

订阅DeepL Pro以编辑此文档。  
访问www.DeepL.com/Pro，了解更多信息。



**Jockey:一个用于记录-重放调试的用户空间库**

Yasushi Saito

互联网系统和存储实验室

惠普实验室帕洛阿尔托

HPL-2005-46

2005年3月7日\*

电子邮件：ysaito@hpl.hp.com

Jockey, 调试, 本文介绍了*Jockey*，一个用于Linux调试分布式Linux程序的执行记录/重放工具。Jockey是作为一个

的共享对象文件，并与目标程序链接。Jockey重写那些具有非确定性效果的系统调用和CPU指令，并记录和重放它们的调用。它支持通过采取周期性的进程检查点来诊断长期运行的程序，并实现"时间旅行"到执行的任意时刻。对于CPU密集型的应用，Jockey的性能开销可以忽略不计，对于I/O密集型的网络服务器，在最坏的情况下，Jockey的性能开销为30%。由于Jockey和目标程序作为一个单一的进程运行，共享包括内存和文件描述符表在内的所有资源，因此Jockey的实现比较复杂。本文介绍了我们开发的技术，以确保确定性的执行记录和重放，即使是错误的程序。

\* 仅限内部加入日期 已批准对外出版 © Copyright 2005 Hewlett-Packard Development Company, L.P. 版权所有。

Jockey:一个用于记录-重放调试的用户空间库

Yasushi Saito

Hewlett-Packard Laboratories ysaito@hpl.hp.com

# 摘要

本文介绍了*Jockey*，一个用于调试Linux程序的执行记录/重放工具。Jockey以一个与目标程序链接的共享对象文件的形式实现。Jockey可以重写系统调用和具有非确定性效果的CPU指令，并记录和重播它们的调用。它支持通过采取周期性的进程检查点来诊断长期运行的程序，并实现"时间旅行"到执行的任意时刻。对于CPU密集型的应用，Jockey的性能开销可以忽略不计，对于I/O密集型的网络服务器，在最坏的情况下，Jockey的性能开销为30%。由于Jockey和目标程序作为一个单一的进程运行，共享所有的资源，包括内存和filedescriptor表，因此Jockey的实现比较复杂。本文介绍了我们开发的技术，以确保确定性的执行记录和重放，即使是错误的程序。

# 介紹

Jockey是一个Linux的记录/重播工具。它记录了一个普通的Linux程序的执行过程，并在之后确定地重放它。Jockey 被设计用来帮助调试交互式或分布式的程序，例如 FAB[28]，这些程序以复杂的方式与操作系统或其他计算机通信。Jockey将通过[http://www.freshmeat.net](http://www.freshmeat.net/) 公开提供[。](http://www.freshmeat.net/)

传统的调试器，如gdb，为调试单节点、顺序程序提供了全面的支持[24]，但它们对于分布式或交互式程序[13，7]并不那么有用。我们确定了三个关键问题，并讨论了Jockey如何缓解这些问题。

首先，这类程序的执行本质上是不确定的。根据与操作系统、用户或其他进程的交互，每个进程的行为都会有所不同。作为琐碎的例子，时间系统调用会根据一天的时间返回值，而选择系统调用会根据此刻内核网络缓冲区的状态返回值。Jockey通过记录一个进程的执行情况，记录程序的每一个非确定性选择，并在以后根据开发者的意愿重放多次，来帮助调试这种非确定性程序。这样一来，一个非确定性程序的调试就被简化为一个连续的、可重复的程序的调试。

其次，这些程序往往会运行很长时间，要么是因为它们需要大量的资源（如科学计算），要么是服务器程序（如分布式哈希表），要么需要大量的用户交互（如电子表格）。简单地在这样的系统中重现bug，往往考验开发者的耐心。Jockey通过允许在执行过程中对进程状态进行检查点来解决这个问题。开发者可以从任何检查点重放执行，并轻松地在执行过程中进行"时间旅行"，以诊断程序到底出了什么问题。

第三，运行一个分布式系统需要在多台计算机上启动多个进程，这很麻烦，而且增加了程序开发的周转时间。Jockey 缓解了这个问题，因为 Jockey 可以记录和重放单个进程--在运行和记录整个系统后，开发者可以使用传统的调试器重放每个进程。当然，这也是一个局限性；当人们想同时查看多个进程的执行情况时，Jockey的作用可能会降低。我们将在第5.2节进一步讨论这个问题。

## 赛马会：目标和方法

Jockey的设计有两个特别的目标。首先是*易用性*。Jockey必须易于和安全地安装和部署；它应该在不需要修改目标源代码或调试器的情况下工作。第二是*通用性*。

Jockey应该能够处理通用的服务器程序，而不仅仅是那些用特定的编程语言或API编写的程序，比如MPI或CORBA[14]。

我们通过将Jockey作为一个用户空间库来实现第一个目标，它作为目标程序的一部分来运行。与基于内核的方法[31]相比，Jockey的方法更安全，也更容易使用--任何人都可以在没有管理员权限、没有打过补丁的内核或调试器的情况下使用它。它也允许目标程序很容易地控制或扩展Jockey。我们将在第4节讨论其中的一些特性。第二个目标是通过记录和重放一个相当低级的事件来实现的--系统调用和CPU指令。

用户空间的实现带来了挑战和限制。首先，它不能处理线程上下文切换或共享内存通信等事件。其次，因为Jockey和目标程序运行在同一个保护域中，所以Jockey从根本上无法阻止恶意程序破坏Jockey。我们将在第1.3节详细讨论这些问题。

效率是Jockey的次要目标。Jockey只打算在测试和调试期间使用。因此，只要Jockey的减速不会对程序的行为产生质的改变，开发者应该可以忍受一些性能的降低。[[1]](#footnote-1)我们在第5节评估了Jockey的开销。它显示性能开销最多为30%，更多的时候接近于零--完全在我们的容忍限度之内。

## 概述

我们在Linux上用C++实现了Jockey。Jockey的核心是libjockey.so，一个x86共享对象文件。图1显示了一个简单的程序，它从/dev/random，Linux的随机数设备中读取。图2显示了Jockey最基本的用法。用Jockey运行一个程序不需要改变源代码或可执行文件。只需在启动时加载libjockey.so，就能让Jockey控制进程。在这个例子中，Jockey 拦截了从 getc 所做的读取系统调用。它在记录阶段记录读取的值。当重放时，它从日志中读取值，而不实际从随机设备中读取。Jockey也可以以几种不同的方式被调用，如图所示。

|  |
| --- |
| // **test.c** int main() {  FILE \*f = fopen("/dev/random", "r");  printf("%x\n", getc(f)); }。 |

**图1：***一个简单的程序test.c，具有非确定性行为。*

|  |
| --- |
| ％cc test.c  % **LD\_PRELOAD=libjockey.so** ％**LD\_PRELOAD=libjockey.so** ％**LD\_PRELOAD=libjockey.so**  **JOCKEYRC="replay=0"** ./a.out # *录音。*  38  % **LD\_PRELOAD=libjockey.so** ％**LD\_PRELOAD=libjockey.so** ％**LD\_PRELOAD=libjockey.so**  **JOCKEYRC="replay=1"** ./a.out #*重播*38号文件 |

**图2：***Jockey的最基本用法。程序a.out虽然是从/dev/random读取，但输出的数字是一样的。设置LDPRELOAD会使动态链接器在其他对象文件之前加载libjockey.so。JOCKEYRC环境变量向libjockey.so传递参数。*

图3.

## 局限性

Jockey的大部分限制来自于它作为目标进程的一部分运行的事实。最严重的问题是，Jockey可能会被一个严重错误或恶意的目标程序破坏--例如，如果它愿意，目标程序可以破坏Jockey内部使用的堆内存。然而，Jockey也试图通过尽可能地隔离Jockey和目标程序所使用的资源来避免这样的问题，这在第3.2节中讨论过。

Jockey不能捕获由外部实体引起的事件。例如，在多线程程序中发生的内存访问竞赛不能被重放，因为墨核上下文切换不能被Jockey捕获。出于这个原因，Jockey不支持多线程程序。[[2]](#footnote-2)同样的，Jockey也不支持任何通过共享内存与其他进程或操作系统交互的程序或API；一个例子包括uDAPL[2]，用于内存映射的网络I/O。

最后，ioctl命令需要单独处理。Jockey只提供对常见ioctl命令的支持。

需要增加对其他晦涩的ioctl命令的支持。

|  |  |
| --- | --- |
| ％**骑师 --replay=0** ./a.out。 | #*记录* |
| a9  ％**骑师 --replay=1** ./a.out。 | #*重播* |
| a9  % **LD\_PRELOAD=libjockey.so \** ./a.out **--jockey=replay=0。** | #*记录* |
| 82  % **LD\_PRELOAD=libjockey.so \** ./a.out **--jockey=replay=1。** | #*重播* |
| 82  % cc test.c **-ljockey**  % ./a.out **--jockey=replay=0。** | #*记录* |
| c1  % ./a.out **--jockey=replay=1** c1。 | #*重播* |

1

2

3

4

6

7

8

9

10

11

13

14

15

16

17

**图3：***在Jockey控制下运行程序的其他方式。第1行到第4行显示了如何从jockey前端启动测试程序，jockey只需设置环境变量LDPRELOAD, JOCKYRC和执行目标程序。第6行到第11行显示了另一种方法，它传递了一个命令行参数--jockey=，这个参数会在libjockey.so加载时被解析。为了使这个方法有效，目标程序必须被设计成忽略以 --jockey= 开头的命令行字符串。 Jockey 也可以被手动链接到目标程序，而不是通过 LDPRELOAD。第13到17行显示了如何做到这一点。*

通过扩展 Jockey 来实现。Jockey提供了来自目标程序的支持，如第4节所述。

# 相关工作

长期以来，执行记录/重放一直被提倡为一种有效的调试方法[9，27，26]。

## 顺序程序的记录/重放

Bugnet[35]是最早的确定性记录/重放工具之一。它通过记录外部事件，如I/O，并定期对所有进程进行检查点（假设时钟松散同步）来重放并行程序。然而，Bugnet只支持一种特殊的API，不像Jockey那样支持通用的UNIX程序。Flashback[31]是沿着这条路线进行的最新工作。它提供了与Jockey几乎相同的功能--记录和重放系统调用以及基于fork的检查点，但是Flashback是作为内核模块提供的。因此，Flashback在使用上不如Jockey简单和安全。

在一个稍微不同的方法中，一些系统记录并重放单个内存访问[23，20，30，10]。与Jockey或Bugnet等基于事件的方法相比，它们有几个优势。首先，它们中的一些可以实现*反向执行*--实际上是将CPU指令向后移[23，10]。它们也更通用，因为它们不需要深入了解系统调用的语义和其他与环境的交互。然而，它们需要大量的程序修改，并且有很大的日志开销。Netzer等人已经提出了智能算法，用于确定需要跟踪的最小负载/存储指令集[20，30]，但这些系统对于一个CPU密集型程序来说，仍然需要1.5MB/秒的日志生成[20]。相比之下，Jockey只需要几百字节/秒，正如我们在第5节中所显示的那样。

## 使用虚拟机录制/重播

Revirt是一个记录和重放整个操作系统的虚拟机[4]。基于用户模式的Linux[3]，它可以记录和重放低级中断。Revirt已经被证明对网络入侵检测和诊断内核bug很有用[8]。其他一些论文也提出了使用虚拟机的分布式系统仿真[5，21]。虽然这些系统功能强大，但使用起来也很麻烦--例如，人们需要为每个虚拟机创建一个单独的文件系统树。当人们只对调试用户空间的程序感兴趣时，它们就显得矫枉过正了。Jockey比这些系统更简单，更容易使用。

## 对 平行 分布式程序 进行记录/重放。

确定性记录/重放在并行和分布式环境中最为有效[26]。事实上，最早的两个工具，Bugnet[35]和Pan & Linton[23]都明确针对这种环境。从那以后，人们对共享内存并行程序[17]和消息传递程序[18，19]都提出了许多理论上的改进。Jockey还不支持分布式程序的重放--它只能记录和重放系统内的单个进程。正如在5.2节中所讨论的，我们还不相信全分布式系统的记录和重放是值得的。这也许是因为我们的主要目标（FAB）本质上是一个只包含几十个节点的"小"系统。当我们面对一个真正大规模的系统时，例如，在模拟广域网络时，我们可能会改变主意。

# 实施赛马会

Jockey在程序启动时执行以下任务。

1. 对于 libc 中的每一个具有定时或上下文相关效果的系统调用，Jockey 都会重写它的前几条指令并拦截对它的调用。Jockey目前拦截了78个Linux系统调用，包括gettimeofday、recvfrom和select。Jockey在录制过程中记录这些调用所产生的值，并重播日志而不实际执行这些调用。
2. Jockey对具有非确定性影响的CPU指令也是如此。我们目前只对rdtsc打了补丁，这个x86指令用于读取CPU的时间戳计数器（周期计数器）。例如，它在libc中被用作伪随机数发生器。
3. Jockey在将控制权返回到目标程序之前检查点进程状态。在"重播"模式下，Jockey只是简单地加载检查点。检查点是需要的，以确保目标程序在记录和重放过程中看到相同的环境变量和命令行参数集。我们将在第3.3节详细讨论检查点。
4. Jockey 将控制权转移到目标程序。目标程序的运行就像Jockey不存在一样。只有当目标程序执行一个系统调用或一个非确定的CPU指令时，Jockey才会变得活跃。

下一节将更详细地描述前两个步骤。第3.2节讨论了Jockey为避免不必要的干扰而将自己与目标程序隔离的努力。第3.3节描述了Jockey的检查点功能(步骤(3))，以及我们必须克服的挑战。

## 指令补丁

图4显示了gettimeofday系统调用的记录和回放方式。(a)显示了libc.so中gettimeofday函数的前几条指令。当赛马会启动时，它会在函数的前5个字节中写入一条jmp指令，如（b）所示。如果第5个字节在另一条指令的中间，就像gettimeofday的情况一样，Jockey会覆盖到下一条指令的边界（如果需要的话，用nop填充内存）。在(c)中，Jockey也会将函数原来的前5个字节(gettimeofday为6个字节)复制到一个新分配的内存区域，以便在必要时让Jockey运行旧的实现。(d)显示了gettimeofday的新实现的入口点。Jockey动态地生成这段代码，这样它就可以在一个单独的堆栈上执行，避免破坏目标内存（见3.2节）。最后，（e）显示了newgettimeofday，Jockey的gettimeofday的实现。当记录时，newgettimeofday调用原始的gettimeofday (c)并记录返回的值。在记录时，它只是简单地从日志中提供值，而不实际执行系统调用。

也许有人会问，为什么libjockey.so不直接提供一个新的同名系统调用的实现。- LDPRELOAD经常被用来做这些事情。原因是这样做会遗漏libc或ld.so内部的系统调用，例如，一个读取的调用会被getc调用。这些内部调用是由静态链接器（ld）预先解析的，不能通过在LDPRELOAD中重新定义来覆盖。

对于任务(2)，Jockey重写所有在目标进程中发现的违规CPU指令。这有两种方式，*慢速*模式和*缓存*模式。在慢速模式下，Jockey首先读取特殊文件/proc*/N*/maps (*N*是目标进程ID)，它显示了目标进程的虚拟内存映射。然后它读取每个映射共享对象的ELF头，发现文本部分的位置，并扫描每个文本部分。Jockey在文本部分中找到非确定性的CPU指令（如果有的话），并对它们进行修补。Jockey还拦截对mmap的调用，并做同样的事情。

在步骤(1)和(2)中，Jockey需要对CPU指令进行解析，鉴于x86复杂的指令编码，这并不是一个微不足道的任务。它使用基于pidgin表的解析器来处理常见的指令和操作数，并参考*libdisasm*[11]，一个开源的x86反汇编库来处理不常见的情况。一些将操作码/操作数映射到指令长度的表格让我们快速解析了80%以上的指令出现。

然而，即使使用这种技术，在1.5GHz的Pentium-M处理器上，解析一个典型的Linux程序中的所有CPU指令也需要大约350毫秒，这对一些用户来说可能太慢了。为了缩短启动延迟，Jockey还采用了缓存模式的指令补丁。在这里，在完成慢速模式后，Jockey会将每个共享的非确定性指令的位置写进

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | | !!! | |  | |  | | --- | | !!! | |
| |  |  |  | | --- | --- | --- | | . |  | / | |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | "# |  | $% |  |  | & | |  | $ " | & |  |  |  | |  | $ $ | $ ’ $ | ( | ) |  | |  | ’ $\*+ | ) |  |  |  | | , | & |  | ) |  |  | |  | $ $ | $ ’ & | ,) |  |  | | ,  , | " ( | -$ ) |  |  |  | | |
| |  |  | | --- | --- | | $  $ | "  "# | |

**图4：***录制和重放gettimeofday。*

对象在文件˜/.jockey-sig中。当Jockey下一次启动时，它只是读取˜/.jockey-sig而不扫描进程的虚拟内存，除非对象文件的时间戳发生了变化。

Jockey的指令解析/重写模块也可以被目标程序或程序员定义的对象文件用来监视目标程序的执行。我们将在第4节详细讨论这个功能。

## 隔离资源使用

Jockey和目标应用程序作为同一个进程的一部分运行。它们共享资源，包括内存和文件描述符。Jockey必须隔离这些资源的使用，以防止Jockey不必要地改变目标程序的行为，并尽量减少行为不当的目标程序破坏Jockey的机会。我们讨论Jockey对堆、栈和文件描述符这三种资源的处理。

**堆。**Jockey不能使用标准的libc函数，如malloc或sbrk，来管理自己的数据。这样做增加了行为不端的目标程序破坏Jockey数据的可能性。而且，它在记录和重放过程中改变了目标进程的内存布局。它将变得无法正确重放无效的内存访问，例如访问free'ed内存，这是比较常见的编程错误之一。

相反，Jockey将其所有的内部数据存储在一个虚拟地址(0x63000000)的mmaped区域中，这个区域不太可能被目标程序意外使用。Jockey使用它自己的内部类似malloc的库来将内存分割成各个数据结构，并在其上建立一个自定义的C++ STL内存分配器。因此，Jockey代码可以完全访问STL功能，包括地图和动态向量。这种设计大大简化了Jockey的开发工作。

一个限制是，Jockey不能调用内部调用malloc或free的libc函数，例如高级I/O函数（FILE，std::fstream）和DNS解析器（gethostbyname）。例如高级I/O函数(FILE, std::fstream)和DNS解析器(gethostbyname)。

**堆栈。**Jockey也隔离了堆栈的使用。这对于正确地重放访问堆栈指针以外的数据的程序是必要的（例如，访问一个负索引的堆栈数组）。图4，步骤(d)显示了如何做到这一点。在它截获对trampoline的调用后的前几条指令中，Jockey将栈指针保存到一个内部静态变量中，将栈指针寄存器切换到其内部内存中，将gettimeofday（一个4字节的指针）的参数从旧栈复制到新栈，并调用newgettimeofday。一旦新的实现返回，它就会恢复栈指针。这样，即使对于一个bug程序来说，也可以进行确定性重放，因为目标的堆栈没有用到原来的堆栈指针之上。

**文件描述符。**Jockey必须每隔一段时间就对自己的文件进行访问，例如，当打开一个事件日志文件，显示一条信息，或采取检查点。因为Jockey和目标进程共享同一个文件描述符表，所以Jockey必须确保它的文件操作不会改变目标进程看到的描述符分配方案。为此，Jockey打开的每个文件描述符都会被移动到一个不可能被应用程序使用的范围（430及以上）。这是通过在打开后立即执行dup2来实现的。

Gdb带来了另一个问题。当启动目标进程时，gdb由于某种原因，除了通常的stdin、stdout和stderr之外，还会打开一些额外的文件描述符。因此，如果在普通shell下记录执行情况，然后在gdb下重放，目标进程打开的文件将被分配不同的描述符，这使得重放出现分歧。我们通过让Jockey在启动时打开描述符0到9的虚拟文件来解决这个问题，然后再将控制权返回给目标程序（它将描述符留在--------）。

|  |
| --- |
| % **jockey --checkpointfrequency=30 ↪LoHan\_4339↩↪LoHan\_4339↩=30**  **--retain checkpoints=5 ．**  **--** httpd -X...以后...。  % **jockey --restore=log/checkpoint-3** httpd。 |

**图5：***每30秒自动进行httpd（Apache）检查点。选项-X会在前台运行Apache，而选项--retaincheckpoints=5则只保留最后五个检查点。--retaincheckpoints=5选项只保留最后五个检查点。最后一行重播第三个检查点的httpd。*

继承自父进程，不受影响）。)假设gdb在启动目标程序时最多打开10个描述符，我们可以确保目标程序在记录和重放时有相同的一组文件被打开。当目标程序被启动时，其继承的文件描述符数量超过标准数量（3个）时，这种技术对其他情况也有效。

## 检查站

Jockey允许用户自动检查点进程状态。图5显示了一个每30秒对httpd（Apache）进行一次检查点的例子。

按照libckpt[25]和flashback[31]所开创的技术，Jockey通过分叉进程和转储子进程的状态来检查点，同时让父进程继续运行。Jockey读取特殊文件/proc/*N*/maps(*N*是进程ID)来获取进程的虚拟内存映射，并且只转储那些被映射的读写部分。要恢复一个检查点，对于检查点文件中记录的每一个部分，如果该内存区域已经被占用，Jockey就会解除映射，并从检查点文件中恢复内容，或者如果该部分被标记为只读，则重新映射文件。

我们面临两个特殊的检查点问题，都与动态链接有关。

**防止ld.so脑损伤** 在还原过程中的一个挑战是，Jockey需要覆盖可能被还原代码本身使用的内存。如果天真地进行还原，Jockey会崩溃。这里，需要照顾到两类内存区域。Jockey的内部堆（3.2节）和动态链接器使用的堆。例如，Jockey必须执行read系统调用来读取检查点内容。如果这个读取的调用恰好是目标应用程序或Jockey有史以来的第一次调用，那么动态链接器就会被调用来解析"读取"这个符号，这就涉及到修改链接器的内部数据结构。

Jockey通过排除它的检查点来处理它的内部堆，但是动态链接器带来了一个特殊的挑战--我们无法先验地知道动态链接器所使用的内存在哪里（链接器对它的堆内存进行匿名mmap；所有匿名mmap的部分在Jockey看来都是一样的）。我们处理这个问题的方法是，在Jockey恢复任何检查点之前，急切地链接所有Jockey使用的libc函数，对open和read等函数进行虚拟调用。

**Exec shield** Exec-shield 是一些 Linux 内核（例如 Red Hat、Fedora Core）中的一个设施，用于阻挡缓冲区溢出攻击[15]。它提供的一个功能是随机化共享对象文件的加载地址。这个特性打破了Jockey，因为Jockey需要保留特定于进程的内存布局的数据结构。我们还没有解决这个问题的办法，目前只是要求通过做以下操作来禁止这个功能。

echo 0 >/proc/sys/kernel/exec-shield

## 处理信号

信号的处理方式与[32]类似。每个信号的传递首先被Jockey拦截。Jockey的信号处理程序只是记录信号的参数（信号号和寄存器上下文），然后完成。在系统调用或rdtsc CPU指令的Jockey处理程序结束时，Jockey检查过去是否有信号被截获。如果是的话，它就会记录这个信号（这样就可以重放），并调用目标定义的信号处理程序。这样，Jockey将异步信号转换为同步上调信号，只有在系统调用后才会立即发生。

当目标程序长时间运行而不发出系统调用（或执行非确定性的CPU指令），并在此期间接收信号时，这种技术可能会扭曲程序行为。然而，赛马会的主要目标，即面向I/O的程序，通常不会受到这个问题的影响。

## 减少记录费用

Jockey根据系统调用的类型，采用了两种不同类型的日志策略，以减少空间开销。

* 对于常规文件或目录的请求，Jockey会执行"撤销"记录。也就是说，对于更新文件的系统调用，Jockey会记录足够的信息来恢复修改*前*的内容。例如，当一个写系统调用覆盖了一个文件的中间部分，Jockey会记录该部分的偏移量和旧内容。或者，当写附加到文件的末尾时，Jockey只是记录文件的旧大小。对于只读的系统调用（如read），它只是直接从文件中读取。Discount检查系统[12]中也采用了类似的方法。
* 对于所有其他类型的事件--对套接字、管道、fifos、设备或select、gettimeofday或rdtsc的I/O--赛马会执行"重做"记录。Jockey在记录期间记录事件产生的值。在重播期间，Jockey只是从日志中读取值，而不执行实际的系统调用。因此，更新文件的系统调用在重放过程中成为无操作。

读和写等系统调用可以对两种类型的文件进行操作。我们拦截对创建文件描述符的函数的调用--例如，open、socket和accept--记住每个描述符的类型，并根据描述符类型进行调度。从父进程继承的文件描述符（如stdin、stdout、stderr）会被重做记录。

各种研究表明，大多数对普通文件的I/O都是读，而大部分的写流量实际上是追加[22，1，33]。对于这些常见的情况，我们的设计允许Jockey只记录请求的类型和偏移量，而不是实际内容。因此，它极大地降低了文件I/O系统调用的日志开销[12]。

基于撤消的日志记录的缺点是，用户不能在记录和重放之间修改目标程序所接触的文件。到目前为止，我们还没有发现这是一个很大的负担。

## 处理内存映射的I/O

Jockey目前需要人工处理通过内存映射文件完成的I/O。为了重放这样的I/O，程序员必须对程序进行仪表化处理，并在每次I/O进行时通知Jockey。图6显示了一个例子，jockey\_prepare\_mmaped\_write记录了指定区域的原始映像，这样Jockey就可以在重放之前恢复映像。

这种方法公认是容易出错的。作为一种替代方法，我们可以通过写保护来应用"页面差异化"。

|  |
| --- |
| int fd = open("blah", O\_RDWR); char \*base = mmap(NULL, size,  PROT\_READ|PROT\_WRITE, MAP\_SHARED,  fd, 0); ...  char \*p = base + offset char \*str = "Hello, World";  jockey\_prepare\_mmaped\_write(fd, offset, p, strlen(str) + 1)。  strcpy(p, str)。 |

1 2

3

4

5

6

7 8

9 10

**图6：***内存映射I/O的一个例子。在程序修改内存映射区域之前，它必须通知赛马会（第6行）文件描述符（fd）、文件偏移量（offset）、mmap指针（p）和修改的大小。*

mmaped页面，并捕捉对它们的访问[6]。实现这种方法是未来的工作。

# 控制骑师

Jockey作为目标程序的一部分的优点之一是它很容易允许用户或目标程序自定义Jockey。本节列出了一些可以改变Jockey行为的旋钮。

**jockeysetforktracemode。**默认情况下，在fork时，Jockey只继续追踪父程序，而不追踪子程序。这个函数，当被目标程序调用时，会改变这个行为--是只追踪父程序，还是追踪子程序，或者在fork后同时追踪父程序和子程序。例如，这个函数可以用于守护进程类型的程序，这些程序通过fork来脱离父进程。

**jockeycheckpoint(*path*)。**这个函数可以被目标程序手动调用，以获取一个检查点。

## jockeyredirectcalls(*name*, *newproc*,

***argsize*).该函数用于在调用函数名时将控制权转移到newproc。**该函数用于在调用函数*名时*将控制权转移到*newproc*。参数*argsize*是函数*名的*在栈参数的大小。例如，这个功能可以用来为一个只有目标应用程序使用的不知名的ioctl命令提供记录/重放功能。函数jockeyinterposecalls(*name*, *newproc*,

*argsize*)类似，但它在*newproc*完成执行后将控制权转回原函数。

|  |
| --- |
| **// testprogram.c** void bar(int i) {  ...做一些复杂的事情...  } void main() { int i;  for (i = 0; i < 100000; i++) bar(i);  } |
| **// checker.c**  #include <jockey/jockey.h> void bar\_checker(int i) { if (i == 95999)  jockey\_breakpoint()。  } void init() { jockey\_interpose\_calls("bar", bar\_checker, 4); }。 |

**图7：***上层列表，testprogram.c是一个简单的程序，它多次调用函数bar。下层列表checker.c是一个用户定义的检查器，它在bar被调用95999次后调用断点。当检查器被加载到内存中时，函数init被Jockey调用。jockeybreakpoint是一个在Jockey中定义的函数，它什么都不做。*

|  |
| --- |
| % cc -c checker.c  % gdb测试程序  (gdb) set env LD\_PRELOAD=libjockey.so。  (gdb) b jockey\_breakpoint (gdb) run --jockey=replay=1;\ checker=checker.o。 |

**图8：***运行用户自定义检查程序。checker.c是用户自定义检查程序的名称。每次bar执行时，都会调用函数barchecker。它在不影响目标执行的情况下检查程序的状态，并在指定点调用断点。*

**用户定义的不变性检查器。**Jockey允许在重放过程中执行任意对象文件。图7显示了一个例子。这里，我们假设我们在Jockey下运行上层程序，发现当i == 95999时，过程bar表现异常。我们可以通过在gdb中对bar设置一个断点，并等待它达到95999次来诊断这种情况，但Jockey提供了一个更快捷的选择，如图8所示。

这个功能的实现是复杂的，因为我们不能依靠动态链接器将对象链接到目标程序中--这样做会改变目标程序的堆结构（3.3节）。Jockey使用静态链接器(ld)来代替解析检查器对象中的符号。Jockey首先选择一个不可能被目标应用程序使用的虚拟地址（0x62000000）。然后，它通过为每个共享对象调用nm来发现进程中所有公共符号的地址（包括那些由Jockey导出的符号）。然后这个信息会被传递给ld，它生成一个包含所有符号解析的二进制文件。然后Jockey将文件内容读入内存并执行。

# 评价

本节报告了Jockey的性能和空间开销，并讨论了使用Jockey调试实际程序的经验。

## 性能和日志空间开销

我们在下面列出的各种程序上试用了Jockey。评估是在一台运行Fedora Core 3 (kernel 2.6.9)的Linux计算机上进行的，该计算机使用1.5GHz Pentium-M CPU，512MB内存，7200 rpm ATA硬盘，以及1Gb/s的

以太网接口。

**g++** g++ 3.4.2编译一个使用STL映射的小C++程序。开销是前端（g++）、编译器本身（cc1plus）和链接器（ld）的总和。

**xclock** 一个X窗口系统的数字时钟，每秒钟更新一次屏幕。

**Emac**s Emacs 21.3运行程序开发会话，涉及主动打字、文件读取和保存。

**httpd** Apache 2.0.52(单进程，无分叉)，服务于100000个HTTP GET请求，从httperf[16]获取一个0.5KB的文件。

**FAB** 一个四块砖的FAB集群[28]，服务于80000个随机的1KB I/O请求。FAB是一个分布式磁盘阵列系统，运行在一个PC集群上。

g++是一个短时运行、CPU密集型程序的例子，这不是Jockey的主要目标。这个例子仍然表明，与涉及内存访问日志记录[20]的方法相比，Jockey的日志空间开销非常低，因为后者每秒可能消耗高达几兆字节的日志记录。对于gcc来说，大部分的性能开销是由于在执行开始时发生的进程检查点（第3节）。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **名称** |  | **运行时间** |  | **对数大小** | | | **本地人** | **记录** | **重播** | **#字节** | **#记录** | | g++ | 1.33 | 1.51 | 1.49 | 73KB | 80 | | 时钟 | 不适用 | 180 | 0.4 | 80KB | 4639 | | Emacs | 不适用 | 210 | 5.81 | 1.4MB | 20769 | | httpd | 16.7 | 17.5 | 9.5 | 2.0MB | 140180 | | FAB | 33.7 | 44.1 | 31.1 | 34MB | 887000 |   **表1：***Jockey的性能和空间开销。运行时间的单位是秒。"Native"是没有使用Jockey的运行时间。"录制"和"重播"分别显示录制和重播时的运行时间。* |

Xclock和emacs是交互式应用的例子。Jockey表现出合理的日志空间开销。Jockey能够极快地重放它们的执行，因为它不需要在重放过程中等待超时或用户输入。这就转化为更有效的调试活动。

Apache和FAB是网络服务器的例子。FAB代表了Jockey最坏的情况。FAB不仅执行大量的网络I/O，它也会重复地覆写现有的文件，导致大量的日志流量（第3.5节）。相比之下，Apache的开销要小得多，因为它只对index.html进行只读访问，并对日志文件进行追加。

## 经验

Jockey经常被用于FAB的开发。它在诊断那些快速表现出来的bug方面，例如，在客户端的第一个I/O请求期间，出奇的有效。Jockey减少了调试的"周转"时间，因为它允许开发人员重放一个进程，而不是重新启动整个集群。这使得开发人员的工作效率更高。

在调试压力测试或回归测试期间发生的非确定性错误时，Jockey是最有用的。在我们拥有Jockey之前，我们不得不多次重启集群，每次都要使用一组稍微不同的"printf"语句，希望最终能够重现并捕捉到错误。Jockey允许开发者按照自己的意愿频繁地可靠地重现错误。然而，即使有了Jockey，诊断这样的bug仍然很困难。错误的真正"原因"往往发生在几百次请求之后才表现出来，往往是在不同的过程中。程序员不得不反复重放多个进程的执行情况来定位原因。人们曾主张用确定性的分布式系统重放来诊断这类bug（第2节），但我们目前怀疑这是否值得。调试一个长期运行的分布式系统，无论如何都需要对程序有深入的了解，而Jockey提供的单进程独立重放似乎可以实现大部分的好处。

有一些赛马会的功能在理论上听起来很有用，但在实践中却被证明不是那么回事。使用定期自动检查点的"时间旅行"概念就是这样的一个想法（第3.3节）。它是一个强大的，但难以使用的工具。如果没有适当的调试器支持[8]，开发者每次想要切换到不同的检查点时，都需要重新启动调试器。这种手动工作很快就会让开发人员筋疲力尽。另一个功能是用户定义的不变性检查（第4节）。问题有两个方面。首先是缺乏适当的调试器支持--每次想调试时都要编写和编译一个程序，这很麻烦。其次是检查器能做的事情相当有限--例如，它不能拦截函数执行中间的调用，也不能检查调用链中的局部变量值。

# 结论

本文介绍了Jockey，一个用于确定性记录/重放调试的用户空间库。Jockey作为目标程序的一部分运行，并拦截对非确定性系统调用和CPU指令的调用。它在记录过程中记录这些操作的效果，并在重放过程中从日志中重放它们。它的性能和日志空间开销很小。Jockey已被广泛用于开发FAB，一个运行在PC集群上的分布式磁盘阵列系统。

# 参考文献

1. Mary Baker, John H. Hartman, Michael D. Kupfer, Ken Shirriff, and John K. Ousterhout.分布式文件系统的测量。In [*13th Symp.Principles (SOSP)*,](http://portal.acm.org/toc.cfm?id=122132&coll=portal) pages 198-212, Pacific Grove, CA, USA, October 1991.
2. DAT合作。用户级直接访问传输API(uDAPL)，2004年。http://www.datcollaborative.org/udapl.html。
3. Jeff Dike等人。用户模式Linux主页。[http://user-mode-linux% -.sourceforge.net/，](http://user-mode-linux.sourceforge.net/)2005。
4. George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen.Revirt.Murtaza Basrai, and Peter M. Chen:通过虚拟机日志和重放实现入侵分析.In [*5th Symp.Design and Impl.(OSDI)*,](http://www.usenix.org/publications/library/proceedings/osdi02) Boston, MA, USA, December 2002.
5. Timothy L. Harris.[Dependable software needs pervasive debugging.](http://www.cl.cam.ac.uk/users/tlh20/papers/tim-harris-sigops.ps.gz)在[*第10届ACM SIGOPS欧洲研讨会*](http://www.diku.dk/ew2002/)上[，](http://www.diku.dk/ew2002/)法国Saint Emilion，2002年9月。
6. Antony L. Hosking, Eric W. Brown, and J. Eliot B. Moss.更新日志的持久性编程语言。A comparative performance evaluation.In *19th Int.Conf. on Very Large Data Bases (VLDB)*, pages 429-440, Dublin, Ireland, August 1993.
7. Joel Huselius.[调试并行系统。最先进的报告。](http://www.mrtc.mdh.se/php/publ_show.php3?id=0434)技术报告63，马拉达伦大学CSE系，2002年9月。
8. Samuel T. King, George W. Dunlap, and Peter M. Chen.调试操作系统与时间旅行的虚拟机。In [*USENIX Annual Tech.Conf.*](http://www.usenix.org/publications/library/proceedings/usenix05)美国加州阿纳海姆，2005年4月。
9. 林立忠.A survey of data breakpoint and reverse execution.SUNY Stony Brook RPE报告，[http://www.ecsl% -.cs.sunysb.edu/tr/rpe12.ps.gz，](http://www.ecsl.cs.sunysb.edu/tr/rpe12.ps.gz)2001年9月。
10. 比尔-刘易斯Debugging backwards in time.在[*第五届自动和算法调试（AADEBUG）研讨会上*，](http://aadebug2003.elis.rug.ac.be/)比利时根特，2003年9月。
11. libdisasm.Libdisasm：x86反汇编库，2004年。Libdisasm：x86反汇编库，2004年。

http://bastard.sourceforge.net/libdisasm.html。

1. David E. Lowell and Peter M. Chen.折扣检查。透明的、低开销的一般应用回收。技术报告CSE-TR-410-99，密歇根大学，1998年11月。
2. Charles E. McDowell和David P. Helmbold.调试并发程序。*ACM Computing Surveys*, 21(4):593-622, December 1989.
3. Michael S. Meier, Kevan L. Miller, Donald P. Pazel, Josyula R. Rao, and James R. Russell.构建分布式调试器的经验。In *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pages 70-79, Philadelphia, PA, USA, May 1996.
4. Ingo Molner.Exec shield，新的Linux安全功能。[http% -://people.redhat.com/mingo/exec-shield/ANNOUNCE-exec-shield，](http://people.redhat.com/mingo/exec-shield/ANNOUNCE-exec-shield)2004。
5. [http://www.hpl.hp.com/personal/David Mosberger/httperf.html，](http://www.hpl.hp.com/personal/David_Mosberger/httperf.html)2001。
6. Robert H. B. Netzer.调试共享内存并行程序的最佳跟踪和重放。In [*ACM workshop on parallel and distributed debugging*,](http://portal.acm.org/toc.cfm?id=174266&coll=portal) San Diego, CA, USA, May 1993.
7. Robert H. B. Netzer 和 Barton P. Miller.调试消息传递并行程序的最佳跟踪和重放。In *Supercomputing*, Mineapolis, MN, USA, November 1992.
8. Robert H. B. Netzer, Sairam Subramanian, and Jian Xu.基于关键路径的消息记录，用于消息传递程序的增量重放。In *14th Int.Conf. on Dist.Sys.(ICDCS)*，页面404-413，波兰波兹南，1994年6月。
9. Robert H. B. Netzer 和 Mark H. Weaver.Optimal tracing and incrementar reexecution for debugging long-running programs.SIGPLAN Conference on Programming Language Design and Implementation (PLDI) , Orlando, FL, USA, June 1994.In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, USA, June 1994.也可作为布朗大学技术报告CS94-11。
10. Oliver Oppitz。A particular bug trap: Execution replay using virtual machines.在[*第五届自动和算法调试（AADEBUG）研讨会上*，](http://aadebug2003.elis.rug.ac.be/)比利时根特，2003年9月。
11. John K. Ousterhout、Herv Da Costa、David Harrison、John A. Kunze、Michael D. Kupfer和James G. 等。

Thompson.A trace-driven analysis of the UNIX 4.2 BSD file system.In [*10th Symp.Principles (SOSP)*,](http://portal.acm.org/toc.cfm?id=323647&coll=portal) pages 15-24, Orcas Island, WA, USA, December 1985.

1. Douglas Z. Pan 和 Mark A. Linton.[支持并行程序的反向执行。](http://portal.acm.org/citation.cfm?id=69227)In [*ACM workshop on parallel and distributed debugging*,](http://portal.acm.org/toc.cfm?id=68210&coll=portal) Madison, WI, USA, May 1988.
2. Vern Paxson。A survey of support for implementing debuggers. [http://citeseer.ist.psu.edu/paxson90survey.html,](http://citeseer.ist.psu.edu/paxson90survey.html) 1990.
3. James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li.[Libckpt:UNIX下的透明检查点.](http://www.cs.utk.edu/~plank/plank/papers/USENIX-95W.html)In *USENIX Winter Tech.Conf., New Orleans, LA, USA, January.*, New Orleans, LA, USA, January 1995.
4. Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmuller.Record/replay for non-determinsitic program executions.*Comm.of the ACM (CACM)*, 46(9), September 2003.
5. Michiel Ronsse, Koen De Bosschere, and Jacques Chassin de Kergommeaux.执行重放和调试。在*第四届自动和算法调试（AADEBUG）研讨会上*，德国慕尼黑，2000年8月。
6. Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence.FAB：从商品组件构建分布式企业磁盘阵列。In *11th Int.Conf.支持Prog.Lang.和Op.Sys.Lang.和Op.Sys.(ASPLOS-XI)*，波士顿，马萨诸塞州，美国，2004年10月。
7. Julian Seward 等人 Valgrind:[http://valgrind.kde.org/,](http://valgrind.kde.org/) 2004.
8. Michael W.Shapiro.RDB: A system for incremental replay debugging.RDB: A system for incremental replay debugging.Master's thesis, Dept of.Computer Science, Brown University, 1997.
9. Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou.闪回。用于软件调试的回滚和确定性重放的轻量级扩展。In [*USENIX Annual Tech.Conf., Boston, MA, USA, June.*](http://www.usenix.org/publications/library/proceedings/usenix04)美国马萨诸塞州波士顿，2004年6月。
10. Daniel Stodolsky, Brian N. Bershad, and J. Bradley Chen.[操作系统内核中的快速中断优先权管理》，Usenix微内核研讨会，第105-110页，1993年9月。](http://www.cs.washington.edu/homes/bershad/Papers/OptSynch.ps)*Usenix Workshop on Microkernels*, pages 105-110, September 1993.
11. Werner Vogels.文件系统在Windows NT 4.0中的使用。In [*17th Symp.Principles (SOSP)*,](http://portal.acm.org/toc.cfm?id=319151&coll=portal) pages 93-109, Kiawah Island, SC, USA, December 1999.
12. Robvon Behren, JeremyCondit, FengZhou,

George C. Necula, and Eric Brewer.[Cappriccio: Scalable threads for Internet services.](http://capriccio.cs.berkeley.edu/publications.html)In [*19th Symp.Principles (SOSP)*,](http://portal.acm.org/toc.cfm?id=945445&coll=portal) Bolton Landing, NY, USA, October 2003.

1. Larry D. Wittie.Debugging distributed C programs by real time replay.In [*ACM workshop on parallel and distributed debugging*,](http://portal.acm.org/toc.cfm?id=68210&coll=portal) pages 57-67, Madison, WI, USA, May 1988.

1. 往往需要扩展超时参数，防止其过早失效。 [↑](#footnote-ref-1)
2. Jockey确实支持用户空间线程包，比如Capriccio[34]。 [↑](#footnote-ref-2)