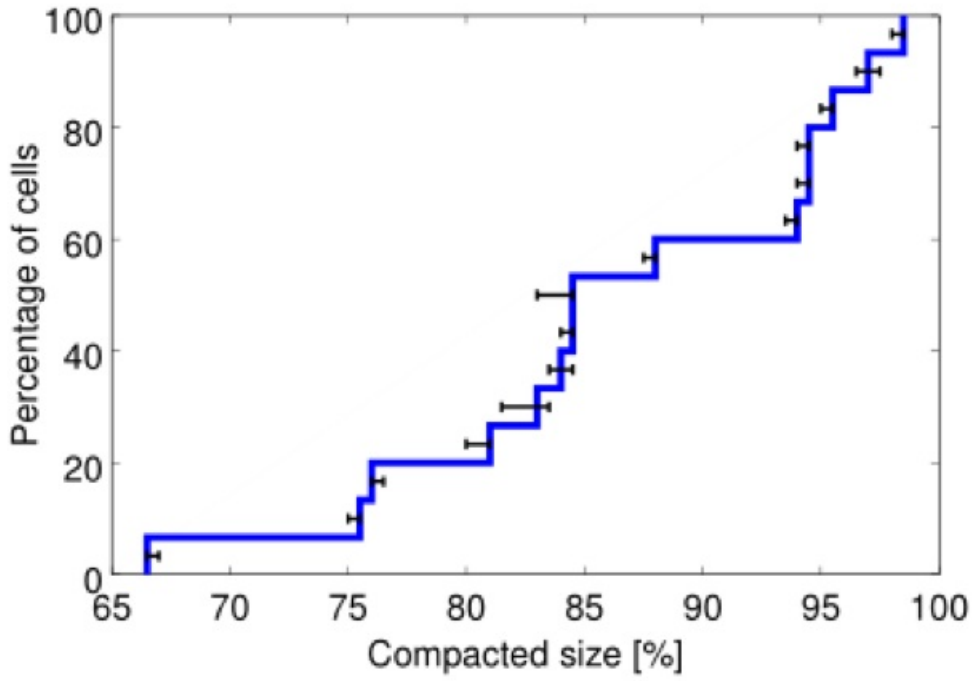


图4：压缩的效果。 对15个cell，在压缩后获得的原始cell大小的百分比的CDF。

使用声明性期望状态表示和幂等变换操作，使得失败的客户端可以无损地重新提交任何被遗忘的请求;

rate-limits找到无法访问的机器的任务的新位置，因为它无法区分大型机器故障和网络分区;

避免重复任务::导致任务或机器崩溃的机器配对;

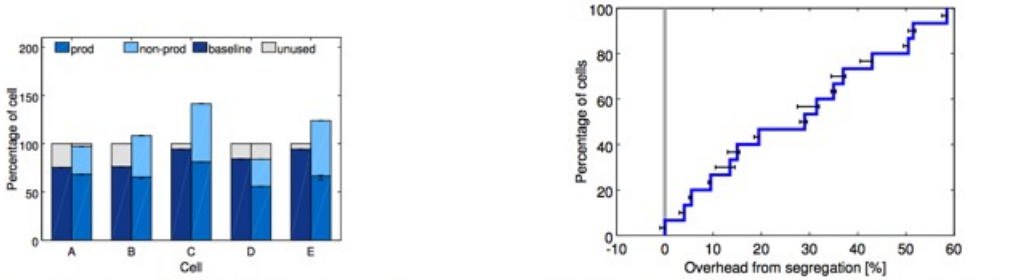
通过不断重新运行日志记录器任务（§2.4）来恢复写入本地磁盘的关键中间数据，即使所连接的alloc已终止或移动到了另一台机器。 用户可以设置系统持续尝试的时间;一般是几天。

Borg的一个关键设计特点是，即使Borgmaster或任务的Borglet关闭，已经运行的任务也会继续运行。但是保持master仍然很重要，因为当它关闭时，无法提交新作业或更新现有的作业，并且无法重新计划故障的计算机上的任务。

Borgmaster使用的技术组合，使其在实践中达到了99.99%的可用性：机器故障复制; 准入控制避免过载; 并使用简单的低级工具部署实例以最小化外部依赖性。 每个单元独立于其他单元，以最小化关联的操作者错误和故障传播的机会。 这些目标，不是可扩展性限制，而是反对较大cell的主要论证。

5.利用

Borg的主要目标之一是高效利用Google的机器，这意味着巨大的财务投资：将利用率提高几个百分点可以节省数百万美元。本节讨论和评估Borg使用的一些策略和技术。



(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.*

图5：将prod和non-prod工作分离到不同的单元将需要更多的机器。这两个图表显示如果prod和non-prod工作负载发送到单独的单元，需要多少额外的机器，使用百分比表示在一个单元中运行工作负载所需的最小机器数量。在当前和随后的CDF图中，每个单元显示的值推导自我们的实验尝试产生的不同cell大小的90%，误差条显示了尝试值的完整范围。

5.1 评估方法

我们的工作存在布局限制，需要处理稀少的工作负载峰值，机器是异构的，在从服务作业中回收的资源中运行批处理作业。所以，为了评估策略选择，需要一个比“平均利用率”更复杂的度量标准。 经过多次实验，我们选择了cell compaction：给定一个工作负载，通过删除cell中的机器直到不再适应，可以发现它能适应的cell有多小，，从头开始重新包装工作负载以确保没有挂在一个糟糕的配置。 这提供了干净的最终条件，促进了自动化比较（没有合成工作生成和建模的陷阱 [31]）。评价技术的定量比较可以在[78]中找到：细节是令人惊讶的微妙。

不可能对真实生产单元进行实验，但可以使用Fauxmaster获得高保真的模拟结果，使用来自实际生产单元和工作负载的数据，包括其所有约束、实际限制、保留和使用数据（§5.5 ）。 这些数据来自2014-10-01 14：00 PDT的Borg检查点。 （其他检查点产生了类似结果）。选择15个Borgcell进行报告，首先消除特殊用途、测试和小（<5000台机器）cell，然后对剩余的cell进行取样，以获得大小均匀的范围。

为了在压缩的cell中维持机器异构性，随机选择机器进行删除。 为了保持工作负载的异构性，保留除了与特定机器（例如，Borglets）绑定的服务器和存储任务外的一切。为大于原cell大小一半的作业更改硬约束为软约束，并允许多达0.2%的任务等待，如果他们非常“挑剔”并只能放置在少数几台机器上; 广泛的实验表明，这产生了低方差的可重复结果。 如果我们需要比原cell更大的cell，可以在压缩之前克隆原始cell几次; 如果我们需要更多cell，只需要克隆原来的cell即可。

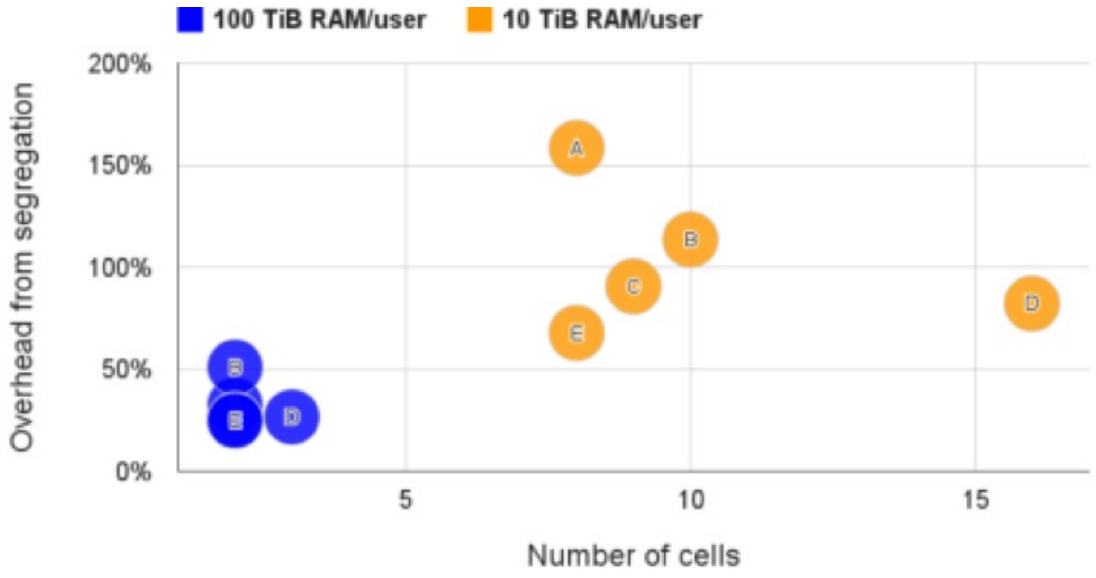
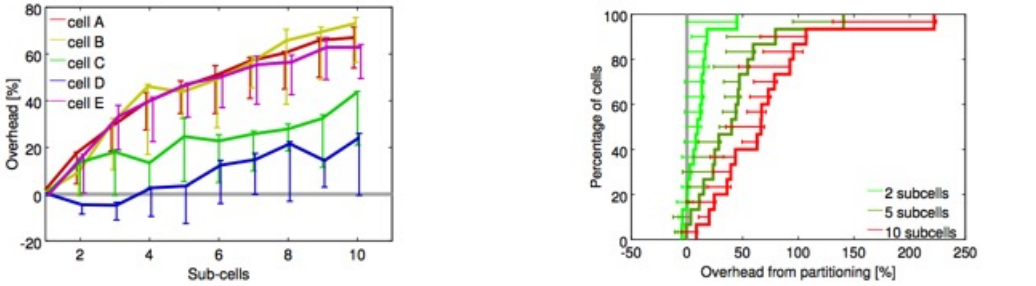


图6：分离用户将需要更多的机器。如果用户多于所示的阈值，针对5个不同的cell，给出了各私有单元的 Cell总数和需要的额外机器。

对于具有不同随机数种子的每个cell，每个实验重复11次。 在图表中，我们使用错误栏显示所需机器数量的最小值和最大值，并选择90%ile值作为“结果”-平均值或中位数并不反映系统管理员如果希望合理地确保工作负载适合时将做什么。 我们认为cell压缩提供了一种公平、一致的方法来比较调度策略，它直接转化为成本/效益结果：更好的策略需要更少的机器来运行相同的工作负载。

我们的实验集中在从一个时间点调度（打包）工作负载，而不是重放长期工作负载跟踪。 这部分是为了避免应对开放和封闭排队模型的困难[71,79]，部分原因是传统的完成时间指标不适用于长期运行服务的环境，部分是为了提供清晰的信号 进行比较，部分是因为我们不相信结果会有明显的不同，部分是一个实际问题：我们发现自己消耗了200000个Borg CPU核用于实验 - 即使按谷歌的规模，这也是一个非平凡的投资 。

在产品中，针对工作负载的增长留出有效的空间，偶然的“black swan”事件，负载高峰，机器故障，硬件更新，以及大规模局部故障（e.g., a power supply bus duct）。图4显示了如果应用了cell压缩，真实的cell有多小。图中跟随的基线使用了压缩的大小。



(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.*

图7：将cell细分成更小的cell将需要更多的机器。 如果将这些特定cell划分为不同数量的较小cell，则需要额外的机器（作为单个cell情况的百分比）。

5.2 cell共享

几乎所有机器同时运行着prod 和 non-prod任务：在共享的Borg cell中98%的机器，有83%跨越Borg管理的整个的机器集合。（有一些特殊用途的专用cell）。

由于很多其它组织在单独的集群中运行面向用户或批量作业，我们对如果我们也这样做会发生什么做了调查。图5显示了分离开proc和non-proc工作，在中值的cell中将需要20-30%的更多的机器来运行工作负载。这是因为proc作业总是会预留资源来处理稀有的负载高峰，但大部分时间并不使用这些资源。Borg回收利用这些未使用的资源来运行大部分non-proc工作，这样总

体上就需要更少的机器了。

大部分Borg cell被数以千计的用户共享。图6显示了原因。对这个测试，如果用户的工作负载消耗了至少10TiB（或者100 TiB）的内存，就分离用户的工作负载到一个新的cell。现存的策略看起来是好的：即使是更大的极限，将需要2–16×作为一些cell，以及20–150%的额外的机器。再次，合并资源显著减少了成本。

但是可能将不相关的用户和作业类型打包到同一台机器上会导致CPU干扰，因此需要更多的机器来弥补？ 为了评估这一点，我们研究了在具有相同时钟速度的相同机器类型上运行的不同环境中的任务的CPI（每个指令的周期数）如何改变。 在这些条件下，CPI值是可比的，并且可以用作针对性能干扰的代理，因为CPI的翻倍加倍了CPU绑定程序的运行时间。在一个星期内从约12000个随机选择的prod任务收集数据，使用[83]中描述的硬件配置架构，在5分钟的间隔内对周期和指令进行计数，并对样本进行加权，使得每秒钟的CPU时间被公平计数。 结果并不清晰。

（1）我们发现CPI与在相同时间间隔内的两个测量结果正相关：机器上的总体CPU使用率，（很大程度上独立地）机器上的任务数量; 向机器添加任务使得其他任务的CPI提高0.3%（使用适合这些数据的线性模型）; 将机器CPU使用率提高10%使得CPI提高小于2%。 但是即使相关性在统计上是重要的，也只显示了在CPI测量中看到的方差的5%; 其他因素占主导地位，例如应用的固有差异和特定的干扰模式[24,83]。

（2）将我们从共享cell中采样的CPI与来自具有较少不同应用的几个专用cell的CPI进行比较，我们看到共享cell的平均CPI为1.58（σ= 0.35），专用cell的平均CPI为1.53（σ= 0.32）- 即，CPU在共享cell中性能降低约3%。

（3）为了解决不同cell中的应用可能具有不同工作负载或还遇到选择偏差（可能将干扰更敏感的程序已经移动到专用cell）的担忧，考察了Borglet的CPI，它在两种类型的所有机器上运行。它在专用cell中的CPI为1.20（σ= 0.29），在共享cell中的CPI为1.43（σ= 0.45），表明它在专用cell中的运行速度为1.19倍，与在共享单元中一样快，虽然这加重了轻负载机器的影响，但轻微偏差结果有利于专用cell。

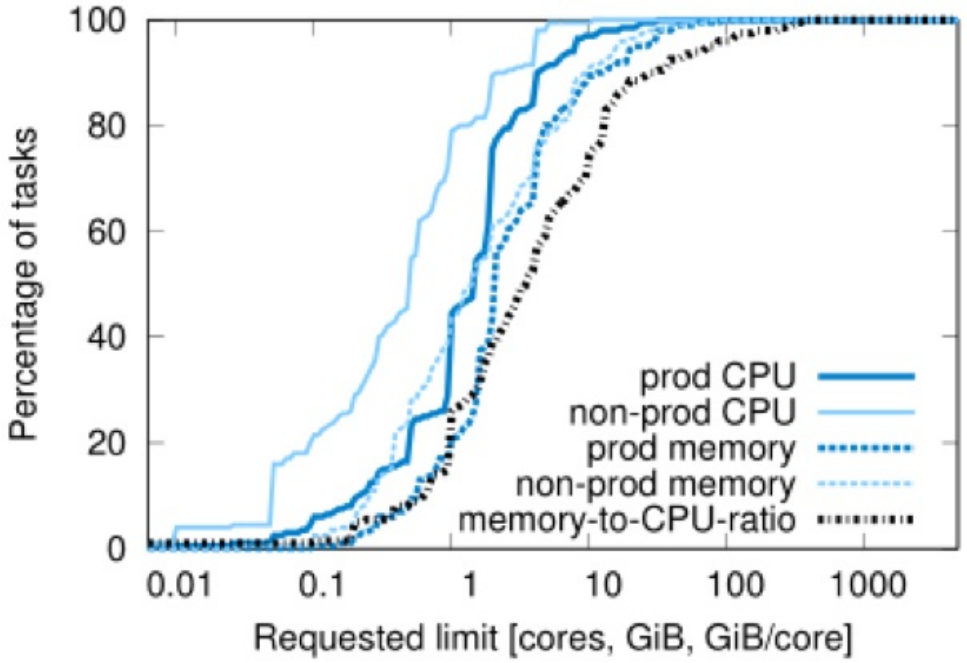


图8：没有适合大多数任务的桶大小。请求的CPU和内存的CDF要求跨越样本单元。

没有一个值突出，虽然几个整数CPU核大小有点更受欢迎。

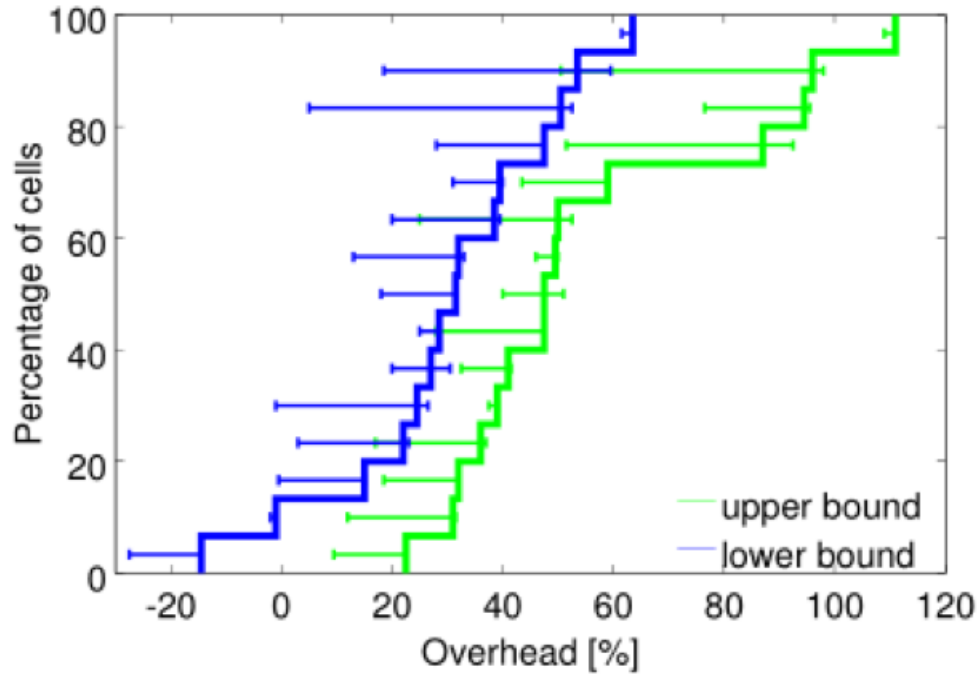


图9：“Bucketing”资源需求将需要更多的机器。

将15个单元中的CPU和存储器请求四舍五入为下一个最接近2的幂所产生的额外开销的CDF。 下限和上限跨越实际值（见文本）。

这些实验证实，仓库规模的性能比较是棘手的，在[51]中加强了观察，并且还表明共享不会大大增加运行程序的成本。

但即使假设最不利的结果，共享仍然是一个胜利：CPU减速超过了几种不同划分方案所需机器的减少，共享优点适用于所有资源，包括内存和磁盘，而不仅仅是CPU 。

5.3 大细胞

Google构建了大cell，以允许运行大型计算，并减少资源碎片。通过将cell的工作负载分到多个较小的cell来测试后者的效果 - 首先随机排列作业，然后在分区之间以循环方式分配作业。 图7证实使用较小cell将明显需要更多的机器。

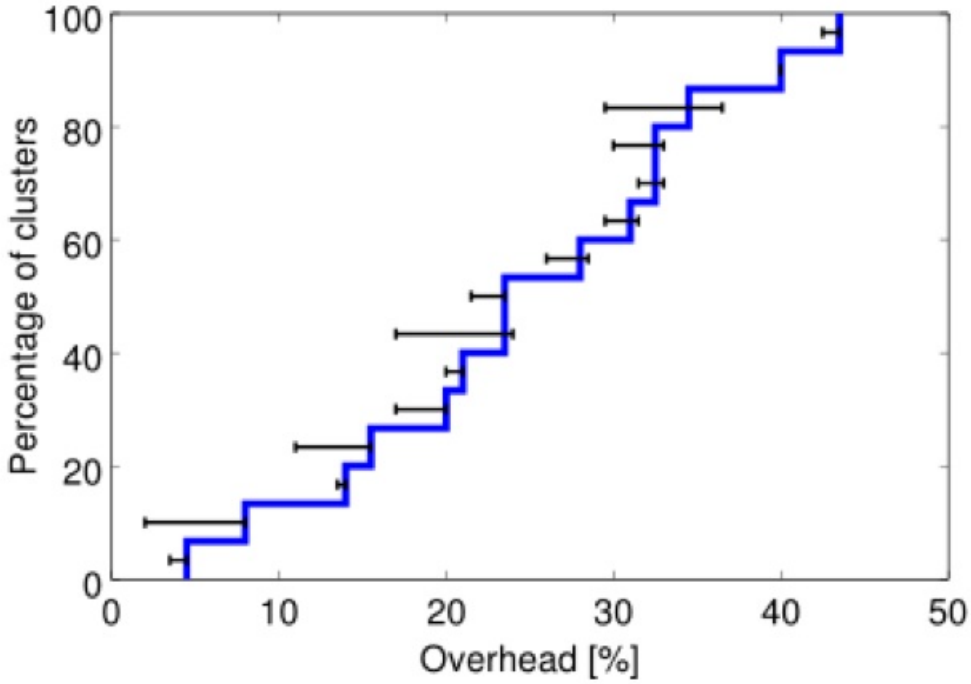


图10：资源回收是相当有效的。 如果禁用15个代表性cell，则需要额外机器的CDF。

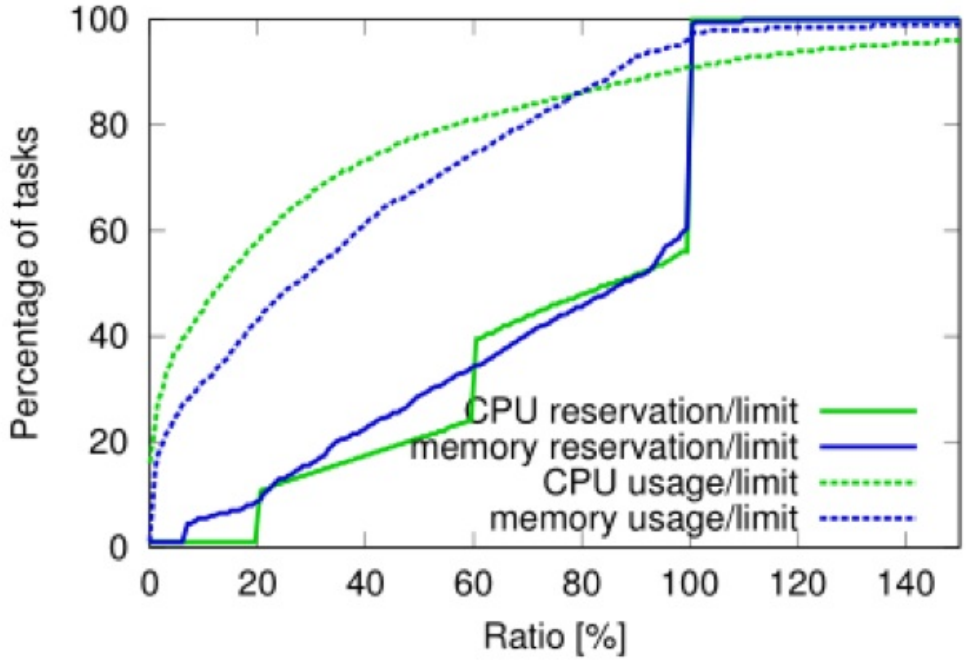


图11：资源估计在识别未使用的资源方面是成功的。虚线显示了15个cell中的任务请求（极限）的CPU和内存使用率的CDF。大多数任务使用远远低于其极限，虽然少数比请求的使用了更多的CPU。实线显示出了CPU和内存保留限制率的CDF; 这些都更接近100%。直线是资源估计处理的伪像。

5.4 细粒度资源请求

Borg用户请求CPU以milli-cores为单位，内存和磁盘空间以字节为单位。（core是处理器超线程，针对机器类型的性能进行标准化）。图8显示了利用这种粒度：在所请求的内存或CPU核的数量上几乎没有明显的“sweet spots”，并且这些资源之间几乎没有明显的相关性。除了在90%及以上的内存请求稍大之外，这些分布与[68]中提出的分布非常相似。

提供一组固定大小的容器或虚拟机虽然在IaaS（基础设施即服务）提供商[7,33]中很常见，但不能很好地满足我们的需求。为了说明这一点，通过在每个资源维度上将它们四舍五入到下一个最接近的2的幂，从CPU的0.5内核和RAM的1GiB开始，对prod作业和分配（§2.4）“bucketed”CPU核和内存资源限制。图9显示这样做将需要比平均情况下多30-50%的资源。上限来自于将整个机器分配给大型任务（在压缩前将原始细胞翻两番后不适合）; 下限来自于允许这些任务进入待定状态。（这小于[37]中报告的大约100%的开销，因为我们支持超过4个buckets，并允许CPU和RAM容量独立扩展。）

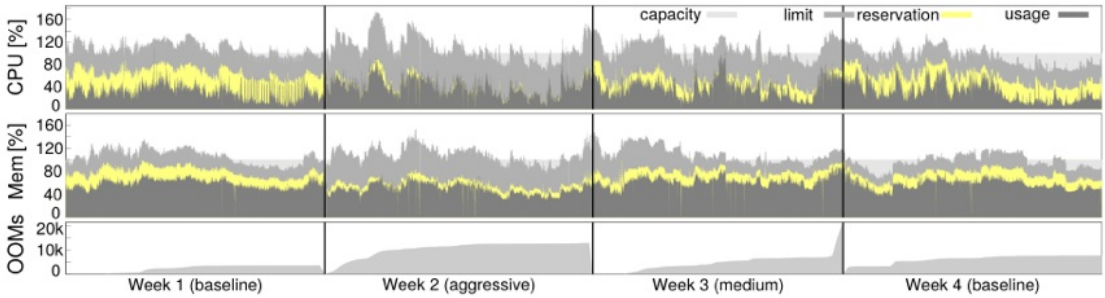


图12：更积极的资源估计可以回收更多的资源，对内存不足事件（OOM）几乎没有影响。一个生产cell使用的时间线（从2013-11-11开始），5分钟窗口平均的预留和限制以及累积的内存不足事件; 后者的斜率是OOM的总速率。竖栏使用不同的资源估计设置按周分开。

5.5 资源回收

作业可以指定资源限制 - 每个任务应被授予的资源的上限。 Borg使用该限制来确定用户是否有足够的配额来接受作业，并确定特定机器是否有足够的免费资源来安排任务。 正如有用户购买比所需要的更多的配额，有用户请求比任务将使用的更多的资源，因为Borg通常会杀死一个试图使用比它需要的更多的RAM或磁盘空间的任务，或节流CPU到 任务所要求的。 此外，某些任务偶尔需要使用其所有资源（例如，在一天的高峰时间或在应对拒绝服务攻击时），但大多数时间不会。

不是浪费当前未被消耗的已分配资源，而是估计任务将使用多少资源，并回收可以容忍低质量资源（例如批处理作业）的工作的剩余资源。这整个过程称为资源回收。 该估计称为任务的预留，并且由Borgmaster每几秒钟使用由Borglet捕获的细粒度使用（资源消耗）信息来计算。初始预留被设置为等于资源请求（限制）;在300s后，为了允许瞬间启动，缓慢向实际使用加上安全余量衰减。如果使用超过，预订会迅速增加。

Borg调度程序使用极限来计算prod任务的可行性（§3.2），因此调度程序从不依赖于已回收的资源，也没有暴露给资源超额订阅; 对于non-proc任务，使用现有任务的预留，以便将新任务安排到已回收的资源中。

机器在运行时可能会耗尽资源，如果这个保留（预测）是错误的 - 即使所有任务使用的资源小于其极限。 如果发生这种情

况，杀死或限制non-proc任务（永不处理proc任务）。

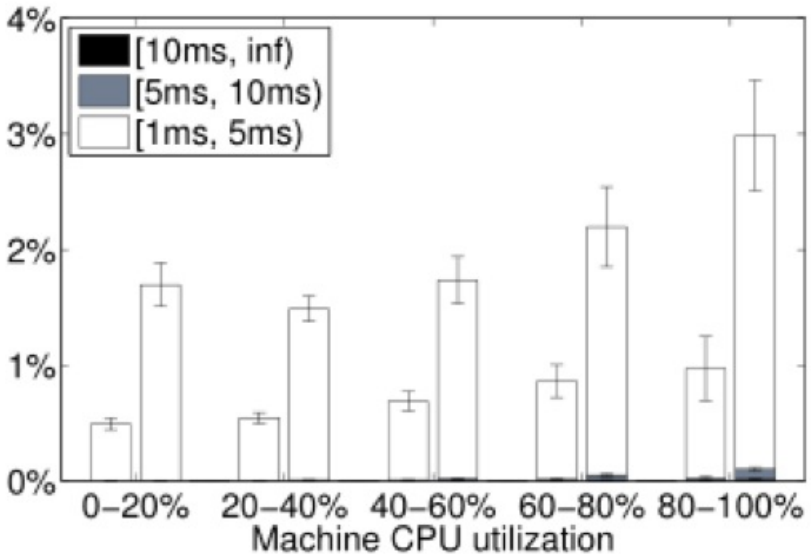


图13：调度延迟作为负载的函数。 描绘了一个可运行的线程必须等待多于1ms的时间来访问CPU的频率，并作为机器有多忙的函数。 在每对竖栏中，延迟敏感的任务在左侧，批处理任务在右侧。仅在几个时间的百分比中，线程必须等待超过5ms才能访问CPU（白色栏）;其它几乎不必等待更长的时间（更黑的栏）。来自从 2013年12月以来代表性单元格的数据；错误栏显示每日差异。

图10显示了更多的机器无需资源回收。大约20％的工作负载（§6.2）在中值cell的回收资源中运行。

可以在图11中看到更多细节，其中显示了预留和使用限制的比率。 如果需要资源，超过其内存限制的任务将首先被抢占，而不管其优先级如何，因此任务超过其内存限制是很少见的。 另一方面，CPU可以容易地被节流，因此短期峰值可以相当无害地推动使用高于预留（push usage above reservation fairly harmlessly）。

图11表明资源回收可能是无必要保守的：在预留和使用线之间存在明显的区域。 为了测试这一点，选择了一个活的生产cell，并通过减少安全边界，在第一周调整资源估计算法的参数到一个积极的设置，然后在下一周调整到基线和积极设置之间的中间设置，然后恢复到基线。图12显示发生了什么。在第二周 保留明显更接近使用，第三周稍差，基准周（第一和第四周）显示了最大差距。如预期的那样，内存（OOM）事件的发生率在第2周和第3周轻微增加。在评估结果后，我们决定净收益超过了消极面，并将中等资源回收参数部署到其他 cell。

6.隔离

50％的机器运行9个或更多的任务; 一个90％ile的机器大约有25个任务，将运行大约4500个线程[83]。虽然在应用之间共享机器增加了利用率，但还需要良好的机制来防止任务彼此干扰。这都适用于安全性和性能。

6.1 安全隔离

使用Linux chroot jail作为同一台机器上多个任务之间的主要安全隔离机制。为了允许远程调试，通过使用自动分发（和撤销）ssh密钥，使得用户只有当机器在为该用户运行任务时才能访问该机器。 对于大多数用户，这已被替换为borgssh命令，它与Borglet协作构建一个ssh连接到一个shell，该shell在与任务相同的chroot和cgroup中运行，从而更紧密地锁定访问。

VM和安全沙盒技术用于通过Google的AppEngine（GAE）[38]和Google Compute Engine（GCE）运行外部软件。在作为Borg任务运行的KVM进程[54]中运行每个托管的VM。

6.2 性能隔离

Borglet的早期版本具有相对原始的资源隔离实施：内存，磁盘空间和CPU周期的事后使用检查，结合使用过多内存或磁盘的任务，以及积极应用Linux的CPU优先级来控制使用太多CPU的任务。 但是欺骗任务对机器上其他任务的性能影响仍然太容易，因此一些用户膨胀了他们的资源请求，以减少Borg可以与他们共同调度的任务数量，这降低了利用率。 资源回收可以收回一些剩余的，但不是所有，因为涉及安全边际。在最极端的情况下，用户请求使用专用机器或cell。

现在，所有Borg任务都在基于Linux cgroup的资源容器中运行[17,58,62]，Borglet操作容器设置，提供更好的控制(因为操作系统内核在循环中)。 即使如此，偶尔的低级资源干扰（例如，存储器带宽或L3高速缓存污染）仍然发生，如在[60,83]中。

为了帮助过载和过量使用，Borg任务有一个应用类或appclass。 最重要的区别存在于延迟敏感（LS）应用类和其余的应用类（在本文中称为批处理）中。LS任务用于面向用户的应用程序和需要快速响应请求的共享基础结构服务。高优先级LS任务得到最佳处理，并且能够一次暂时使批量任务挨饿几秒钟。

二次分割存在于可压缩资源（例如，CPU周期，磁盘I / O带宽）和不可压缩资源（例如，存储器，磁盘空间）中，可压缩资源是基于速率的并且可以通过在不杀死它的情况下降低其服务质量而从任务中回收；不可压缩资源通常不能在不杀死任务的情况下被回收。 如果机器耗尽了不可压缩资源，则Borglet立即终止任务，从最低优先级到最高优先级，直到可以满足剩余保留为止。 如果机器耗尽了可压缩资源，Borglet会限制使用（有利于LS任务），使得可以处理短负载高峰而不杀死任何任务。 如果

情况没有改善，Borgmaster将从机器中删除一个或多个任务。

Borglet中的用户空间控制循环基于预测的未来使用情况（针对prod任务）或内存压力（针对非prod的情况）给容器分配内存；处理来自内核的Out-of-Memory（OOM）事件; 并且当尝试分配超出其内存限制时，或者当过度提交的机器实际上耗尽内存时，杀死任务。 Linux的渴望文件缓存由于需要准确的内存核算，显着地使实现复杂化。

为了提高性能隔离，LS任务可以保留整个物理CPU核，从而阻止其他LS任务使用。 批处理任务允许在任何核上运行，但被赋予相对于LS任务的小调度程序共享。Borglet动态调整贪婪LS任务的资源上限，以确保不会使批处理任务挨饿多分钟，在需要时选择性地应用CFS带宽控制[75]； 共享是不足的，因为我们有多个优先级。

像Leverich [56]，我们发现标准的Linux CPU调度器（CFS）需要大量调整，以支持低延迟和高利用率。 为了减少调度， CFS版本使用扩展的每个群组的负载历史[16]，允许由LS任务抢占批任务，并且当多个LS任务在一个CPU上是可运行时减少调度量。 幸运的是，许多应用程序使用线程请求模型，这减轻了持续负载不平衡的影响。谨慎使用cpusets将CPU核分配给具有特别紧迫延迟要求的应用程序。 这些努力的一些结果如图13所示。这一领域的工作继续，添加线程布局和CPU管理，即NUMA-，超线程和功率感知（例如，[81]），并提高Borglet的控制保真度。

允许任务使用达到其限制的资源。大多数被允许超过用于诸如CPU的可压缩资源，以利用未使用（松弛）资源。 只有5％的LS任务禁用此功能，可能获得更好的可预测性; 少于1％的批量任务使用此功能。默认情况下，禁止使用闲置内存，因为这增加了任务被终止的机会，但即使如此，10％的LS任务会覆盖此功能，而且79％的批处理任务这样做，因为这是MapReduce框架的默认设置。这补充了回收资源的结果（§5.5）。批处理任务会随机利用未使用的以及回收的内存：大多数时间这是可行的，虽然偶尔当一个LS任务急需资源时，批处理任务会被牺牲。

7.相关工作

已经研究了几十年的资源调度，在不同的上下文中，如广域HPC超级计算网格，工作站网络和大规模服务器集群。 在这里只关注大型服务器集群环境中最有意义的工作。

最近的一些研究分析了来自Yahoo!，Google和Facebook的集群跟踪[20,52,63,68,70,80,82]，并说明了这些现代数据中心和工作负载中固有的规模和异构性的挑战。 [69]包含集群管理器架构的分类。

Apache Mesos [45]使用基于供应的机制在中央资源管理器（有点像Borgmaster减去其调度程序）和多个“框架”（如Hadoop [41]和Spark [73]）之间划分资源管理和放置功能。 Borg主要使用基于请求的机制来集中化这些功能，这种机制可以很好地扩展。 DRF [29,35,36,66]最初是为Mesos开发的; Borg使用优先级和许可配额来代替。Mesos开发商已经宣布扩展Mesos以包括投机的资源分配和回收，并解决[69]中确定的一些问题。

YARN [76]是一个以Hadoop为中心的集群管理器。 每个应用程序都有一个管理器，它与中央资源管理器协商所需的资源; 这与Google MapReduce作业自2008年以来用于从Borg获取资源的方案大致相同.YARN的资源管理器最近才变得容错。 相关的开源工作是Hadoop容量调度器[42]，它为容量保证，分层队列，弹性共享和公平性提供多租户支持。

YARN最近已经扩展到支持多种资源类型，优先级，抢占和高级准入控制[21]。俄罗斯方块研究原型[40]支持工时感知的（makespan-aware）作业打包。

Facebook的Tupperware [64]是一个类似Borg的系统，用于在群集上调度cgroup容器; 只有一些细节已经被公开，尽管它似乎提供了一种资源回收的形式。 Twitter有开源的Aurora [5]，一个类似Borg的调度器，用于在Mesos之上运行的长时间运行的服务，配置语言和状态机类似于Borg。

Microsoft的Autopilot系统提供了“自动化软件配置和部署; 系统监控; 执行修复动作来处理软件和硬件故障“。Borg生态系统提供了类似的功能，但限于篇幅这里不做讨论； Isaard [48]概述了我们坚持的许多最佳实践。

Quincy [49]使用网络流模型为几百个节点的集群上的数据处理DAG提供公平性和数据位置感知调度。Borg使用配额和优先级在用户之间共享资源，并扩展到数万台机器。Quincy直接处理执行图，而这是单独构建在Borg的顶部。

Cosmos [44]专注于批处理，重点是确保其用户能够公平地访问他们捐赠给集群的资源。 它使用每个工作管理器（per-job manager）来获取资源; 几个细节是公开的。

微软的Apollo系统[13]使用每个作业调度器进行短期批处理作业，以在看来与Borg cell大小相同的集群上实现高吞吐量。Apollo使用机会主义执行低优先级后台工作，以多日排队延迟为代价（有时）来提高利用率。Apollo节点提供任务的开始时间的预测矩阵作为两个资源维度上大小的函数，其中调度器结合启动成本的估计及远程数据访问以进行布置决定，由随机延迟调制以减少冲突。Borg使用中央调度器基于先前分配的状态来布置决定，能处理更多的资源维度，并专注于高可用性、长期运行的应用程序的需求; Apollo可以处理更高的任务到达率。

阿里巴巴的Fuxi [84]支持数据分析工作负载；它从2009年开始运行。像Borgmaster一样，中央FuxiMaster（复制用于容错）从节点收集资源可用性信息，接受来自应用程序的请求，并将一个匹配到另一个。Fuxi增量调度策略与Borg的等价类相反：Fuxi不是将每个任务与一组合适的机器相匹配，而是将新可用资源与积压的待处理工作进行匹配。 像Mesos一样，Fuxi允许定义“虚拟资源”类型。 只有合成工作负载结果是公开的。

Omega [69]支持多个平行的，专门的“垂直”，每个大致相当于Borgmaster减去其持久存储和链接分片。 Omega调度器使用乐观并发控制来操作存储在中央持久存储器中的期望和观察到的cell状态的共享表示，其通过单独的链路组件被同步到Borglet。Omega架构旨在支持多个不同的工作负载，这些工作负载具有特定于应用程序的RPC接口，状态机和调度策略（例如，长期运行的服务器，来自各种框架的批处理作业，基础架构服务（如集群存储系统），虚拟机Google Cloud Platform）。另一方面，Borg提供了“一个适合所有”的RPC接口，状态机语义和调度器策略，随着时间的推移，由于需要支持许多不同的工作负

载，它们的规模和复杂性都有所增长，可扩展性尚未成为问题（§3.4）。

Google的开源Kubernetes系统[53]将应用程序放置在Docker容器[28]中（在多个主机节点上）。它运行在裸机（如Borg）和各种云托管提供者中，如Google Compute Engine。它是由许多建立Borg的工程师积极开发的。 Google提供了一个名为Google Container Engine的托管版本[39]。将在下一节讨论如何将Borg的教训应用于Kubernetes。

高性能计算团体在这一领域有着悠久的工作传统（例如，Maui，Moab，Platform LSF [2，47，50]）; 然而，规模、工作负载和容错的要求不同于谷歌的cell。通常，这样的系统通过具有等待工作的大积压（队列）来实现高利用率。

虚拟化提供商如VMware [77]和数据中心解决方案提供商（如HP和IBM [46]）提供集群管理解决方案，通常扩展到O（1000）机器。此外，几个研究团体具有以某些方式改进调度决策质量的原型系统（例如，[25,40,72,74]）。

最后，正如已经指出的，管理大规模集群的另一个重要部分是自动化和“运算符scaleout”。 [43]描述了如何计划故障，多租户，健康检查，准入控制和可重新启动性对于每个操作者完成大量机器是必要的。Borg的设计理念是类似的，能够为每个操作员（SRE）支持数万台机器。

8.经验教训和未来工作

本节讲述了十多年来在生产中操作Borg所学到的一些定性教训，并描述了在设计Kubernetes时如何利用这些观察结果[53]。

8.1经验教训：坏的方面

从Borg的一些特性开始，作为告诫和在Kubernetes中知情的替代设计。

作为任务的唯一分组机制，作业是限制性的。 Borg没有一流的方式来管理整个多作业服务作为一个单一的实体，或指的是服务的相关实例（例如，canary和生产轨迹）。作为一个黑客，用户在作业名称中编码服务拓扑，并构建更高级别的管理工具来解析这些名称。在范围的另一端，不可能引用作业的任意子集，这导致诸如滚动更新和作业调整大小的不灵活语义问题。

为了避免这种困难，Kubernetes拒绝了作业概念，而是使用标签组织调度单元（pods） - 用户可以附加到系统中任何对象的任意键/值对。同等的，可以通过附加到一个作业上实现Borg作业：将作业名标签附加到一组pod，但是也可以表示任何其他有用的分组，例如服务，层或发行类型（例如，生产、分段、测试）。 Kubernetes中的操作通过标签查询来识别其目标，该查询选择了将应用操作的对象。 这种方法比作业的单个固定分组提供更多的灵活性。

每个机器一个IP地址使事情复杂化。 在Borg中，机器上的所有任务都使用其主机的单个IP地址，从而共享主机的端口空间。这导致了一些困难：Borg必须调度作为资源的端口; 任务必须预先声明需要多少个端口，并且愿意在启动时被告知使用哪些端口；Borglet必须强制端口隔离；并且命名和RPC系统必须处理端口和IP地址。

由于Linux命名空间，虚拟机，IPv6和软件定义网络的出现，Kubernetes可以采取对用户更加友好的方法，消除这些复杂性：每个pod和服务都有自己的IP地址，允许开发人员选择端口，而不是要求软件适应选择 ，并消除了管理端口的基础架构复杂性。

针对高级用户进行优化，牺牲了休闲用户。Borg提供了一大套针对“超级用户”的功能，以便用户可以微调程序运行方式（BCL规范列出约230个参数）：最初的重点是支持谷歌上最大的资源消费者，他们的效率增益是至关重要的。不幸的是，这种API的丰富性使事情变得更难针对“休闲”用户，而限制了其发展。解决方案是构建在Borg之上运行的自动化工具和服务，并通过实验确定合适的设置。这些都受益于容错应用程序提供的实验自由：如果自动化出现错误，这是一个麻烦事，而不是灾难。

8.2经验教训：好的方面

另一方面，一些Borg的设计特性已经非常优越，并经受住了时间的考验。

Allocs是有用的。 Borg alloc抽象概念产生了广泛使用的日志存储模式（§2.4），另一个流行的模式是简单的数据加载器任务定期更新Web服务器使用的数据。Allocs和包允许这种帮助服务由不同的团队开发。

Kubernetes中同alloc等价的是pod，它是一个或多个容器的资源封装，这些容器总是被调度到同一机器上并且可以共享资源。Kubernetes在相同的pod中使用辅助容器代替alloc中的任务，但是想法是一样的。

集群管理不仅仅是任务管理。尽管Borg的主要作用是管理任务和机器的生命周期，但是运行在Borg上的应用程序可以从许多其他集群服务中受益，包括命名和负载平衡。 Kubernetes使用服务抽象支持命名和负载平衡：服务具有名称和由标签选择器定义的动态pod集。群集中的任何容器都可以使用服务名称连接到服务。在封面下，Kubernetes自动负载平衡与标签选择器匹配的pod中的服务连接，并且跟踪pod在由于故障而随着时间重新安排时运行的位置。

内省是至关重要的。 虽然Borg几乎总是“只是工作”，当出了问题，找到根本原因可能是挑战性的。Borg中一个重要的设计决策是要向所有用户显示调试信息，而不是隐藏：Borg有成千上万的用户，所以“自助”必须是调试的第一步。 虽然这使得更难以轻视特性和改变用户依赖的内部策略，但它仍然是一个赢家，还没有找到任何现实的替代品。为了处理大量数据，提供了多个级别的UI和调试工具，因此用户可以快速识别与其作业相关的异常事件，然后从其应用程序和基础架构本身深入查看详细的事件和错误日志。

Kubernetes旨在复制Borg的许多内省技术。 例如，它附带了诸如cAdvisor [15]等用于资源监视和基于Elasticsearch / Kibana [30]和Fluentd [32]的日志聚合的工具。可以查询主机的对象状态的快照。Kubernetes具有统一的机制，所有组件都可以用来记录事件（例如，被调度的pod，容器失败），这些事件对客户端也是可用的。

主机是分布式系统的内核。Borgmaster最初设计为一个单片系统，但随着时间的推移，它变得更像一个内核，位于服务生态

系统的核心，协作管理用户作业。例如，将调度程序和主UI（Sigma）拆分为单独的进程，并增加了服务用于准入控制、垂直和水平自动缩放，重新打包任务，定期作业提交（cron），工作流管理以及用于离线查询的归档系统操作。 总之，这使得能够在不牺牲性能或可维护性的情况下扩展工作负载和功能集。

Kubernetes架构更进一步：它的核心有一个API服务器，只负责处理请求和操作底层状态对象。集群管理逻辑被构建为小的可组合的微服务（该API服务器的客户端），例如复制控制器，用于在面对故障时保持pod的期望数量的副本，以及节点控制器，用于管理 机器生命周期。

8.3结论

在过去十年里，几乎所有的Google集群工作负载都转而使用Borg。我们继续发展它，并将从中学到的教训应用到Kubernetes。