CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data

Crush：可控，可扩展，非中心化布局的副本数据

作者：Sage A. Weil， Scott A. Brandt， Ethan L. Miller ，Carlos Maltzahn

地址：Storage Systems Research Center，University of California，Santa Cruz

邮箱：{sage, scott, elm, carlosm}@cs.ucsc.edu

译者：JIA Ru

地址：Faculty of Information and Communication Technologies，Swinburne University of Technology，Melbourne，Australia 3122

邮箱：rjia@swin.edu.au

摘要

新兴的大规模分布式存储系统正面临这这样的问题，P-TB 大小的数据文件分布在数百乃至于数千的存储设备中。这样的系统必须分布式这些数据和负载，以有效的利用资源，最优化系统的性能，与此同时，随着外围设备的增加，要管理号硬件失效。我们开发了 CRUSH，其中，设计了一个可扩展的伪随机数据分布函数主要为了分布的基于对象存储系统，该函数能够有效的映射数据对象到存储装置而不需要依赖中心目录。因为大的系统是固有动态的，当最小化不必要的数据移动的时候，设计 CRUSH 主要用来稳固添加和移除储存。该算法可以容纳各种不同类型的数据副本和可信赖的机制，并且依据用户自定义的加强跨区域失效的副本独立化策略来分布数据。

1. 简介

基于对象的存储作为一种新兴的架构，可以提升可管理性，扩展性和性能[Azagury et al. 2003]。不像传统的基于模块的硬件装置，基于对象的存储设备（OSDs）管理内部的磁盘模块分布，提供一个接口，允许其他用户读写所谓的不同大小的对象。在该系统中，每个文件数据在相对少量的这种分布在存储集群

中的对象实现剥离。为了防止在出现失效的情况下数据丢失，对象通过多装置设备（或者使用一些其他的数据冗余脚本）实现副本化。基于对象的存储系统采用小对象列表和分布式的低级别模块分布方式来替代大规模模块列表的方法来简化数据层次。尽管通过降低文件分布元数据大小和复杂性的方法可以极大地提高可扩展性，然而在数以千计的存储设备上分布数据的基本任务仍然存在，尤其是这些存储设备的能力和性能有很大的区别。

大部分的系统可以简单地写新数据到未利用充分的设备。然而这种方式所带来的最基本的问题就是这类数据通常是很少量，甚至有时候一旦写入，就会被立刻移走。当扩展存储设备时，即使是完美的分布式也将会变得不均衡，这是因为新加入的磁盘可能是空的或者只有很少量的新数据。新旧磁盘繁忙时，只能依靠系统的负载，但是只有极其稀少的情况才会均衡地使用新旧磁盘去完全的利用空闲的资源。

一种鲁棒的解决方式就是在一个系统中随机地分布所有的数据到可使用的存储设备中。这会导致概率地均衡分布以及均匀的讲新旧数据混合。当加入新的存储装置时，将已有数据的一个随机采样迁移到新的存储设备上以用来重置平衡。该方法的显著优点就是，平均上来讲，所有的设备都有相似的负载，并且在有任何潜在工作负载的情况下仍然能使系统运行的很好 [Santos et al. 2000]。进一步来讲，在一个大规模的存储系统中，单个的大文件会被随机的分布到大量可使用设备中的一个上，通常这个被选中的设备可以实现高级别的并行化以及总带宽。然而，简单的基于哈希（Hash）函数的分布不能处理设备数量的改变，这将会导致出现大量的要重置数据。而且，通过在许多其他的设备上分布每个磁盘的副本来实现副本分离的已有的随机分布模式面临着这样的问题：当设备同时失效时，数据丢失的可能性是非常大的。

我们已经开发了 CRUSH（Controlled Replication Under Scalable Hashing 广泛分布条件下的可控副本），一种伪随机数据分布算法，可以在异构的结构的存储集群高效鲁棒地分布地分布对象副本。CRUSH 作为一种伪随机的确定性的函数将输入值，对象或者对象组标示符映射到存储对象副本的设备列表上。与传统方法的区别在于 CRUSH 的数据布局并不依赖于每个文件或者每个对象目录的排序，CRUSH 只需要一个“合同”，可以层级的描述那些有设备构成的存储集群以及了解副本布局策略。该方法有两个优点：首先，它可以完全地分布在一个大规模系统的任意一方并且可以独立的计算出任意对象的位置；其次，所需要的极少的元数据通常是静态的，只有在设备添加或者移除的时候元数据才会改变。

CRUSH 可以优化数据分布使得可用资源能够更好的被使用；当存储设备添加或者移除时，可以高效地重组数据；关于对象副本布局的系统参数设定更加灵活，这使得全部或者相关的硬件失效时，能够最大化数据的安全性。CRUSH 提供多种多样的数据安全机制，包括多重副本（镜像法），RAID 校验模式或者其他模式的编码删除，以及混合方法（例如 RAID-10）。这些特点使得 CRUSH 非常适用于超大规模（多重 Pb 级）存储系统的对象分布管理，因为在这样规模的系统中，可扩展性，性能，可靠性都是非常重要的。

2.相关的工作

基于对象的存储作为一种可以提高存储系统可扩展性的机制最近引起了人们很大的兴趣。大量的关于文件系统的研究以及文件系统的生成都采用了基于对象的方法，包括开创的NASD文件系统 [Gobioff et al. 1997]，Panasas文件系统 [Nagle et al. 2004], Lustre [Braam 2004], 以及其他 [Rodeh and Teperman 2003; Ghemawat et al. 2003]。其他的基于块的分布式文件系统像GPFS [Schmuck and Haskin 2002]和联合数组块（Federated Array of Bricks (FAB)）[Saito et al. 2004]正面连着相似的数据分布挑战。在这些系统中，半随机或者启发式的方法用来把新数据分配到有可用存储空间（能力）的存储设备上，但是为了保持时时的分布平衡，数据极其少地会被重新分配。更重要的是，所有的这些系统都是通过对元数据目录进行某种排序来实现分配数据的，而不同的是CRUSH是依赖集群描述“合同”和确定性的映射函数来实现分配数据的。最显著的区别主要体现在写数据上，因为系统通过利用CRUSH可以计算新数据的存储目标而不用访问/咨询中心分配器。Sorrento [Tang et al. 2004]存储系统利用散列法（哈希函数hashing）[Karger et al. 1997]这一点，使得其与CRUSH十分的翔翔，但是该系统并不支持可控权重的设备，良好的数据均衡分布以及失效区域的数据安全性的提升。

尽管数据迁移问题已经被广泛地研究学习了，尤其在需要精确分配映射的系统方面[Anderson et al. 2001; Anderson et al. 2002] ，但是这些方法仍需要很大的元数据需求，而这些需求采用函数的方法例如CRUSH则是可以避免的。Choy, et al. [1996] 描述了一些用于磁盘间分配数据的算法，当磁盘增加时这些算法可以移动适量的对象，但是这些算法并不支持加权，副本或者磁盘移除。Brinkmann, et al.[2000] 利用哈希函数（hash）实现异构而非静态集群的数据分布。SCADDAR [Goel et al. 2002]解决了添加和移除存储设备的问题，但是却只支持受限的副本策略子集。这些方法里，没有一个有CRUSH那样的灵活性，以及在故障区域里提高可靠性的能力。

CRUSH与RUSH [Honicky and Miller 2004]族的算法最为接近，这是因为CRUSH正是基于该类算法设计的。RUSH保留了在参考文献中仅存的一系列利用映射函数分布精准元数据以及支持高效添加和移除加权设备的算法。尽管具有这些基本的属性，但是一系列的问题使得RUSH在实际中仍不是一个良好的解决方法。CRUSH全面地泛化归纳了 $RUSH_p$ 和 $RUSH_T$ 中有用的元素，重新解决了之前没有解决的关于可靠性和副本问题，并且提高了性能和灵活性。

3. CRUSH 算法

CRUSH 算法根据每个设备的权重值，近似出的一个均匀概率分布（曲线），将数据对象分布在存储装置中。该分布由层级集群映射控制，其中该集群映射表示有可用的存储资源以及构成其的逻辑器件组合。例如，可用服务器机柜的行数来描述大规模装置，其中“机柜”由磁盘架构成，而“架”则是由存储设备构成的。数据分布策略是由一系列的布局规则定义的，而这些规则设定了从集群中选择副

本目标的数量以及制定用于副本布局的限制条件。例如，可以制定 3 个镜像副本放置在不同物理机柜的设备上，因此这些副本就不可以共享相同的电路。

假设给出一个单一的整数输入值 ，CRUSH 将输出一个顺序列表$\vec{R}$，其中包括n个不同的存储目标。 CRUSH 主要是利用一个强多输入整数哈希函数，该函数的输入包括x，通过用一个集群映射（map），布局规则和x，就可以实现完全确定性的以及独立可计算的映射匹配。该分布是伪随机的，这主要体现为根据相似的输入而得出的输出结果之间没有明显的协相关性，以及在所有存储设备上的存储项之间也没有明显的协相关性。我们可以这样认为 CRUSH 生成了一个分簇的副本分布，其中共享一个项分布的一系列设备独立于所有其他的项。

3.1 层级集群映射

集群映射是由一些具有数字识别符和与之相关的权重值的设备（devices）和存储桶（buckets）组成的。存储桶可以包含任意数量的设备或者其他的存储桶，允许它们在同一存储级内形成内部节点，而在这样的存储级中，设备通常都位于叶节点上。管理员给存储设备跟配权重来控制它们所负责存储的数据的相关数据量。尽管大规模的系统很可能包含具有各种不同处理能力和性能的设备，统计地使用工作负载的相关设备随机化的数据分布，平均来讲，这样的一个设备负载是与存储的数据量是成比例的。这就导致了一个一维布局度量和权重可以用设备的处理能力来推导。存储桶的权重是由它们包含的所有项的权重总和来定义的。

存储桶可以通过任意地组合来构造一个层级代表可用存储。例如，可以在最底层创建一个有“架”存储桶的集群映射来代表一系列具有标识的设备，这些设备被安装后，可将“架”合并到“机柜”存储桶以便将安装在同一个框架上的架子组合在一起。在大规模系统中机柜可以进一步被组合到存储桶里的“行”或“空间”。通过采用伪随机哈希类的函数可以递归地选取嵌套的存储桶项使得数据可以放置到层级中。与传统的散列法（hashing）技术相比，目标存储箱（设备）里的任何改变都能导致存储箱内容的大量重组，CRUSH 是基于四种不同类型存储桶而设计的，每个都有一个不同的选择算法以用来解决由于增加或移除设备导致的数据移动以及整体计算复杂度的问题。

3.2 副本布局

CRUSH 用来在有权重的设备之间均匀的分布数据来维持存储利用率和设备带宽资源的统计地平衡。在层级间中的存储设备上的副本布局对数据的安全也有很大的影响。通过对底层物理装置结构的反应，CRUSH 可以建模，并且处理潜在的可能产生相关的的设备失效的问题。比较典型的问题包括物理上的接近性，共享的电力资源以及共享的网络。通过把信息编码到集群映射上，CRUSH 的布局策略可以分离不同失效区域的对象副本同时使分布仍像预期的那样维持不变。例如，为了解决并发失效的概率性这个问题，需要把数据副本安置在不同的架（shelve），

框架（rack）电力供给，控制器以及物理位置的设备上。

为了更好的解释 CRUSH 应用的多种不同的情景，根据副本布局策略以及底层硬件配置，CRUSH 给每个应用在存储系统或者管理（员）程序上的用来精确的是设定对象副本布局的副本策略或分布策略制定了一系列的布局规则。例如，可以设定一个选择成对目标的规则来实现双边镜像，选择两个不同数据中心里的三个目标以实现三边镜像，跨六个存储设备的 RAID-4 规则，等等[1]。

每个规则（rule）都包含一系列可以应用于简单执行环境的层级结构的操作指令，用算法 1 的伪代码表示。CRUSH 函数的证书输入 x，是典型的对象名或者标示符，例如可以作为一组副本部署在相同设备上的对象的标示符。“take”操作在同一存储层级选取一个项（多为 bucket 存储桶）并把它分配给负责把输入传递到后续操作的向量 $\vec{i}$。$select(n,t)$ 操作迭代每一个元素 $i \in \vec{i}$，并选择 n 个属于类型 t 的完全不同的项，这些项属于同一根节点的子树。存储设备有一个已知的固定类型，并且系统中的每个存储桶有一个类型域，该类型域主要用于区分存储桶类（例如，有些代表“行”，有些代表“机柜”）。对于每个 $i \in \vec{i}$，$select(n,t)$ 调用需要的 $r \in 1,...,n$ 项，并且通过任意的中间存储桶递归地降序排列，采用函数 $c(r,x)$（为 3.4 节提到的每种类型的存储桶所定义的）伪随机地选择每个存储桶里的嵌套项，直到找到需要的属于类型 t 的项。结果 $n|\vec{i}|$ 个完全不同的项重新被放回输入 $\vec{i}$ 中；形成接续的 $select(n,t)$ 的输入或者用“emit”操作指令转移到结果向量中。

举例而言，表 1 中定义的规则从层级的根开始，如图 1 所示，用第一个 $select(1,row)$ 选取了一个“行（row）”（选取了 $row2$ ）类型的单一存储桶。接着后续 $select(3,cabinet)$ 选择了 3 个完全不同的嵌套在之前选择的 $row2$（$cab21,cab23,cab24$）之下的机柜，同时最后的 $select(1,disk)$ 迭代输入向量里的这三个机柜存储桶并且分别选择一个磁盘嵌套在每一个机柜存储桶下方。最终结果就是 3 个磁盘分布在 3 个机柜上，并且所有的都位于同一行。该方法允许同时分离副本并且根据容器的类型（如行（row），机柜（cabinet），架（shelve））限制副本，并且允许有一个很有用的同时考虑可靠性和性能两者的属性。规则里可以包含多路“take”和“emit”块，这些“emit”块允许允许存储目标精确地

---

[1]：尽管有太多不同的数据冗余机制，但是为了简化，我们都讲数据对象存储为副本（replica）而没有任何通用性的损失。

从不同的存储池中抽拉出来，也可以保存在远程副本场景里（远程副本场景：是指其中一个副本存储在远程地点）或分层装置里（如快速，近线性存储和慢速，高处理能力的数组）。

算法 1：对象 x 的 CRUSH 分布

Algorithm 1 CRUSH placement for object x

1: **procedure** TAKE(a) ▷ Put item a in working vector $\vec{i}$
2: $\vec{i} \leftarrow [a]$
3: **end procedure**

4: **procedure** SELECT(n,t) ▷ Select n items of type t
5: $\vec{o} \leftarrow \emptyset$ ▷ Our output, initially empty
6: **for** $i \in \vec{i}$ **do** ▷ Loop over input $\vec{i}$
7: $f \leftarrow 0$ ▷ No failures yet
8: **for** $r \leftarrow 1, n$ **do** ▷ Loop over n replicas
9: $f_r \leftarrow 0$ ▷ No failures on this replica
10: $retry\_descent \leftarrow false$
11: **repeat**
12: $b \leftarrow bucket(i)$ ▷ Start descent at bucket i
13: $retry\_bucket \leftarrow false$
14: **repeat**
15: **if** "first n" **then** ▷ See Section 3.2.2
16: $r' \leftarrow r + f$
17: **else**
18: $r' \leftarrow r + f_r n$
19: **end if**
20: $o \leftarrow b.c(r', x)$ ▷ See Section 3.4
21: **if** $type(o) = t$ **then**
22: **if** $o \in \vec{o}$ or $failed(o)$ or $overload(o,x)$ **then**
23: $f_r \leftarrow f_r + 1, f \leftarrow f + 1$
24: **if** $o \in \vec{o}$ and $f_r < 3$ **then**
25: $retry\_bucket \leftarrow true$ ▷ Retry collisions locally (see Section 3.2.1)
26: **else**
27: $retry\_descent \leftarrow true$ ▷ Otherwise retry descent from i
28: **end if**
29: **end if**
30: **else** ▷ Not type t
31: $b \leftarrow bucket(o)$ ▷ Continue descent
32: $retry\_bucket \leftarrow true$
33: **end if**
34: **until** $\neg retry\_bucket$
35: **until** $\neg retry\_descent$
36: $\vec{o} \leftarrow [\vec{o}, o]$ ▷ Add o to output $\vec{o}$
37: **end for**
38: **end for**
39: $\vec{i} \leftarrow \vec{o}$ ▷ Copy output back into $\vec{i}$
40: **end procedure**

41: **procedure** EMIT ▷ Append working vector $\vec{i}$ to result
42: $\vec{R} \leftarrow [\vec{R}, \vec{i}]$
43: **end procedure**

3.2.1 冲突，失效，过载

$select(n,t)$操作可以贯穿存储层级的不同级，目的是定位n个嵌套在起始点下方的指定类型t的不全不同的项，并且用$r=1,...,n$局部参数化一个递归过程，同时确定所需的副本数量。在该过程中，CRUSH 可以用一个修订的输入r'因为以下的三个原因来拒绝和重选项：如果一个项目已经在当前的集合里被选中了（冲突-$select(n,t)$的结果一定是完全不同的），如果一个设备失效，或者一个设备过载。在集群映射 map 中这样标记失效或者过载的设备，但是层级中余下的就可以避免不必要的数据转换。CRUSH 通过伪随机概率（在该集群映射中，这个概率是设定好的）拒绝的方式选择性地转移过载设备的部分数据-通常拒绝所报告的过度使用要求。对失效或者过载的设备而言，CRUSH 通过采取在$select(n,t)$起始点重启递归的方法在整个存储集群中均匀地重新分布项（参见算法 1 第 11 行）。在冲突实例中，替代输入r'最先用于递归的内部级去尝试实现局部搜索（参见算法 1 第 14 行）并且避免某种偏斜的发生，该偏斜是指整体的数据分布远离冲突发生概率更高的子树（例如，在这样的子树种存储桶小于n）。

Action	Resulting $\vec{i}$
take(root)	root
select(1,row)	row2
select(3,cabinet)	cab21 cab23 cab24
select(1,disk)	disk2107 disk2313 disk2437
emit	

Table 1: A simple rule that distributes three replicas across three cabinets in the same row.

表 1：在同一行中在三个机柜上分布三个副本的简单规则

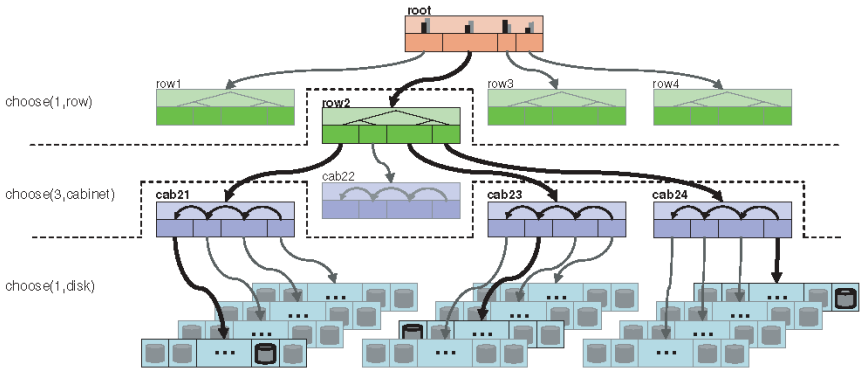

Figure 1: A partial view of a four-level cluster map hierarchy consisting of rows, cabinets, and shelves of disks. Bold lines illustrate items selected by each *select* operation in the placement rule and fictitious mapping described by Table 1.

图 1： 包含行（row），机柜（cabinet），磁盘架（shelve）的四级集群映射层级的局部示意图。加粗的线表示根据表 1 中所描述的布局规则中的每个 *select* 操作和虚拟的映射所选择的项

3.2.1 副本等级

奇偶校验和删除编码模式与副本相比，布局布局需求还是稍有不同的。在主拷贝副本模式中，在前一个副本目标失效（该副本目标已经有了一个数据拷贝）后，很有可能变成一个新的主拷贝。在这种情况下，CRUSH 通过用 $r' = r + f$ 去重选，然后得到可以利用的“最先的 n（first n）”个合适目标，其中 f 是用当前的 $select(n,t)$ 去布局尝试失败的个数（参见算法 1 第 16 行）。然而在奇偶校验和删除编码模式中，存储设备在 CRUSH 输出中的等级或者位置是非常重要的，这是因为每个目标存储了不同位的数据对象。特别是，如果一个存储设备失效，那么它会在 CRUSH 输出列表 $\vec{R}$ 中相应的位置上被替换掉，这样列表中其他设备的位置排名就会保持不变（如 $\vec{R}$ 中的位置，见图 2）。在这种情况下，CRUSH 用 $r' = r + f_r n$ 进行重选，其中 f_r 表示在 r 上尝试失败的个数，因此为每个副本等级排名定义一个候选序列，这些候选项很有可能独立于其他的失败。与此形成鲜明对比的是，RUSH 并没有处理失效设备的特殊手段；像其他的散列法（hashing）分布函数一样，RUSH 采用“first n”方法来跳过结果中的失效设备，所以它非常不适用于奇偶校验模式。

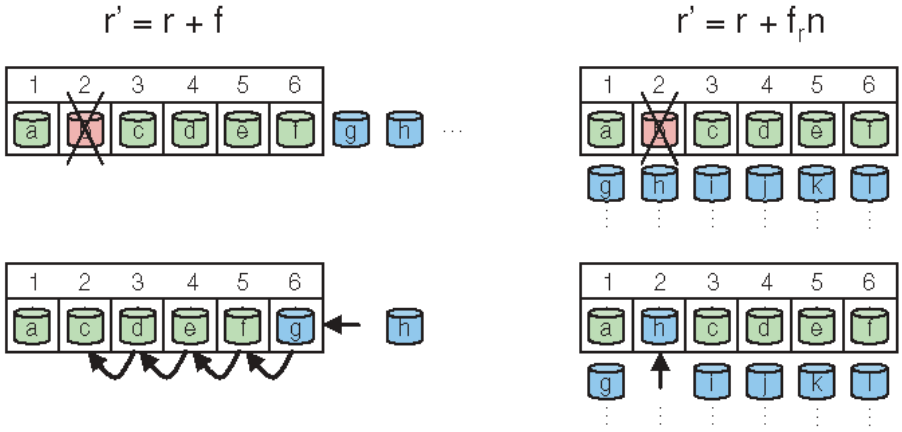

Figure 2: Reselection behavior of *select(6,disk)* when device $r = 2$ (b) is rejected, where the boxes contain the CRUSH output $\vec{R}$ of $n = 6$ devices numbered by rank. The left shows the "first n" approach in which device ranks of existing devices (c,d,e,f) may shift. On the right, each rank has a probabilistically independent sequence of potential targets; here $f_r = 1$, and $r' = r + f_r n = 8$ (device h).

图 2：当设备 $r = 2(b)$ 被拒绝时，$select(6,disk)$ 的重选行为，这些盒子包含 CRUSH 的输出 $\vec{R}$，在该输出中，有 $n = 6$ 个设备且其等级排名以数字化的形式表示。图 2 左侧表示“first n”方法中的对已有设备 (c,d,e,f) 的设备等级排名可能会发生转移。在图 2 右侧，每个排名都有一个概率独立的潜在的目标序列；其中 $f_r = 1$，和 $r' = r + f_r n = 8$（设备 h）。

3.3 映射改变和数据移动

在大文件系统中，数据分布的一个很重要的元素就是对存储资源增删的响应。CRUSH 对数据和负载采用均匀分布，用以避免负载不均匀以及不能充分利用相关的资源。当一个独立的设备失效时，CRUSH 会对该设备进行标示，但把该设备保留在层级结构中，并且，如果该设备拒绝，则该设备里的内容会用替代算法（参见章节 3.2.1）均匀地重新分布。这样的集群映射改变会对一个优化的（最小）分式 w_{failed} / W （其中 W 是所有设备的权重）产生影响，这是因为当仅有失效设备的数据被移走时，所有的数据都要重新映射到存储目标中。

随着存储资源的增加和删除，集群层级结构被修改时，这种情况更加复杂。CRUSH 的映射过程，主要是把集群映射当做一个具有不同权重层级的决策树，因此可能导致增添的数据移动会超过理论的优化值 $\frac{\Delta w}{W}$。在每一层层级结构中，在相关的子树中的任何一个转变都会改变整个分布，某些数据对象必须要从权重递减的子树转移到权重递增的子树上。因为在层级中每个节点的伪随机分配决策都是统计独立的，数据迁移到子树中，都是均匀的分布在每个点下方，不必要重新映射到每个叶项而最终对权重的改变负责。只有在随后的（更深的）级别的分配过程需要（不同的）数据转换来维持整体相关分布的正确性。这种通用的效果在图 3 的二进制层级实例中给出了详细的表述。

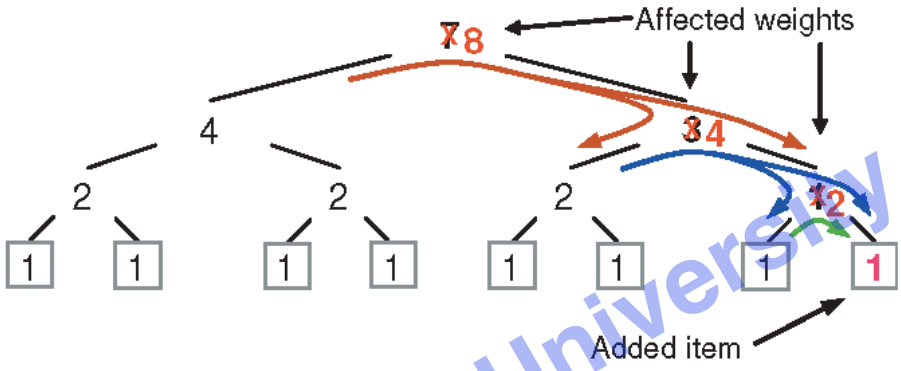

Figure 3: Data movement in a binary hierarchy due to a node addition and the subsequent weight changes.

图 3： 由于节点增加和随之的权重改变造成的二进制层级结构中的数据移动

一个层级结构中的数据移动量都由一个低边带 $\frac{\Delta w}{W}$，数据的小数部分表示最新添加权重为 Δw。随着层级高度 h 的变化数据移动在不断的增加，但是会有一个渐进的上边界 $h\frac{\Delta w}{W}$。当 Δw 相对于 W 小时，数据移动量接近上边界，因为在每一步递归中数据对象移动到子树都会有极低的概率可能会映射到相对低权重的项里。

3.4 桶（Bucket）类型

大体说来，设计 CRUSH 主要用来平衡两点：映射算法的有限性和可扩展性，以及当增加或移除设备而改变整个集群时重置均衡分布的最小的数据迁移。就后者而言，CRUSH 定义了四种不同类型的桶来表示集群层级结构的内部（非叶）节点：均匀桶（uniform buckets），列表桶（list buckets），树桶（tree buckets），草帽桶（straw buckets）。每种桶的类型都是基于不同的内部数据结构设定的并

且利用不同的$c(r,x)$函数在副本分配过程中来伪随机地选择嵌套的项，并且表示在计算能力和重组效率之间的不同权衡（tradeoff）。均匀桶的局限性在于它们包含的项重量必须是相同的（这更像是传统的基于哈希散列的分布函数），然而其他类型的桶必须包含重量不同的混合项。表 2 总结了这些差异。

Action	Uniform	List	Tree	Straw
Speed	O(1)	O(n)	O(log n)	O(n)
Additions	poor	optimal	good	optimal
Removals	poor	poor	good	optimal

Table 2: Summary of mapping speed and data reorganization efficiency of different bucket types when items are added to or removed from a bucket.

表 2：当向桶中添加项或者移除项时，不同类型的桶的映射速度以及数据重组效率的总结

3.4.1 均匀桶（Uniform Buckets）

在一个大规模的系统里，设备很少会被独立的增加。取而代之的是，新的存储装置多是部署到块里，这些块是由一系列有标识的设备构成的，通常都是作为一个额外的架（shelf）添加到服务器或许是一整个机柜（cabinet）里。这些设备有着相似的使用寿命可作为一个整体不再使用（个别设备失效情况除外），所以很自然的可以把这些设备看做一个单元。CRUSH 均匀桶通常用来表示在这种情况下的一系列设备的集合，而这些集合通常都是带有身份标识的。这样做在性能方面的优点是：CRUSH 可以在连续的时间里映射副本到均匀桶。一旦均匀化限制不适合这种情形，其他的类型的桶仍可以使用。

假设 CRUSH 的输入值为x 以及副本数量为r,我们用$c(r,x)=(hash(x)+rp)$函数模（mod）m的方法，从大小为m的均匀桶中选择其中一项，其中 p是随机地（但是可控制地）选择比m大的主要数。对于任意$r \le m$，我们通常都是采用一些简单数值理论引理来选择一个区别比较明显的项[2]。对于$r > m$，就不能有这种保证了，这也就意味着有着相同输入x的两个不同副本r可以分解相同的项。实际上，这也不过就是非零概率冲突以及随之相应的采用分配算法的回溯（参见章节 3.2.1）。

---

[2]: The Prime Number Theorem for Arithmetic Progressions [Granville 1993]更进一步地展示该函数能够在$m\phi(m)$不同的安排情境下分布对象x的副本，并且每种安排都基本相同。其中$\phi(\bullet)$是 Euler Totient 函数。

如果均匀桶的大小改变了，设备间的数据会完全的重组，这更像是传统的基于哈希的分布策略。

3.4.2 列表桶（List Buckets）

列表桶结构的内容更像是一个链接列表，并且可以拥有任意权重的项。为了安置一个副本，CRUSH 通常从最常常添加项的列表头开始，然后把该项的权重与其他余下所有项的权重和进行比较。依靠 $hash(x,r,item)$ 的值，当前项在某些恰当的可能性下可能被选取，或者是递归地执行到列表尾。这种方法是从 $RUSH_p$ 中推倒出来的，改正了分配问题中的“最常常添加的项或者是更陈旧的项”。 对于一个扩展的集群来说，这个一个中肯且直观的选择：不管是在某种概率条件下把一个对象重新分配给最新的设备上还是像以前一样仍然保留还在比较陈旧的设备上。结果是当对桶添加项时，数据迁移是可最优的。然而从列表中间或者尾部移除的项往往导致大量不必要的移动，使得列表桶最适合于桶从不紧缩或者很少紧缩的情况下。

$RUSH_p$ 算法与包含单个列表桶以及多个均匀桶的两层 CRUSH 层级近似相等。它固定的集群表述阻碍了分配原则的使用或者用来加强可靠性的 CRUSH 控制数据分配失效域。

3.4.3 树桶（Tree Buckets）

像其他链接列表数据结构一样，列表桶对小规模项集合是非常有效的但是并不适用与大规模项集合，因为他们的运行时间 $O(n)$ 太大了。树桶作为一种从 $RUSH_T$ 推倒而来的方法，可以解决这个问题。树桶在二进制树种存储项，把分配时间降低到 $O(\log n)$，并且使得它们更适合管理大规模的设备结合或者嵌套的桶。

$RUSH_T$ 算法与包含单个列表桶以及多个均匀桶的两层 CRUSH 层级等效。

树桶可以被结构化成具有重量的二进制搜索树，并且项都分布在叶上。每一个内部节点都知道她左右两侧子树的权重和并且根据固有的策略（描述如下）加以标记。为了在桶里选取一项，CRUSH 从树的根节点开始执行并且计算输入 x，副本数量 r，桶标示符以及当前树节点（初始化为根节点）标签的哈希值。计算结果与左右两侧子树的权重率相比，来决定下一步访问哪个子节点。重复执行该过程直到到达叶节点，且在该叶节点选取桶里相应的项。只需把 $\log n$ 哈希值以及

节点比较加载在该项上。

桶的二进制树节点采用一种简单的固有策略来二进制值地标记，这种设计可以避免随着树的生长或者紧缩标记而有所改变。通常树中最左边的页标记为“1”。每次当树扩展时，原来的根节点则变成新的根节点的左侧子节点，而把之前的旧根节点的标签向左偏移一位（1，10，100，等等）得到的值作为新的根节点的标签。除了把“1”预留到每个值以外，右侧树的标签与左侧的树标签是镜像对称的。图 4 就描述了一个有 6 个叶子的带有标签的二进制树。这种策略特点在于当新的项从桶中添加（或者移除）以及树增长（或者紧缩）时，对于通过已经存在的叶项二进制树的路径只在根处添加（或者移除的）的额外节点处发生改变，在分配决策树的起始阶段。一旦一个对象被分配到某个特殊的子树中，它的最终映射只依赖于所在子树的权重和节点标签，并且只要子树中的项保持不变，那么映射就不会发生改变。尽管层级式的决策树可能在嵌套的项之间引入了一些额外的数据迁移，但是该策略只在合理的层中移动，甚至可以为一些非常大的桶提供很有效的映射。

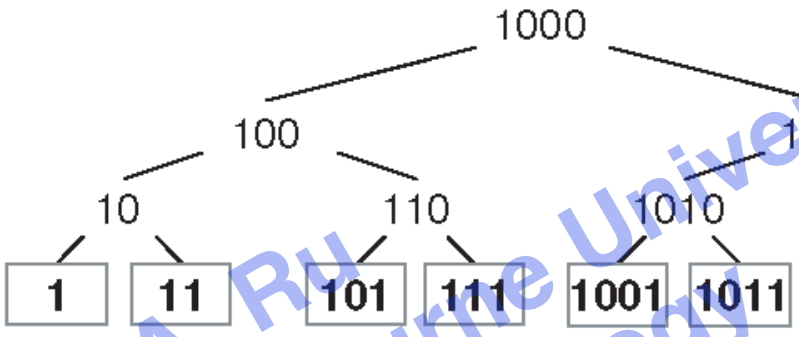

Figure 4: Node labeling strategy used for the binary tree comprising each tree bucket.

图 4：二进制树的节点标记策略以及每个树桶的比较

3.4.4 草帽桶（Straw Buckets）

列表桶和树桶都是结构化的，因此为了选取一个桶项需要计算有限的哈希值并且还要和权重进行对比。这样做，可以划分和避免关于某些特定项的优先权（例如那些位于列表顶端的点）或者根本不必考虑子树的所有项。这可以提高副本分配过程的性能，但是当由于增删项或者重置项的重量造成桶内容改变时，也引入了不恰当的重组行为。

草帽类型桶允许通过用一个类似编草帽的过程使所有的项公平与其他项竞争来实现副本分配。为了实现副本分配，草帽的随机长度是根据桶中每个项的长度推倒出来的。草帽长度最长的项最终实现副本分配。基于 CRUSH 输入x，副本数量r，桶项目i的哈希值，每个草帽的初始长度是在固定的数值范围内的。每个

草帽的长度都可以根据基于项权重[3]因子 $f(w_i)$ 来增加，因此权重更大的项更容易赢得分配权，举例： $c(r,x)=\max_i(f(w_i)hash(x,r,i))$ 。尽管该过程比列表桶过程慢两倍甚至比树桶（对数扫描）更慢 ，但是当项目被修改时草帽桶可以得到不同嵌套的项之间最优的数据移动。

根据期望集群的增长模式来选择不同类型的桶来指导实现高效数据移动的交易映射函数的计算。当桶的类型需要固定时（例如一机架带有标示符的硬盘），均匀桶最快。如果仅仅是要扩展一个桶，当把新项添加到表头时，列表桶可以提供最优的数据移动。这允许 CRUSH 可以非常精确的转移数据到新设备上，而不需要移动桶间的其他项。当旧项被移走或者重新设定总量时，下降趋势是 $O(n)$ 映射速度以及额外的数据移动。当需要去除项或者比较注重重组效率时，草帽桶可以在子树间提供最优的迁移行为，同时提供卓越的性能以及相当好的重组效率。

4. 评估

CRUSH 的设计初衷是为了实现各种不同的目标，包括异构存储设备间的平衡的权重分布，由于增删存储（包括单独的硬盘失效）引发的最小数据移动，通过分离跨失效区域的副本来提高系统可靠性，灵活的集群描述以及描述存储和分布数据的规则系统。我们评估与 $RUSH_p$ 和 $RUSH_T$ 集群相关的 CRUSH 配置下的各种不同的行为，这些评估都是通过模拟仿真设备上的对象分配以及检测这种分配结果来实现的。 $RUSH_p$ 和 $RUSH_T$ 通常是由两级 CRUSH 层级结构衍生出来的，通常只有一个列表桶或者树桶（分别地），但是包含多个均匀桶。尽管 RUSH 固有的集群表述不能充分利用分配原则也不能利用跨失效区域的副本分离，我们考虑它的性能和数据迁移行为。

4.1 数据分布

CRUSH 的数据分数通常是随机的与对象标示符 x 和存储目标不相关，所以这就导致了相同重量的设备间的分布平衡。我们根据以往经验测量包含在不同类型桶里设备上的对象分布，并且比较设备利用方差与二项式概率分布，我们所期待的理论行为可以从均匀随机分布中推倒出来。当我们需要分布 n 个对象时，是以概

---

[3]：尽管 $f(w_i)$ 作为简单的封闭形式不为人所知，但是它可以按照步骤相对直接地计算权重因子（源代码已知）。只有在每次桶被修改是，才需要这种类型的计算。

率 $p_i = \frac{w_i}{W}$ 为已知设备 i 分配每一个对象，期望的设备利用率可以用二项式 $b(n, p)$

来预测，其中 $\mu = np$，标准差 $\sigma = \sqrt{np(1-p)}$。在有很多设备的大规模系统中，

我们可以近似地认为 $1-p \simeq 1$，因此标准差 $\sigma \simeq \sqrt{\mu}$，这也就是说当数据对象数量非常大[4]时，利用率也多是均衡的。正如所期望的那样，我们发现 CRUSH 分布一直与二项式的均值与方差相匹配，不管是在同构的集群还是在混有不同设备权重的集群里。

4.1.1 过载保护

尽管 CRUSH 可以很好地平衡（设备利用率的方差很小）大量的对象，因为在任意的随机过程中任何设备上的分配转换为非零概率都比平均值大很多。不像已有的概率映射函数（包括 RUSH），CRUSH 包括一个设备过载纠正机制，该机制可以重新分布设备上的数据的任意一部分。这可以用来度量当它过度装载处于危险中时（可选择地“leveling off”过度装载的设备）设备分配与它的过度利用之间的比例。当数据分布超过一个拥有 1000 台设备的集群容量的 99%时，我们发现 CRUSH 映射执行次数增长不到 20%，尽管在 47%的设备上进行过载调整，方差减少到 4 倍（如期望的那样）。

4.1.2 差异和部分失效

先前的研究[Santos et al. 2000]表明随机的数据分布提供的实际的系统性能与数据带状话相似（仅仅慢一点而已）。我们自己的 CRUSH 性能测试只能作为一个基于对象的分布式存储系统的一部分[Weil et al. 2006]，我们发现随机化对象分布会产生大约 5%的写性能损失，这主要是由于 OSD 工作负载的差异，以及相关层次的 OSD 利用率的变化。然而实际中，当精确的带策略起效时，这种差异主要与同构的工作负载（通常是写）有关。更常见的是，工作负载多是混合的并且当他们传入硬盘时随机出现（至少与硬盘的布局无关），这就导致了在设备工作负载和性能中有着相似的差异（尽管已经很小心布局结构）并且有着类似的下降的聚集通量。我们发现面对潜在的工作负载时 CRUSH 缺少元数据以及鲁棒分布，这种潜在的工作负载远比在小规模工作负载中带来微少的性能损失重要的多。

该分析假设设备的容量基本都是静态的。以往真实系统的经验告诉我们分布式存储系统的性能往往会被少量的慢速的，过载的，碎片的以及其他的性能不佳的设备所牵连。传统的，精确的分配模式可以手动地避免这样的问题设备，而类哈希式的分布函数则不能。CRUSH 允许采用已经存在的过载修正机制，把退化的设

---

[4]：当对象较多时（如：n 非常大）二项式分布近似于高斯分布。

备当做“部分失效”来处理，通过转移适量的数据和工作负载可以避免这样的性能瓶颈以及修正实时的工作负载不均衡。

由存储系统所导致的细粒负载均衡可以通过把读工作负载分布到数据副本的方法进一步地减轻设备负载差异，DSPTF 算法做了详尽的说明[Lumb et al. 2004]；这样的方法尽管相互互补，但是仍落后于 CRUSH 映射函数和本文的研究范围。

4.2 重组和数据移动

当在一个拥有 7290 个设备的集群中使用 CRUSH 和 RUSH，我们估计由于增删存储引起的数据移动。CRUSH 集群有四个层次：一共 7290 个设备，其中 10 个存储设备，每个存储设备有 9 个架（shelve），每个架有 9 个机柜（cabinet），每个机柜有 9 行（row）。$RUSH_T$ 和 $RUSH_P$ 与包含单一树桶或者单一列表桶（分别地）以及 10 个设备（每个设备包含 729 个均匀桶）的两层 CRUSH 映射等同。结果与理论的最优移动量 $m_{optimal} = \frac{\Delta w}{W}$ 相比，其中 Δw 是存储设备添加或删除的混合权重，W 是整个系统的权重。举例来说，双倍扩容系统的容量在最优重组的情况下需要有恰好一半的数据移动到新的设备。

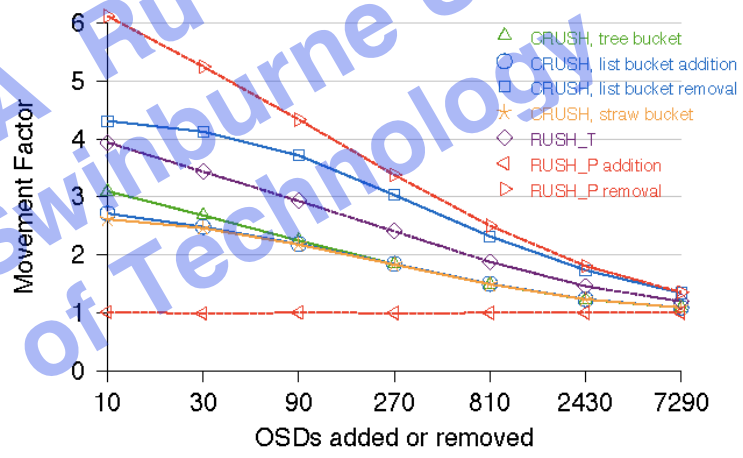

Figure 5: Efficiency of reorganization after adding or removing storage devices two levels deep into a four level, 7290 device CRUSH cluster hierarchy, versus $RUSH_P$ and $RUSH_T$. 1 is optimal.

图 5：在增删存储设备后两层结构深入到第四层的高效重组，拥有 7290 个设备的 CRUSH 集群层级，与 $RUSH_T$ 和 $RUSH_P$ 的对比，其中 1 是最优的。

图 5 表明了与移动因子 $m_{actual} / m_{optimal}$ 相关的重组效率，其中 1 表示最优的移动的

对象个数以及大数值表示增加的移动。X 轴表示增加或删除的 OSD 的数量，Y 轴表示移动因子以 log 比例的出图。在所有的情况下，较大的权重改变（与整个系统相比）会导致更有效的重组。 $RUSH_P$（单一的大规模的列表桶）控制断点，增加项产生最少的（最优的）移动，而移除项产生最多的移动（会有一个很大的性能损失，参见下面章节 4.3）。列表桶（仅为增加）或者草帽桶的 CRUSH 多层次的层级具有第二少的移动量。具有树桶的 CRUSH 相对低效一些，但是仍比简单的 $RUSH_T$ 好大概 25%（由于每个树桶中有轻微不平衡的 9 项二进制树）。从具有列表桶的 CURSH 层级移除性能很差，正如预期的一样（参见章节 3.3）。

图 6 描述了当嵌套的项增加或删除时不同类型桶的重组效率。 $\log n$ （二进制树的深度）设定修正的树桶的移动因子的边界。向草帽或者列表桶添加项近似是最优的。均匀桶的修正会导致整体数据的重组。修正列尾（从最旧的存储移除）会类似地引入与桶大小成比例的数据移动。不管某种限制，当移除操作很少且在某种程度上性能影响是最小的时，列表桶可以在整个存储层级中被合理地分配。在最常见的重组情景下，结合了均匀，列表，树和草帽桶的混合方法可以最小化数据移动同时又可以保持很好的映射性能。

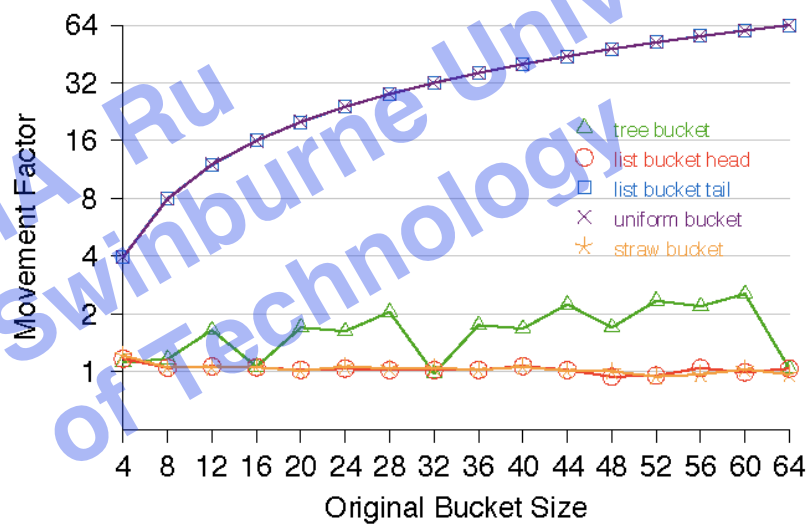

Figure 6: Efficiency of reorganization after adding items to different bucket types. 1 is optimal. Straw and list buckets are normally optimal, although removing items from the tail of a list bucket induces worst case behavior. Tree bucket changes are bounded by the logarithm of the bucket size.

图 6： 给不同类型的桶增加项之后的重组效率。1 是最优的。尽管从列表桶的尾部移除项已是最差的实例行为，草帽和列表桶通常是最优的。树桶的改变是有界的，边界由桶大小的对数决定

4.3 算法性能

设计计算 CRUSH 映射的方法是很快速的-一个包含 n 个 OSDs 的集群的速度是 $O(\log n)$，因此设备可以在集群映射改变时快速地部署任意一个对象或者重新评估那些恰好已经有存储对象的存储目标。我们测试不同规模集群中超过一百万的映射操作的 CRUSH 性能，当然我们所关注的 CRUSH 性能都与与 $RUSH_T$ 和 $RUSH_P$ 相关。图 7 表述了映射一系列副本到一个包含 8 项-树和均匀桶（层级的深度是可变的）CRUSH 集群的平均时间（微秒为单位）与映射到 RUSH 固定的两层层级机构的平均时间的对比。X 轴表示系统中设备的数量，并且以 log 对数比例出图，因此所出的图与存储层级结构的深度相关。CRUSH 性能是与设备的数量是对数相关的。由于更低的代码复杂性，$RUSH_T$ 与树桶的 CRUSH 图像接近重叠，且接着列表桶和草帽桶的图像。在该测试中，$RUSH_P$ 的图像则是线性的（消耗的时间比拥有 32768 个设备的 CRUSH 集群多 25 倍），尽管在实际情况下，最新部署的硬盘的大小是以指数成倍增长的，但是仍可以改善的次线性的扫描图像[Honicky and Miller 2004]。所有的这些测试都是在奔腾 4，2.8GHz 主频的机器上运行的，而整体的映射的时间是以几十微秒下的。

Figure 7: CRUSH and $RUSH_T$ computation times scale logarithmically relative to hierarchy size, while $RUSH_P$ scales linearly.

图 7：CRUSH 和 $RUSH_T$ 的计算时间与层级规模的对数比例图，而 $RUSH_P$ 的图像则是呈现线性的

CRUSH 的效率与存储层级的深度以及构成该层级结构的桶的类型有关。图 8 对比了以桶的大小（X）作为基础，从每种类型桶选取一个副本 $c(r,x)$ 所需的时间（Y）。在高层（high-level）中，CRUSH 以 $O(\log n)$ 为单位进行扫描测试，一层级的深度呈线性增长—所提供的大小大概为 $O(n)$ 独立的桶（列表和草帽桶线性扫描测试）不会超过规定的最大尺寸。何时何地采用不同类型的独立桶主要取决于增加，删除或者需要重定义重量的项的多少。尽管移除操作可能产生过度的数据移动，但列表桶能提供比草帽桶稍微好一点的性能优势。

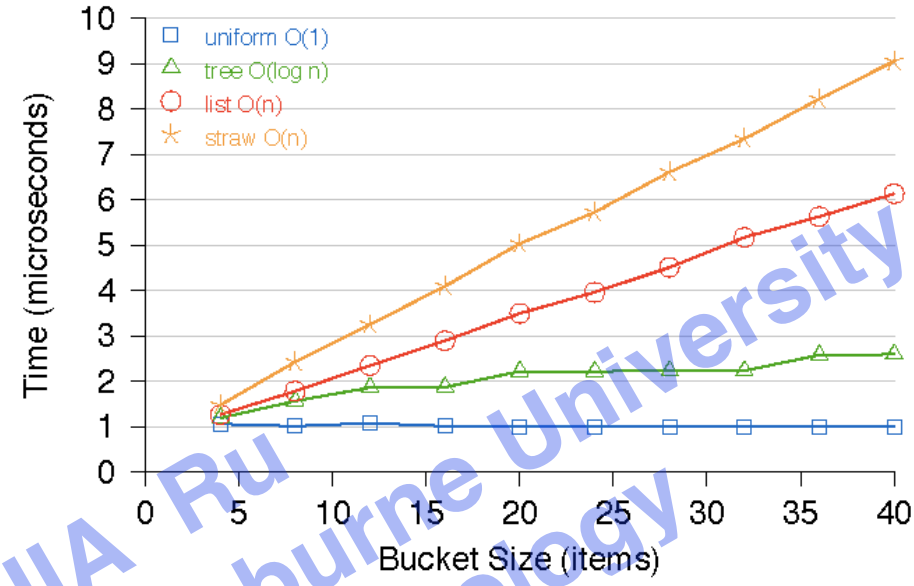

Figure 8: Low-level speed of mapping replicas into individual CRUSH buckets versus bucket size. Uniform buckets take constant time, tree buckets take logarithmic time, and list and straw buckets take linear time.

图 8：映射副本到独立的 CRUSH 桶的底层（Low-level）速度与桶规模的关系图。均匀桶的时间是不变的，而树桶的时间是对数型的，而列表桶和草帽桶的时间则是线性的。

CRUSH 的中心性能—执行时间和结果的质量—是采用整数哈希函数的。采用基于 Jenkin32 位哈希混合[Jenkins 1997]的多路输入整数哈希函数来计算伪随机值。在现有的表述形式里， 花费在 CRUSH 映射函数上的时间有大约 45%都用来计算哈希值，这使得哈希成为全局速度与分布质量的关键，同时也是优化目标。

4.3.1 可忽略的老化

在存储层级中 CRUSH 只是把失效的设备留在原位，这是因为失效多是临时的问

题（失效的硬盘通常都会被换掉）且这样做可以避免低效的数据重组。如果一个存储系统的年限可以忽略，那么失效而没有被替代的设备的数量对于系统来说则是至关重要的。尽管 CRUSH 会在未失效的设备上重新分布数据，但是由于在分配算法中，会很大的可能性去执行回溯法，所以这样做仍会产生一些性能损失。在有 1000 个设备的集群里，随着失效设备的比例的改变，我们测试不同的映射速度。在有一半设备不工作的极限失效的情况下，映射计算时间增加了 71%（通过双倍增加每个设备的工作负载使得 I/O 性能严重退化的情景）。

4.4 可靠性

对于大规模存储系统，数据的安全性是至关重要的，大量的设备使得硬件失效很常见而非是例外。像 CRUSH 那样重新聚合副本的随机化的分布策略引起了人们的兴趣，这是因为这些策略可以扩展那些可以任意共享数据的设备节点。这样做有两个优势，总体来说是两个相反的效果。首先，失效后，修复机制可以并行执行，这是因为较少位的（bits）的副本数据遍布在大量不同的节点上，可以减低修复次数同时也可以缩小那些容易受到额外失效攻击的窗口。其次，大规模的节点组可以增加失去共享数据的并发次失效的可能性。在双向镜像中，随着重新聚类，当拥有超过两个副本的数据的整体安全性增强时[Xin et al. 2004]，这两个因素可以相互取消。

然而，多路失效仍是一个很重要的课题，大体上来讲，就是不能使它们相互独立，在许多情况下，像电力失效或者物理干扰这样的独立事件都会对多路设备产生影响，而且与重聚合的副本相关的大规模的节点组也会增加数据损失的风险。CRUSH 的副本跨用户自定义的失效区域（RUSH 不存在这种情况或者存在基于哈希的情境）分布设计可以阻止由数据损失引发的相关的并发失效。尽管我们可以明确的看出这种风险被降低了，但是在不考虑存储集群配置和相关的失效历史数据研究的情况下，系统的整体可靠性提高程度还是很难被量化。尽管我们希望未来我们可以做这方面的研究，但是这已经超过了本文的研究范围。

5. 展望

我们设计 CRUSH 作为 Ceph-多路-PT 级别的分布式文件系统[Weil et al. 2006]的一部分。目前的研究包括基于 CRUSH 独有功能的智能可信赖的分布式对象存储。目前采用的 CRUSH 首要规则结构已经足够复杂，所以可以支持我们能想到的数据分布策略。某些系统有一些特殊的需求，需要更强大的规则结构来满足。

尽管设计 CRUSH 的首要目的是解决并发失效相关的数据安全问题，但是在用 Makov 或者其他的量化模型来评测准确的关于系统平均数据丢失时间（MTTDL）的效应前，我们需要研究真正的系统来判定它们的特点和频率。

CRUSH 的性能高度依赖于适时的强多输入整数哈希函数。因为它同时影响算法准确性—分布结果的质量以及速度，以及研究非常强的 CRUSH 必需的更快的哈

希散列技术。

6. 结论

分布式存储系统为数据分布的可扩展性带来了一系列明确的挑战。CRUS 把数据分配设定为一个伪随机映射函数，去除传统的分配元数据的需求，取而代之的是把数据分布到基于权重的层级描述型存储结构，通过以上这些手段，CRUSH 满足了这些挑战。集群映射层级的结构能够反映设备的底层物理组织和框架，如机架（shelve），机柜（cabinet），数据中心行结构上存储设备的组成，以及加强分配原则的定制—定义一类比较宽泛的策略来分离副本到不同的用户自定义的失效域（也就是说，独立的电力和网络结构）。这样做，CRUSH 可以减轻对已有的重新聚合副本的伪随机系统的相关设备失效的攻击。CRUSH 通过采取用最小的计算成本，选择性地从过度装载的设备上转移数据的方法，也解决了内部随机过程中设备过度装载的风险。

CRUSH 用一个非常高效的与计算效率和所需元数据有关的函数解决了以上所有的问题。映射计算需要 $O(\log n)$ 运行时间，仅需要几十微秒就可以执行数千个设备。对于大规模分布式存储系统而言，这种集效率，可靠性和灵活性为一身的鲁棒组合使得 CRUSH 不失为一个不错的选择。

7. 致谢

R. J. Honicky's 在 RUSH 上卓越的工作表现鼓舞促进了 CRUSH 的开发。与 Richard Golding, Theodore Wong, 存储系统研究中心的学生，院方的研讨极大地帮助我们来设定以及修改完善所提出的算法。同时该项研究由 Lawrence Livermore 国立实验室部分支持。

8. 可用性

CRUSH 源代码被 LGPL 授权，可在以下的网站上查找到：
http://www.cs.ucsc.edu/~sage/crush

参考文献

ANDERSON, E., HALL, J., HARTLINE , J., HOBBS , M., KARLIN , A. R., SAIA , J., SWAMINATHAN , R., AND WILKES , J. 2001. An experimental study of data migration algorithms. In Proceedings of the 5th International Workshop on Algorithm Engineering, Springer-Verlag, London, UK, 145–158.

ANDERSON, E., HOBBS, M., KEETON , K., SPENCE , S., UYSAL , M., AND VEITCH , A. 2002. Hippodrome: running circles around storage administration. In Proceedings of the 2002 Conference on File and Storage Technologies (FAST).

AZAGURY, A., DREIZIN , V., FACTOR , M., HENIS , E., NAOR , D., RINETZKY , N., RODEH , O., SATRAN , J., TAVORY , A., AND YERUSHALMI , L. 2003. Towards an object store. In Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies, 165–176.

BRAAM, P. J. 2004. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug.

BRINKMANN, A., SALZWEDEL, K., AND SCHEIDELER , C. 2000. Efficient, distributed data placement strategies for storage area networks. In Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA), ACM Press, 119–128. Extended Abstract.

CHOY, D. M., FAGIN, R., AND STOCKMEYER , L. 1996. Efficiently extendible mappings for balanced data distribution. Algorithmica 16, 215–232.

GHEMAWAT, S., GOBIOFF, H., AND LEUNG , S.-T. 2003. The Google file system. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), ACM.

GOBIOFF, H., GIBSON , G., AND TYGAR , D. 1997. Security for network attached storage devices. Tech. Rep. TR CMU-CS-97-185, Carniege Mellon, Oct.

GOEL, A., SHAHABI, C., YAO , D. S.-Y., AND ZIMMERMAN, R. 2002. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In Proceedings of the 18th International Conference on Data Engineering (ICDE '02), 473–482.

GRANVILLE , A. 1993. On elementary proofs of the Prime Number Theorem for Arithmetic Progressions, without characters. In Proceedings of the 1993 Amalfi Conference on Analytic Number Theory , 157–194.

HONICKY, R. J., AND MILLER , E. L. 2004. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004), IEEE.

JENKINS, R. J., 1997. Hash functions for hash table lookup. http://burtleburtle.net/bob/hash/evahash.html.

KARGER, D., LEHMAN , E., LEIGHTON , T., LEVINE , M., LEWIN , D., AND PANIGRAHY , R. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on theWorldWideWeb. In ACM Symposium on Theory of Computing, 654–663.

LUMB, C. R., GANGER, G. R., AND GOLDING , R. 2004. D-SPTF: Decentralized request distribution in brick-based storage systems. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 37–47.

NAGLE, D., SERENYI, D., AND MATTHEWS , A. 2004. The Panasas ActiveScale storage cluster-delivering scalable high bandwidth storage. In Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04).

RODEH, O., AND TEPERMAN , A. 2003. zFS-a scalable distributed file system using object disks. In Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies, 207–218.

SAITO, Y., FRØLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. 2004. FAB: Building distributed enterprise disk arrays from commodity components. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 48–58.

SANTOS, J. R., MUNTZ, R. R., AND RIBEIRO-NETO, B. 2000. Comparing random data allocation and data striping in multimedia servers. In Proceedings of the 2000 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, ACM Press, Santa Clara, CA, 44–55.

SCHMUCK, F., AND HASKIN, R. 2002. GPFS: A shared disk file system for large computing clusters. In Proceedings of the 2002 Conference on File and Storage Technologies (FAST), USENIX, 231–244.

TANG, H., GULBEDEN, A., ZHOU, J., STRATHEARN, W., YANG, T., AND CHU, L. 2004. A self-organizing storage cluster for parallel data-intensive applications. In Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04).

WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. 2006. Ceph: A scalable, high-performance distributed file system. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI).

XIN, Q., MILLER, E. L., AND SCHWARZ, T. J. E. 2004. Evaluation of distributed recovery in large-scale storage systems. In Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC), 172–181.