|  |  |
| --- | --- |
| 学号 | 2017282110250 |
| 密级 |  |

**海量数据存储**

|  |
| --- |
| **《**Knockoff: Cheap Versions in the Cloud**》**  **读书报告** |

|  |  |
| --- | --- |
| 院（系）名 称 ： | 计算机学院 |
| 专 业 名 称 ： | 计算机技术 |
| 学 生 姓 名 ： | 王婷婷 |
| 指 导 教 师 ： | 何水兵 副教授 |

二〇一七年十二月

目录

[1 论文信息 1](#_Toc500430481)

[1.1 题目 1](#_Toc500430482)

[1.2 论文链接 1](#_Toc500430483)

[1.3 作者信息 1](#_Toc500430484)

[2 论文的主要工作 2](#_Toc500430485)

[2.1 工作目的 2](#_Toc500430486)

[2.2 核心方法 2](#_Toc500430487)

[2.3 论文贡献 2](#_Toc500430488)

[3 论文背景 3](#_Toc500430489)

[4 论文重要概念 5](#_Toc500430490)

[4.1 确定性记录和重放 5](#_Toc500430491)

[4.2 文件：值还是操作? 5](#_Toc500430492)

[5 Knockoff方案 7](#_Toc500430493)

[5.1 版本控制 7](#_Toc500430494)

[5.2 Knockoff体系结构 8](#_Toc500430495)

[5.2.1 客户端 8](#_Toc500430496)

[5.2.2 服务器 9](#_Toc500430497)

[5.3 写入数据 9](#_Toc500430498)

[5.4 存储数据 11](#_Toc500430499)

[5.5 读取数据 13](#_Toc500430500)

[5.6 优化：日志压缩 15](#_Toc500430501)

[6 系统性能评估 17](#_Toc500430502)

[论文总结 18](#_Toc500430503)

# 论文信息

## 题目

Knockoff: Cheap Versions in the Cloud

## 论文链接

<https://www.usenix.org/conference/fast17/technical-sessions/presentation/dou>

## 作者信息

Xianzheng Dou

University of Michigan

Peter M. Chen

University of Michigan

方向：操作系统 虚拟机 文件系统 容错 安全性

Google资料：

<https://xue.glgoo.net/citations?user=dVZNQqUAAAAJ&hl=zh-CN&oi=sra>

Jason Flinn

University of Michigan

方向：操作系统 移动计算 存储

Google 资料：

<https://xue.glgoo.net/citations?user=YxGHbuMAAAAJ&hl=zh-CN&oi=ao>

# 论文的主要工作

## 工作目的

设计并实现一种云端数据存储与读写机制，在保证用户数据不受破坏，系统性能均衡稳定的条件下，实现大幅降低在客户端和服务器之间传输文件数据的成本；降低保存和维护多个版本数据的成本。

## 核心方法

映入log机制，用操作序列代替数据文件。

## 论文贡献

描述了分布式文件系统Knockoff的设计和实现，该文件系统选择性地用文件数据替换为生成该数据所需的不确定性日志，以便与云服务器通信以及云存储。Knockoff支持多种版本控制频率：无版本控制版本，基于文件关闭的版本，基于系统调用的版本，以及基于存储指令（映射文件）的版本。

* 通过利用确定性记录和重放，我们为分布式文件系统提供第一个通用运行解决方案。
* 我们展示了如何使用压缩来压缩计算和存储来减少非确定性日志的大小。
* 我们量化了该方案应用在现有分布式文件系统的成本和收益。

我们通过对软件开发场景和20天的单用户纵向研究来评估Knockoff。如果没有版本控制，Knockoff在这些研究中将文件发送到云端的平均成本分别降低了24％和21％。 使用Knockoff的好处会随着版本的保留频率的增加而增加。这种方法的成本是录制执行的性能开销（在我们的评估中为7-8％）以及在取回以前的版本中更大的延迟（对于我们的默认设置高达60秒）。

# 论文技术背景

Knockoff是基于这样一个原则，即人们可以通过数值或计算所需的输入日志来表示由计算生成的数据。我们称之为等价原则（值与计算之间）； 它已被观察到并在许多环境中使用； 例如容错机制[16]，状态机复制[37]，数据中心存储管理[20]和状态同步[19]。

与我们最相关的项目也使用等价原则，即减少分布式文件系统中客户端和服务器之间的通信开销。Lee等人首先在Coda文件系统中应用这个原则[22,23]，并创造了术语*operation shipping*。客户端登录并将用户操作（例如，shell命令）发送到服务器代理，以重播操作以重新生成数据。Chang等人利用这个想法来记录和发送用户活动，如键盘和鼠标输入[10]。

虽然*operation shipping*的基本思想是强大的，但是先前的系统记录和运输非确定性的类型非常有限，因此不能保证通过日志接收到的状态与原始状态相匹配。shell命令的日志和用户活动日志都不足以重现通用程序的计算。研究人员认识到了这个缺点，并通过补充重复计算与前向纠错等补偿这个缺陷，使用hash检测剩余差异并恢复值来减轻这一缺陷。 不幸的是，转向多处理器和多线程程序意味着许多不确定性程序在这些现有系统中是不能处理的。此外，由于这些现有系统处理的非确定性输入非常有限，所以它们在记录和重放方面需要相同的环境，这在许多客户端 - 服务器设置中是不现实的。

Knockoff应用了相同的基本原则，但是它使用非确定性的综合日志来为所有的无竞争程序（以及偶尔的数据竞争的许多程序）提供等同性。这使得Knockoff可以在更多设置中使用*operation shipping*，而且还可以对此类设置进行*operation shipping*的第一实际评估（以前的研究大都假定程序是确定性的，这是不现实的）。Knockoff还将操作运输应用于版本控制文件系统。我们发现，在保存多个版本的情况下，*operation shipping*的收益较大，并且在eidetic粒度保存版本时不可或缺。

Adams等人识别重新计算是一种减少存储的方法[1]，但是不要基于这种观察来实现或评估任何系统。其他系统使用重计算来减少在受限环境中的存储，其中计算保证是确定性的。 Nectar [20]将这个想法应用于DryadLINQ应用程序，这两个应用程序都是确定性的和功能性的。BADFS [6]使用重新计算来代替复制数据；用户必须指定明确的依赖关系，并且计算必须是确定性的才能生成相同的数据。

除了复制数据之外，文件系统中还使用日志记录来跟踪文件的来源[29,41]，并指导何时保存新版本[28]。 更一般地说，重做日志[26]在出现故障时提供事务性质。

许多先前的系统对文件数据进行重复数据删除以减少通信和存储开销[12,21,31,38,40,43] 。LBFS [31]使用基于块的重复数据删除，其中Rabin指纹将文件分割成块，计算每个块的散列值，以及客户端和服务器使用散列值来避免通信对方已经看到的块。在按值传输数据时，Knockoff使用LBFS样式的重复数据删除。

版本控制文件系统[24,30,36,39,44]以特定粒度保留过去的状态，例如每个文件关闭或每个修改。云存储提供商DropBox [14]和Google Drive [18]等，目前允许用户保留以前版本的文件数据。Knockoff通过减少存储和计算成本使版本控制文件系统更高效。它还支持比以前的系统更精细的版本控制。

# 论文重要概念

论文中提到了基于log的记录重放和文件表示方式两个重要的概念，下面是对两个概念的总结。

## 确定性记录和重放

为了使用运行实际的工作负载和环境，我们需要一个通用的确定性记录和重放系统。记录/重放系统应支持未经修改的应用程序，并为多线程程序工作。为了在实际的客户机/服务器配置中工作，记录/重放系统应该允许记录的执行在与记录的执行环境不同的环境中重播。最后，为了使operation shipping可以用于一些（但不是全部）过程，系统应该单独记录每个应用程序，并允许每个应用程序在服务器上单独重播。

Knockoff使用符合这些要求的Arnold系统[13]。Arnold使用修改的Linux内核来记录Linux进程的执行情况。它记录了进入某个过程的所有非确定性数据，包括系统调用的结果（如用户和网络输入），信号的时间以及实时时钟查询。由于它在重放时提供记录的值，而不是重新执行与外部依赖进行交互的系统调用，所以Arnold可以在一台计算机上记录应用程序，并在另一台计算机上重放。唯一的要求是两台计算机运行Arnold内核并具有相同的处理器体系结构（x86）。

Arnold通过记录所有同步操作（例如，pthread锁和原子硬件指令）来启用多线程程序的确定性重播。阿诺德可以通过数据竞争来检测程序，但是并不能保证这些程序的重放将与其记录的执行相匹配。阿诺德确实保证重播总是可重复的（即对于其他重放是确定性的），即使是对于活泼的程序。

## 文件：值还是操作?

Knockoff可以用以下两种方法之一来表示文件：作为普通文件数据（按值）或作为重新创建文件所需的非确定性的记录（通过操作）。哪种表示方式更具成本效益取决于生成文件数据的程序的特性以及计算，通信和存储的相对成本。由大部分确定性的程序（例如，照片编辑软件）生成的较大的文件最好用操作表示。相反，使用大量非确定性数据（例如，密钥生成）的程序生成的小文件最好用值表示。

在任何时候，Knockoff都可以使用确定性重放来将由操作表示的文件转换为值（但反之并不如此）。为此，Knockoff加载并重新执行原始程序，然后输入原始执行中记录的不确定性日志。请注意，来自Knockoff的系统调用读取的文件数据不包含在日志中。而这些日志条目是指读取的文件和版本，Arnold的重播系统从Knockoff读取这些数据。通常情况下，应用程序的二进制文件、动态库、配置文件等都存储在Knockoff中，因此服务器重放在原始记录期间看到与客户端上存在的相同的应用程序文件。如果二进制文件或库没有存储在Knockoff中，则重放提供的文件数据将被包含在非确定性日志的值中。先前录制的应用程序的重放可能需要保留以前的文件版本。或者，我们可以通过生成数据时应用程序的递归重播来重新生成这些旧版本。

每当Knockoff通过操作呈现一个文件，它必须首先验证Arnold的重播是否忠实地重建了文件数据，因为Arnold并不保证具有数据竞争的程序完全按照记录重播。Knockoff使用每个文件的SHA-512散列来验证重播是否正确地生成了原始数据。 因为在Arnold中的重放是可重复的，所以在第一次重播中产生匹配数据的运行被保证在随后的所有重放中产生匹配的数据。如果重放没有产生匹配的数据，Knockoff切换到按值表示文件。

当它在客户端和服务器之间传送文件时以及在服务器上存储文件时，Knockoff会在这两个表示之间进行选择。为了指导它的选择，Knockoff需要测量创建每个文件的计算时间和每个文件的大小。

# Knockoff方案

Knockoff是一个C/S模式的分布式文件系统，服务器托管在云中。服务器存储所有文件的当前版本，并根据用户选择的版本策略选择存储所有文件的以前版本。 Knockoff客户端有一个本地磁盘缓存，用于存储当前和过去的文件版本（可选）。

Knockoff实现了Coda风格的弱文件一致性[27]。客户端异步传输文件系统的更新到服务器。客户端在缓存当前版本的文件时向服务器注册一个回调，服务器在另一个客户端修改文件时通过向客户端发送消息来中断回调。

Knockoff将版本向量[33]与每个文件相关联，以识别特定的版本并检测冲突的更新。Knockoff为客户分配唯一的标识符；每当客户端执行修改文件的系统调用时，它就增加与其标识符相关联的版本向量中的整数。因此，每个文件的过去版本都有一个唯一的版本向量，可以用来命名和检索该版本。服务器通过比较每个更新的版本向量来检测冲突的更新。如果发生冲突，则服务器保留两个版本，用户手动解决冲突。

Knockoff客户端记录几乎所有的用户级进程执行（不包括某些服务器，如X服务器和sshd），内核为每次操作生成一个不确定性日志。不确定性日志存储在客户端的日志缓存中，也可以发送到服务器并存储在相应的数据库中。服务器有一个重放引擎，允许它从这些日志重新生成文件数据。

## 版本控制

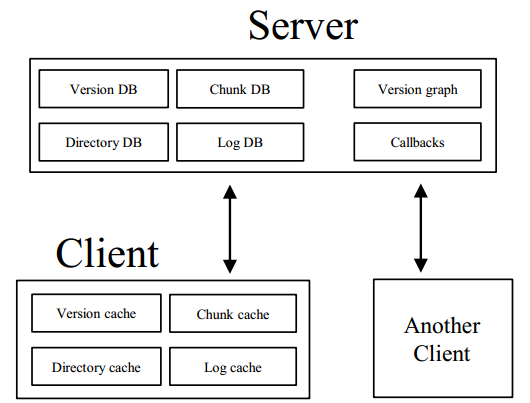
Knockoff支持基于每个文件系统的多种版本控制策略。用户可以选择下列之一：

* 没有版本控制。 Knockoff只保留所有文件的当前版本。对于持续性，客户端将关闭的修改后的文件发送到服务器。在第一次关闭之后，Knockoff最多等待10秒钟，将文件修改发送到服务器（这个延迟允许合并多个更新，这些更新在时间上紧密地结合在一起）[27]。在收到文件修改后，服务器将覆盖该文件的以前版本，并中断其他客户端的回调。只有在冲突更新的情况下，服务器才会保留多个版本。
* 版本关闭。Knockoff以紧密的粒度保留所有以前的版本；对于以前的版本，Knockoff可能会存储实际数据或重新生成数据所需的日志。在接收文件修改时，服务器保留以前的版本而不是覆盖它。客户可以通过指定其唯一的版本向量来请求版本。
* 写入版本。Knockoff在写入粒度中保留所有以前的版本。每个修改文件的系统调用都会创建一个新的版本，Knockoff可以重现所有这些版本。
* Eidetic。Knockoff以指令粒度保留所有以前的版本。它可以重现任何计算或文件数据，并通过Arnold确定数据的来源。服务器存储所有应用程序日志。客户端可以通过指定版本向量和指定计数来指定文件的特定版本，该计数指定何时停止重播（以恢复映射文件的特定状态）。

## Knockoff体系结构

图1显示了Knockoffs存储架构。客户端运行FUSE [17]用户级文件系统守护进程

图1 Architecture overview



### 客户端

Knockoff客户端将文件系统数据存储在四个永久性缓存中，以隐藏网络延迟。整个文件版本存储在版本缓存中；这个缓存可以同时保存同一文件的多个版本。Knockoff中的每个文件都有一个唯一的fileid域，因此可以通过指定fileid域和版本向量来从缓存中检索特定版本的文件。版本缓存跟踪它存储的当前版本，它为这些条目设置回调，如果服务器中断了一个回调（因为另一个客户端已经更新了文件），该版本被保留在缓存中，但是它不会被指定为“当前版本”了。

块缓存存储基于块的重复数据删除为版本缓存块中的每个文件生成的存储数据块；因此版本高速缓存仅包含指向块高速缓存中的块指针。Knockoff使用基于块的重复数据消除的LBFS算法[31]，将数据库中的每个条目分成块，并计算每个这样的块的SHA-512哈希。块数据库利用这些hash值进行索引。

目录数据存储在一个单独的Berkeley DB [7]目录缓存中。Knockoff客户端通过这个数据实现一个内存索引的快速路径查找。日志缓存存储记录应用程序执行产生的不确定性日志。

所有的客户端缓存都通过LRU驱逐来管理，优先考虑保留当前版本而不是旧版本。如果版本缓存中的任何版本不再使用块，则会从块缓存中删除。修改的值被固定在缓存中，直到它们被保存到服务器。

### 服务器

服务器维护这些客户端存储的类似产品。服务器的版本数据库存储每个文件的当前版本，并根据版本策略确定过去的文件版本。块数据库存储版本数据库中所有文件的块数据，由块SHA-512散列值进行索引。目录DB存储Knockoff的所有目录信息，日志DB存储客户端记录的非确定性日志。

如果版本控制策略是eidetic，则日志数据库将存储由Knockoff客户端每个应用程序产生的日志。如果版本控制策略是写入版本，则每个过去的文件版本都存储在版本数据库中，或者重现该版本所需的日志存储在日志数据库中。如果版本控制策略是close版本，则此不变量仅适用于与文件关闭对应的版本。如果版本控制策略不是版本控制，则仅存储文件和目录数据的当前版本。

服务器还维护客户端为每个文件设置的回调。如果客户端更新文件，则服务器使用这些来确定向哪些客户端发出更新通知。

最后，服务器维护将文件版本与产生数据的计算相关联的版本图。图中的节点是记录非确定性（表示特定执行）或文件版本的记录。每个边代表一个记录写入的文件数据的范围，并且可以由另一个记录或文件版本的一部分读取。一个索引允许Knockoff通过fileid和版本向量快速找到一个特定的文件版本。如果版本数据库中没有版本，则版本图允许Knockoff确定重播哪些日志来重新生成数据。

## 写入数据

Knockoff旨在降低云存储成本，而绝大部分成本花费在通信上。例如，AT＆T [4]和Comcast [11]目前每50GB的数据通信收费高达10美元，而目前亚马逊2个vCPU和4GB内存的实例每个小时收费0.052美元 [5]。这意味着如果Knockoff可以使用1秒的云计算来节省76 KB的通信，那么可以降低成本。

我们首先描述Knockoff如何处理针对其版本策略的写入，然后推广到其他策略。像Arnold一样，Knockoff将应用程序的执行不确定性序列记录下来。当应用程序关闭文件时，Knockoff客户端检查文件的版本向量以确定它是否被修改。如果被修改，Knockoff则启动一个10秒的定时器，允许应用程序进行进一步的修改。当计时器到期时，Knockoff再启动一个事务，在该事务中，该应用程序对文件的所有未完成的修改都被发送到服务器。如果应用程序终止，Knockoff立即启动事务。

为了将修改传播到服务器，Knockoff首先计算发送和重播给定预定义的通信成本（costcomm）和计算（costcomp）的非确定性的日志成本：

|  |  |
| --- | --- |
|  | （1） |

sizelog是通过压缩写入文件的应用程序的不确定性日志并直接测量其大小来确定的。 由于Knockoff目前不支持检查点，因此每个日志都必须从应用程序启动重播。

为了估计时间，Knockoff修改了Arnold以将所记录的应用程序到目前为止消耗的用户CPU时间与每个修改文件数据的日志条目进行存储。这是对重放客户端日志所需时间的一个很好的估计[34]。为了估计服务器重播时间，Arnold将该值乘以一个转换因子，以反映客户端和服务器的相对CPU速度。

|  |  |
| --- | --- |
|  | （2） |

Knockoff实现LBFS使用的基于块的重复数据删除算法，以降低传输文件数据的成本。它将所有修改后的文件分解为块，散列每个块，并将散列发送到服务器。服务器用它存储的hash集合进行响应。sizechunks是服务器未知的块需要传输的大小； Knockoff使用gzip压缩来减少为这样的块传输的字节。

如果正在重播的应用程序没有数据竞争，则重放会产生相同的数据。对于某些程序（例如单线程程序），可以保证数据库的自由，但对于复杂的应用程序则不能。因此，Knockoff会将每个修改文件的SHA-512hash值与日志一起发送到服务器。Knockoff服务器验证这个散列关闭。如果验证失败，则要求客户发送文件数据。请注意，这样的竞争在Arnold中很少见，因为重放系统本身就是一个有效的数据竞争探测器[13]。

如果costdata <costlog，则Knockoff可以通过将唯一块发送到服务器来降低当前事务的成本。但是，对于长时间运行的应用程序，可能会发送和重播迄今为止收集的日志，这有助于降低未来文件修改的成本（因为从这一点重放的成本低于从程序的开始）。 Knockoff通过查看应用程序的costdata / costlog比率的历史来预测这一点。如果发送日志历史上是有利的，并且当前的应用程序行为与过去的执行相似（比率相差不到40％），则发送日志。否则，它发送唯一的数据块。

## 存储数据

Knockoff可以通过值（作为正常的文件数据）或操作序列（作为重新计算该数据所需的非确定性的日志）来将文件数据存储在服务器上。如果非确定性的日志小于它产生的文件数据，则通过操作存储文件节省空间和金钱。但是，按操作存储文件会延迟将来读取数据，因为Knockoff将需要重放生成数据的原始计算。一般来说，这意味着如果数据非常冷，Knockoff应该只能通过操作来存储文件数据；即，如果将来读取数据的概率是非常低的。

Knockoff目前实现了一个简单的策略来决定如何在服务器上存储数据。它总是按值存储每个文件的当前版本，以使其当前文件数据的读取性能与传统文件系统相同。如果通过日志存储数据的存储要求小于按值存储数据的存储要求，则关闭可以通过操作存储以前的版本。但是，Knockoff也有一个配置参数，它设置了一个最大的实现延迟，这是重建操作存储的任何版本的时间。默认实现延迟是60秒。

根据这个策略，Knockoff保留所有不确定性的日志，因为它们需要重现过去的计算和瞬态过程状态。这足以重新计算任何文件版本。但是，由于递归依赖关系，重新计算时间是无界的。例如，生成过去的文件版本可能需要重放记录的计算。 该计算可能已经从Knockoff读取文件数据，所以Knockoff还必须通过重放其他应用程序来重现这些文件版本。 这继续递归，直到Knockoff遇到没有更多的过去的版本重现。

图中的版本节点包含该版本中由不同系统调用写入的所有文件字节范围的列表。对于每个范围，节点将日志和特定系统调用存储在写入数据的日志中。重播所有这些日志到指定的系统调用将足以重新计算该特定的版本。但是，版本数据库中已经存在的字节范围不需要重新计算。如果由节点表示的版本本身在版本DB中，则所有字节范围都存在。否则，特定的字节范围可能仍然在版本数据库中，因为同一个文件的另一个版本是按值存储的，并且范围在两个版本之间没有被覆盖。

如果日志包含写入不在版本数据库中的字节范围的系统调用，则Knockoff将版本节点的边缘插入日志节点。每个边的权重等于重新计算所有这些字节范围的时间。

## 读取数据

默认情况下，从Knockoff读取数据的任何应用程序都将接收文件的当前版本。客户端首先检查其版本缓存，以查看它是否在本地存储了当前版本的文件。如果版本存在，Knockoff从缓存中读取请求的数据。它还在读取应用程序的非确定性日志中附加一条记录，该记录指定写入刚才读取的数据的应用程序的fileId，版本向量以及logId和系统调用。后两个值来自于Arnold的文件图[13]。

如果版本不存在，则Knockoff从服务器获取并缓存该版本。 Knockoff缓存整个文件版本，客户端通过value从服务器获取版本。客户端发送一个指定fileId的请求。服务器用当前版本向量和包含该文件的每个块的散列列表进行响应。服务器也发送版本的文件映射。客户端指定哪些数据块没有被缓存，服务器将数据发送到客户端。服务器在文件上设置一个回调。客户端将版本插入到其版本缓存中，将其标记为当前版本，并将版本的块放入其块缓存中。

应用程序也可以通过指定版本向量来读取以前版本的文件。 如果请求的版本在服务器的版本数据库中，则按上述方式发送。如果不存在，则必须通过重播一个或多个日志来重新计算它。 我们接下来描述这个过程。

在其版本图中，服务器维护所有版本的索引；这使得它可以快速找到正在请求的特定版本节点。版本节点显示由不同系统调用写入的不同字节范围。如果范围在版本DB中，则直接使用该范围。否则，服务器必须重放写入范围的应用程序的日志以重新生成数据。对于每个这样的日志，它决定了必须重播的最长前缀；这是写入任何正在读取的范围的日志中的最后一个系统调用。Knockoff检查每个这样的日志前缀，以确定重播日志是否需要不在版本缓存中的文件数据。如果是这样，它会递归地访问写入所需数据的日志。请注意，Knockoff的实现延迟限制了生成任何版本所需的计算量。然后敲门重放访问日志以重新生成所需的版本。它将这个版本放在它的版本数据库中，然后像上面描述的那样发送给客户端。

## 优化：日志压缩

在实施Knockoff的同时，我们看到了基于块的重复数据删除在降低通信和存储成本方面的有效性。这使我们想知道：我们可以将相同的压缩技术应用于当前文件系统应用于文件数据的不确定性日志吗？

我们首先尝试直接应用基于块的重复数据删除来记录数据。这个工作很合理。但是，在仔细检查我们生成的日志之后，我们意识到日志之间的相似之处是往往不同于文件之间的相似之处。类似的文件往往有相同的大块连续块，而类似的日志往往缺乏这样的区域。相反，两个日志区域内的大部分字节可能是相同的，但在每个区域中都存在一些不同的值，例如时间戳。所以，即使是非常相似的日志块散列到不同的值。

因此，我们转向增量编码。Knockoff首先标识一个预期与当前日志类似的引用日志。然后它通过xdelta [45]生成一个二进制补丁，编码当前日志和参考日志之间的差异。同时给定参考日志和补丁，Knockoff可以重建日志中的原始值。

当应用程序生成日志时，客户端和服务器将识别引用日志。客户端查询日志高速缓存，以查找与本地存储的相同可执行文件的所有先前日志。对于每个日志，日志高速缓存将参数存储到应用程序，非确定性数据的大小，应用程序的运行时间以及用户级CPU时间。客户端通过这些指标的相似性来对缓存的日志进行排序；如果应用程序在发送日志之前尚未完成执行，则只有参数用于确定相似性，因为其他参数尚不知道。使用余弦字符串相似性来比较参数。

最后，我们研究在不确定性日志上使用delta压缩的好处。 在所有日志中，增量压缩将存储这些日志所需的字节数减少了42％。相比之下，基于块的重复数据删除将日志的大小仅减少了33％。

我们感到有趣的是，基于块的重复数据删除对压缩文件数据更有效，而压缩数据则更有效地压缩产生该数据的计算中的不确定性。有可能重构日志以使其更适合于增量压缩 或基于块的重复数据删除可能会导致进一步节省。

# 系统性能评估

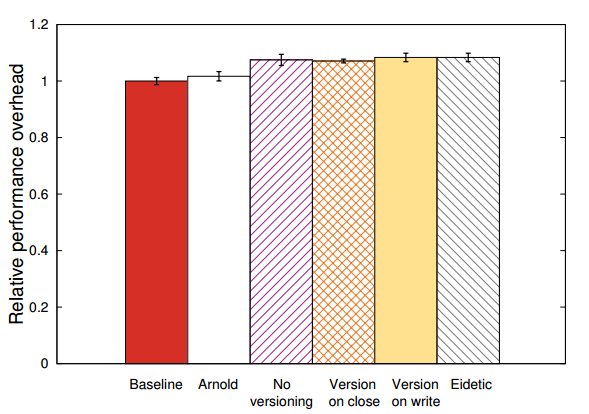
评估回答了以下问题：

* 与当前的云存储解决方案相比，Knockoff可以降低多少带宽使用量？
* Knockoff减少多少通信和存储成本？
* 什么是Knockoff的性能开销？
* 日志压缩的效果如何？

接下来，我们将检查Knockoff的用户感知的性能开销。我们通过编译libelf-0.8.9来测量所有源文件，可执行文件和存储在Knockoff中的编译输出。我们报告了8次试验的平均时间。请注意，Knockoff异步地将数据发送到服务器，服务器也以异步方式重放日志。作为基准，我们使用FUSE文件系统，将所有文件系统操作简单地转发到本地的ext4文件系统。

图2显示了我们的实验结果。第一栏显示了基准文件系统，下一栏显示了使用Arnold记录基准文件系统中存储的数据的编译的相对性能。这表明使用孤立的成本 确定性记录和重播是2％。然后，我们使用Knockoff的不同版本策略显示相对性能。 使用Knockoff的平均性能开销范围从7％到8％；不同策略的相对成本在实验误差内是相等的，如重叠的95％置信区间所示。

图2 Performance overhead building libelf-0.8.9



论文总结

随着云存储的不断发展，用户对在云端文档存储多版本的需求越来越强烈，本论文就是在这个大环境下，提出一种基于日志的文档历史版本存储策略。本论文设计实现了Knockoff系统，并对该系统进行了性能测试。实验测试结果表明该方法可以很好的降低网络通信花销和存储花销。

Knockoff可以用数据文件和日志文件两种方法之一来表示文件，数据文件就是原来存储的用户文件，日志文件是用户在之前文档基础上的操作记录。具体那种方式更好不能一锤定音，它取决于该文件通信、存储成本和由日志文件生成文件的计算，通信和存储成本。

（1）在写入数据方面：通过论文5.3公式（1）和公式（2）计算相应的costdata与costlog，并通过一定的判断机制，决定是现将数据文件直接写入服务器还是将操作日志文件写入数据库。

（2）在存储数据方面: Knockoff目前实现了一个简单的策略来决定如何在服务器上存储数据。它总是按值存储每个文件的当前版本，以使其当前文件数据的读取性能与传统文件系统相同。如果通过日志存储数据的存储要求小于按值存储数据的存储要求，则关闭可以通过操作存储以前的版本。但是，Knockoff也有一个配置参数，它设置了一个最大的实现延迟，这是重建操作存储的任何版本的最大延迟，默认实现延迟是60秒。

（3）读取数据方面：默认情况下，从Knockoff读取数据的任何应用程序都将接收文件的当前版本。客户端首先检查其版本缓存，以查看它是否在本地存储了当前版本的文件。如果版本存在，Knockoff从缓存中读取请求的数据。

**该系统可能存在的问题：**

1. 多个用户同时访问同一块数据，基于操作记录文件，可能会是控制逻辑变得复杂，处理不当坑能会出现读“脏数据”的情况。
2. 利用日志重现文档的方式重现的时间是无法准确预估的，存在迭代重现的问题，即重现c文件需要先重现b文件，而重现b文件由需要重现a文件，额外的时间消耗并不小。虽然文中提到保存一部分中间版本，打断迭代重现，但是具体保存那些版本并不能给出可规可循的依据。