工程记录和回放以实现可部署性

# 摘要

以低开销记录和回放程序执行的能力使许多应 用程序成为可能，例如反向执行调试、难以重现 的测试故障的调试以及部署系统中故障的“黑盒 ”取证分析。现有的记录和回放方法通过记录整 个虚拟机（重量级）、修改操作系统内核（增加部 署和维护成本）、需要普遍的代码检测（施加显 着的性能和复杂性开销）或修改编译器来限制可 部署性和运行时系统（限制一般性）。我们研究 了是否有可能建立一个实用的记录和回放系统 来避免所有这些问题。答案是肯定的——如果 CPU 和操作系统满足某些非显而易见的约束。 幸运的是，现代 Intel CPU、Linux 内核和用 户空间框架确实满足了这些限制，尽管这只是最 近才成为现实。通过一些新颖的优化，我们的系 统 RR 使用库存硬件、编译器、运行时和操作系 统，以低开销记录和回放现实世界的低并行度工 作负载，并具有完整的用户空间实现。RR 构成 了开源反向执行调试器的基础，在实践中得到了 显着的应用。我们介绍了 RR 的设计和实现， 描述了它在各种工作负载上的性能，并确定了支 持我们的方法所需的硬件和操作系统设计的限 制。

# 1 简介

以低开销记录程序执行并精确回放的能力有 许多应用 [14, 15, 19]，并且在研究界受到 了极大的关注。它甚至已在 VMware Works tation [28]、Simics 等产品中实施 ∗ 在 Mozilla Research 的支持下进行的大部分工作。 [20]、UndoDB [1] 和 TotalView [22]。不 幸的是，由于各种原因，这些技术的部署受到 了限制。一些方法 [17, 20, 28] 需要记录 和回放整个虚拟机，这是重量级的。其他方法 [6, 14, 26, 29] 需要运行修改后的操作 系统内核，阻碍部署并增加系统的安全性和稳 定性风险。要求编译器和语言运行时更改 [ 29] 也会阻碍部署，尤其是当应用程序包含自 己的 JIT 编译器时。一些方法 [24, 30, 35] 需要尚未可用的定制硬件。许多方法 [ 1, 7, 22, 34] 需要对代码进行普遍的检测 ，这增加了复杂性和开销，特别是对于自修改 代码（通常用于多态内联缓存 [25] 和现代即 时插入中的其他实现技术）-time 编译器）。 高性能动态代码检测引擎的构建和维护成本 也很高。 我们着手构建一个系统，通过避免所有这些问 题来最大限度地提高部署能力：使用股票 Li nux 内核、编译器、语言运行时和 x86/x86- 64 CPU 记录和重放未经修改的用户空间应用 程序，具有完全用户空间实现在没有特殊权限 的情况下运行，也没有使用普遍的代码工具。 我们假设 RR 应该运行未经修改的应用程序， 它们会有我们希望忠实记录和重放的错误（包 括数据竞争），但这些应用程序不会恶意尝试破 坏记录或重放。我们结合了已知但之前未在实 际系统中演示过的技术：主要是使用 ptrace 记录和重放系统调用结果和信号，通过一次只 运行一个线程来避免非确定性数据竞争，并使 用 CPU硬件性能计数器来测量应用程序进度， 以便在正确的时刻传递异步信号和上下文切换 事件 [33]。第 2 节更详细地描述了我们的方 法。

有了它，我们发现了主要的表现—— USENIX 协会 2017 USENIX 年度技术会议 377

低并行工作负载的瓶颈是使用 ptrace 监视 系统调用引起的上下文切换。我们实施了一种 新颖的进程内系统调用拦截技术来消除这些上 下文切换，从而显着减少重要现实世界工作负 载的记录和重放开销。这种优化依赖于现代 Linux 内核特性：seccomp-bpf 选择性地抑制 某些系统调用的 ptrace 陷阱，以及 perf 上下文切换事件以检测内核中记录的线程阻塞 。第 3 节描述了这项工作，第 4 节给出了一 些性能结果，表明在重要的应用程序工作负载 上，RR 记录和重放的减速小于两倍。 我们依赖为其他目标而设计的硬件和操作系统功 能，因此 RR 的工作令人惊讶。事实上，它绕过了可 行性的边缘。特别是它不能在 ARM CPU 上实现。 第 5 节总结了 RR 的硬件和软件要求，我们希望 这些要求会影响系统设计人员。 RR 是许多开发人员日常使用的基础 一个高效的反向执行调试器的日期，适用于 复杂的应用程序，如 Samba、Fire fox、Ch romium、QEMU、LibreOffice 和 Wine。它是 免费软件，可从 https://github.com/mo zilla/rr 获得。本文做出以下研究贡献： • 我们表明，在没有普遍代码检测的情况 下，在现代库存硬件和软件上记录和重放 用户空间进程是可能且实用的。 • 我们引入了一种进程内系统调用拦截技 术，并表明它显着减少了开销。 • 我们表明，对于低并行性工作负载， RR 记录和重放开销相当低，低于具有可 比可部署性的其他方法。 • 我们确定支持我们的方法所需的硬件 和操作系统设计约束。 一份包含额外技术细节的扩展技术报告和在 RR 的开发和使用过程中“经验教训”的回顾是 可用的 [32]。 2 设计 2.1 总结 大多数低开销的记录和回放系统都依赖于观 察到 CPU 大多是确定性的。我们确定了围绕 状态和计算的边界， 记录边界内的所有非确定性来源和所有进入 边界的输入，并通过重放非确定性和输入来重 新执行边界内的计算。如果确实捕获了所有输 入和不确定性，则重放期间边界内的状态和计 算将与记录期间的状态和计算相匹配。 为了启用任意 Linux 应用程序的记录和回放 ，而不需要内核修改或虚拟机，RR 记录和回放 一组进程的用户空间执行。为了简化不变量，并 使重放尽可能忠实，重放几乎保留了用户空间执 行的每一个细节。特别是，用户空间内存和寄存 器值被准确地保留了下来，本文后面会提到一些 例外情况。这意味着记录和回放之间的 CPU 级控制流是相同的，内存布局也是如此。 虽然重放保留了用户空间状态和执行，但在重放 期间只复制了最少量的内核状态。例如，不打开 文件描述符，不安装信号处理程序，不执行文件 系统操作。相反，这些操作的记录的用户空间可 见效果以及未来相关的操作会被重放。我们确实 为每个记录的线程创建一个重播线程（不是绝对 必要的），并且我们为每个记录的地址空间创建 一个重播地址空间（即进程），以及匹配的内存映 射。 通过这种设计，我们的记录边界是用户空间和 内核之间的接口。非确定性的输入和来源主要 是系统调用的结果和异步事件的时序。 2.2 避免数据竞争 由于线程在多个内核上运行，不同线程对同一内 存位置的竞速读写访问将是不确定性的来源。 因此，我们采用通用方法 [17, 28, 1, 15] 一次只运行一个线程。RR 抢先调度这些线程， 因此上下文切换时间是必须记录的不确定性。 如果上下文切换发生在执行中的正确点，仍然可 以观察到数据竞争错误（尽管无法观察到由于弱 内存模型导致的错误）。 这种方法比替代方案 [7, 18, 34, 39, 29 ] 更简单，更易于部署，避免假设程序是无竞争 的 [14, 29]，并且对于低并行度工作负载有效 。对于具有持续高度并行性的工作负载，速度会 大幅下降；然而，即使对于可能高度并行的应用 程序，用户也经常应用 RR 来测试具有相对较 小数据集的工作负载，因此并行性有限。

2.3 系统调用 系统调用通过修改寄存器和内存将数据返回到用 户空间，这些变化必须记录下来。ptrace 系统调 用允许进程监督其他“tracee”进程和线程的执 行，并在tracee 线程进入或退出系统调用时得 到同步通知。当跟踪线程进入内核进行系统调用 时，它会被挂起并通知 RR。当 RR 选择再次运行 该线程时，系统调用将完成，再次通知 RR，使其 有机会记录系统调用结果。RR 包含大多数 Li nux 系统调用的模型，描述了他们可以修改的用 户空间内存，给定系统调用输入参数和结果。 如上所述，RR 通常通过一次只调度一个线程来 避免竞争。但是，如果内核中的系统调用阻塞， RR 必须尝试调度其他应用程序线程在阻塞系统 调用完成时运行。正在运行的线程有可能（尽管 不太可能）访问系统调用的输出缓冲区并与内核 对该缓冲区的写入竞争。为了避免这种情况，我 们将系统调用输出缓冲区重定向到每个线程的 临时“暂存内存”，否则应用程序将不会使用它。 当我们收到一个阻塞系统调用完成的 ptrace 事件时，RR 将暂存缓冲区内容复制到真正的用 户空间目标，而没有其他线程在运行，从而消除 了竞争。 图 1 说明了记录一个简单的读取系统调 用。灰色框代表内核代码。 重放过程中，当下一个要重放的事件是被拦截的 系统调用时，我们在系统调用指令的地址处设置 一个临时断点（记录在trace中）。我们使用 p trace 运行 tracee 线程，直到它遇到断点， 移除断点，将程序计数器推进到系统调用指令之 后，并应用记录的寄存器和内存更改。这种方法 很简单，并且最大限度地减少了 RR 和跟踪线 程之间的上下文切换次数。（有时它是不安全的 ，我们会退回到更复杂的机制。） 一些系统调用操作线程或地址 空格并在重播期间需要特殊处理。例如，使用 MAP FIXED 重播记录的 mmap，以确保在正 确的地址创建映射。 2.4 异步事件 我们需要支持两种异步事件：抢占式上下文切 换和信号。我们将前者视为后者的特例，通过向 正在运行的跟踪线程发送信号来强制进行上下 文切换。我们需要确保在回放期间，当节目处于 与录制期间传递信号时完全相同的状态时传递 信号。 与之前的工作 [17, 33, 10] 一样，我们使用 CPU 硬件性能计数器测量应用程序进度。理想 情况下，我们会在记录期间计算导致异步事件的 已退出指令，并且在重播程序期间，CPU 在许多 指令已退出后触发中断——但这种方法需要修改 才能在实践中工作。 2.4.1 非确定性性能计数器 我们要求每次执行给定的用户空间指令序列都 会改变计数器值，该值仅取决于指令序列，而不 是用户空间不可见的系统状态（例如缓存的内容 ，页面的状态表或推测的 CPU 状态）。这个属性 （通常被描述为“确定性”[40]）在实践中并不适 用于大多数 CPU 性能计数器 [17, 40]。例如 ，它不适用于任何已知 x86 CPU 模型上的任何 “指令退休”计数器（例如，因为触发页面错误的 指令被重新启动并计数两次）。 幸运的是，现代 Intel CPU 有一个确定的性 能计数器：“退休条件分支”（“RCB”），所以我们 使用它。我们不能仅在录制期间计算 RCB 的数 量，并在重放期间执行该数量的 RCB 后传递信 号，因为 RCB 计数并不能唯一地确定传递信号 的执行点。因此，我们将 RCB 计数与通用寄存 器（包括程序计数器）的完整状态配对以识别执 行点。 通常，它仍然不能唯一标识执行点（例如，考虑 无限循环标签：inc [global var]; jmp la bel;）。但是，在实践中，我们发现它可以可靠地 工作。没有中间条件分支返回到同一条指令的代 码必须非常少见，并且只有在这样的指令上发生 异步事件时才对 RR 很重要——在这种情况下， 重放可能会发散并失败。

2.4.2 延迟触发 另一个主要问题是，尽管 CPU 可以编程为在观 察到指定数量的性能事件后触发中断，但中断不 会立即触发。在实践中，我们经常观察到它在数 十条指令退出后触发。为了弥补这一点，在重放 期间，我们将中断编程为比我们预期的实际 R CB 计数更早触发一些事件。然后我们在程序计 数器值处为我们试图达到的状态设置一个临时 断点，并重复运行到断点，直到 RCB 计数和通 用寄存器值匹配它们记录的值。 2.5 共享内存 通过一次只调度一个线程，只要共享内存仅由跟 踪线程写入，RR 就避免了共享内存上的竞争问 题。记录的进程可以与其他进程甚至内核设备驱 动程序共享内存，其中未记录的代码可以执行与 跟踪线程访问竞争的写入。幸运的是，这对于在 常见 Linux 桌面环境中运行的应用程序来说很 少见，仅在四种常见情况下发生：应用程序与 PulseAudio 守护程序共享内存，应用程序与 X 服务器共享内存，应用程序与内核图形驱动程 序和 GPU 共享内存，和 vdso 系统调用。我们 通过自动禁用 PulseAudio 和 X 共享内存的 使用（在两种情况下都回退到套接字传输）和禁 用从应用程序直接访问 GPU 来避免前三个问题 。 vdso 系统调用是一种 Linux 优化，它完全 在用户空间中实现一些常见的只读系统调用 （例如 gettimeofday），部分通过读取与内核 共享的内存并由内核异步更新。我们通过修补 它们的用户空间实现来禁用 vdso 系统调用 ，以执行等效的真实系统调用。 应用程序仍然可以以有问题的方式与未记录的 进程共享内存，尽管这在实践中很少见，并且通 常可以通过扩大 RR 记录的进程组的范围来解 决。 2.6 非确定性指令 几乎所有 CPU 指令都是确定性的，但有些不是 。一种常见的非确定性 x86 指令是 RDTSC，它 读取时间戳计数器。这个特定的指令很容易处理 ，因为 CPU 可以配置为在 RDTSC 上进行捕获 ，而 Linux 通过 prctl API 公开这一点，因此 我们可以捕获、模拟和记录每个 RDTSC。 其他相对较新的 x86 指令更难处理。RDRAND 生成随机数，希望完全不是确定性的。我们只 在 GNU libstdc++ 的一个地方遇到过它，所以 RR 明确地修补了它。XBEGIN 和相关的指令支 持硬件事务内存。从用户空间的角度来看，这些 是不确定的，因为硬件事务可能成功或失败，具 体取决于 CPU 缓存状态。幸运的是，到目前为 止，我们只发现系统 pthreads 库使用了这些 ，并且我们动态地将自定义补丁应用于该库以禁 用硬件事务的使用。CPUID 指令主要是确定性 的，但它的一个特性是返回正在运行的内核的索 引，这会影响 glibc 中的行为，并且可以随着 内核在内核之间迁移进程而改变。我们使用 L inux sched setaffinity API 来强制所有跟 踪线程在特定的固定核心上运行，并在重放期间 强制它们在该核心上运行。如果我们可以捕获并 模拟 CPUID 指令，我们可以很容易地在表现良 好的程序中避免大多数这些问题，因为那时我们 可以屏蔽指示支持 RDRAND、硬件事务等的功能 位。现代 Intel CPU 支持这（“CPUID 故障” ）；我们正在为此向 Linux 添加一个 API。 2.7 减少跟踪大小 对于许多应用程序来说，它们的大部分输入是内存 映射文件，主要是可执行代码。在每次执行时将所 有可执行文件和库复制到记录的跟踪中会占用大 量时间和空间。RR 创建指向内存映射可执行文件 的硬链接，而不是复制它们；只要系统更新或重新 编译用新文件替换可执行文件，而不是写入现有文 件，链接就会保留旧文件数据。这在实践中效果很 好。更好的是，XFS 和 Btrfs 等现代文件系统提 供了文件的写时复制逻辑副本（甚至文件中的块范 围），非常适合我们的目的。当映射文件与记录的跟 踪位于同一文件系统上，并且文件系统支持克隆时 ，RR 会将映射文件克隆到跟踪中。这些克隆操作在 时间和空间上基本上是免费的，直到/除非原始文 件被修改或删除。 RR 压缩所有跟踪数据，克隆文件除外 和块，使用 zlib “deflate” 方法。通过这些 优化，在实践中跟踪存储不是问题。4.4 节介 绍了一些结果。 380 2017 USENIX 年度技术大会 USENIX 协会

2.8 其他细节 除了这些大问题，构建一个完整的录播系统还 需要很多细节，这里就不多说了。一些系统调用 （例如 execve）处理起来特别复杂。录制和重 放信号传递很复杂，部分原因是信号传递对用 户空间内存的副作用记录很少。高级 Linux 内核功能，例如 unshare（内核命名空间）和 seccomp，需要经过深思熟虑的处理。其中许多 细节很有趣，但它们不会影响整体方法。 3 进程内系统调用拦截 上一节中描述的方法有效，但开销高得令人 失望（参见图 5 低）。核心问题是，对于每个 tracee 系统调用，如图 1 所示，tracee 执行四个上下文切换：两个阻塞 ptrace 通知，每个都需要从 tracee 到 RR 和返回 的上下文切换。对于常见的系统调用，例如 gettimeofday 或从缓存文件中读取，即使 是单个上下文切换的成本也会使系统调用本 身的成本相形见绌。为了显着减少开销，我们 必须在处理这些常见的系统调用时避免上下 文切换到 RR。 因此，我们将一个库注入到记录的进程中，该库 拦截常见的系统调用，执行系统调用而不触发 ptrace 陷阱，并将结果记录到与 RR 共享的专 用缓冲区。RR 定期将缓冲区刷新到其跟踪。这 个概念很简单，但有一些问题需要克服。 3.1 拦截系统调用 在进程中拦截系统调用的常用技术是使用动态 链接在进行系统调用的 C 库函数上插入包装函 数。在实践中，我们发现这种方法是不够的，因为 应用程序直接进行系统调用，并且由于 C 库的 变化和需要自己预加载的应用程序 [37, 3]） 而脆弱。 相反，当被跟踪者进行系统调用时，RR 通过 ptrace 陷阱得到通知，并尝试重写系统调用指 令以调用我们的拦截库。这很棘手，因为在 x 86 上，系统调用指令有两个字节长，但我们需 要用一个 5 字节的调用指令来替换它。在实践 中，经常执行的系统调用指令后面跟着一些已知 的、固定的指令序列。例如，许多系统调用指令 后跟一个 cmpl $0xfffff001,%eax 指令测试 系统调用结果。我们在拦截中添加了五个手写存 根 Tracee thread T read(fd, buf, size) syscall\_hook() redirect arg2 to syscall buffer N = syscall\_result\_reg copy N syscall buffer bytes to buf write system-call record to syscall buffer seccomp-bpf filter → ALLOW sys\_read 图 2：使用系统调用拦截进行录制 库在返回修补代码之前执行系统调用后指令。 收到 ptrace 系统调用通知后，RR 将系统调用 指令及其后续指令替换为对相应存根的调用。 我们（尝试）将所有系统调用指令重定向到拦截 库，但为简单起见，它只包含最常见系统调用的 包装器，对于其他系统调用，它回退到执行常规 的 ptrace 捕获系统调用。 3.2 选择性地捕获系统调用ptrace 系统调用监控会触发所有系统调用的陷 阱，但是我们的拦截库需要避免针对选定系统调 用的陷阱。幸运的是，现代 Linux 内核支持选择 性地生成 ptrace 陷阱：seccomp-bpf。seccom p-bpf 主要用于沙盒。一个进程可以将一个以字 节码表示的 seccomp-bpf 过滤函数应用于另一 个进程；然后，对于目标进程执行的每个系统调用 ，内核运行过滤器，传入传入的用户空间寄存器值 ，包括程序计数器。过滤器的结果指示内核要么允 许系统调用，要么以给定的 errno 失败，要么终 止目标进程，要么触发 ptrace 陷阱。过滤器执行 的开销可以忽略不计，因为过滤器直接在内核中 运行，并在大多数架构上编译为本机代码。 图 2 说明了使用进程内系统调用拦截记录一 个简单的读取系统调用。实线框代表拦截库中的 代码，灰色框代表内核代码。 RR 将一个特殊的内存页注入到每个 tracee 进程在固定地址（紧接在 execve 之后）。该页 面包含一个系统调用指令——“未跟踪指令”。R R 对每个记录的进程应用 seccomp-bpf 过滤 器，为每个系统调用触发 ptrace 陷阱——除非 程序计数器位于未跟踪指令处，在这种情况下允 许调用。每当拦截库需要进行未跟踪的系统调用 时，它都会使用该指令。

3.3 检测阻塞的系统调用 一些常见的系统调用有时会阻塞（例如在空管道 上读取）。因为 RR 一次运行一个跟踪线程，如 果一个线程在没有通知 RR 的情况下进入阻塞 系统调用，它将挂起并可能导致整个记录死锁（ 例如，如果另一个跟踪线程即将写入管道）。每 当未跟踪的系统调用阻塞时，我们需要内核通知 RR 并挂起跟踪线程，以确保我们可以调度不 同的跟踪线程。 我们使用 Linux perf 事件系统来监控 PE RF COUNT SW CONTEXT SWITCHES。内核每次 从 CPU 内核调度线程时都会引发这些事件之 一。拦截库为每个线程监视这些事件，并在每次 事件发生时请求内核向被阻塞的线程发送信号 。这些信号触发对 RR 的 ptrace 通知，同时 阻止线程进一步执行。为了避免虚假信号（例如 ，当线程由于正常的时间片到期而被取消调度 时），该事件通常被禁用并在可能阻塞的未跟踪 系统调用期间显式启用。尽管如此，在启用和禁 用事件之间的任何时间点都可能发生虚假 S WITCHES。我们通过仔细检查跟踪状态来处理 这些边缘情况。 图 3 说明了使用系统调用拦截记录阻塞读取 系统调用。内核调度线程，触发性能事件，该事 件向线程发送信号，重新调度它，中断系统调用 ，并向记录器发送 ptrace 通知。记录器记下 一个被拦截的系统调用在线程 T 中被中断， 然后检查阻塞系统调用中的任何跟踪线程是否 已经进展到系统调用退出并生成 ptrace 通 知。在这个例子中 T2 有 完成了一个（未拦截的）阻塞 futex 系统 调用，所以我们继续执行 T2。 恢复被信号中断的拦截系统调用（例如图 3 中的 T 的读取调用）更加复杂。解释这需 要了解 Linux 系统调用重启语义，这太复杂 了，无法在此处的可用空间中解释。 3.4 处理重放 从概念上讲，在录制期间，我们需要将系统调用 输出缓冲区复制到跟踪缓冲区，在重放期间， 我们需要将结果从跟踪缓冲区复制到系统调用 输出缓冲区。这是一个问题，因为拦截库是录制 和回放的一部分，因此在两种情况下都应该执 行相同的代码。（以前使用用户级系统调用拦截 的工作 [11, 36, 21, 29] 通过对重放保真 度的要求不那么严格来避免这些问题。） 出于这个原因（并且为了避免第 2.3 节中讨论 的那种竞争），拦截库重定向系统调用输出以直接 写入跟踪缓冲区。系统调用完成后，拦截库将输出 数据从跟踪缓冲区复制到原始输出缓冲区。在重 放期间，未跟踪的系统调用指令被替换为空操作 ，因此系统调用不会发生；结果已经存在于跟踪缓 冲区中，因此从跟踪缓冲区到输出缓冲区的系统 调用后副本可以满足我们的需要。 在录制过程中，每个未跟踪的系统调用都会设置 一个结果寄存器，并且拦截库将其写入跟踪缓冲 区。重放必须改为从跟踪缓冲区中读取结果寄存 器。我们使用条件移动指令，以便控制流在录制 和回放之间完全一致。条件是从一个重播全局变 量中加载的，因此保存条件的寄存器在很短的指 令跨度内是不同的（并且之后显式清除）。 处理“in-out”系统调用内存参数很棘手。在记录 期间，我们将输入缓冲区复制到跟踪缓冲区，将 系统调用指针传递给跟踪缓冲区，然后将跟踪缓 冲区内容复制回输入缓冲区。在重放期间执行第 一个副本将覆盖保存系统调用结果的跟踪缓冲 区值，因此在重放期间，我们使用条件移动将源 地址副本设置为目标地址，将该副本变为无操作 。 我们可以允许拦截库的重放与其记录的行为有 更大的差异，但这必须非常小心。我们必须确保 RCB 计数在两条路径上相同，并且每当我们退 出拦截库或在拦截库中捕获到 RR 时，寄存器值 都是一致的。最小化分歧是最简单的。 382 2017 USENIX 年度技术大会 USENIX 协会

3.5 使用块克隆优化读取 当输入文件与记录的跟踪位于同一文件系统上 并且文件系统支持文件块的写时复制克隆时， 对于大块对齐读取，系统调用拦截代码将数据 克隆到每个线程“克隆数据”跟踪文件，绕过正 常的系统调用记录逻辑。这大大减少了文件读 取密集型工作负载的空间和时间开销；见下一 节。 此优化通过克隆输入块然后从原始输入文件中读 取输入数据来工作。这引发了一场可能的竞争： 在克隆和读取之间，另一个进程可能会覆盖输入 文件数据，在这种情况下，重放期间读取的数据将 与录制期间读取的数据不同，从而导致重放失败 。但是，当在 Linux 下文件读取与写入竞争时， 读取器可以接收到新旧数据的任意混合，因此这 种行为几乎可以肯定是一个严重的错误，实际上 这种错误似乎并不常见。可以通过从克隆数据文 件而不是原始输入文件中读取来避免争用，但这 表现非常差，因为它破坏了 Linux 的 reada h ead 优化（因为克隆数据文件中的数据在需要之 前永远不可用）。 4 个结果 4.1 工作负载 选择基准以阐明 RR 的优势和劣势，同时还包含 实际使用的代表。它们被调整为适合系统内存（ 以最小化 I/O 对测试结果的影响），每个运行大 约 30 秒（除了 cp，30 秒的运行时间要求它不 适合内存）。 cp 使用 cp -a 复制 glibc 的 git che ckout（修订版 2d02fd07）（根据 du -h，1 5200 个文件构成 732MB 数据）。cp 是单线 程的，大量使用同步读取和其他各种与文件系 统相关的系统调用。 make 使用 make -j8 构建 DynamoRio [8 ]（版本 6.1.0）（限制为单核时省略 -j8）。 这测试了许多短期进程的潜在并行执行。 octane 在 Mozilla Spidermonkey Ja vascript 引擎（Mercurial 修订版 9bd 900888753）下运行 Google Octane 基准 测试。这说明了复杂语言运行时中 CPU 密 htmltest 运行 Mozilla Firefox HTML 表单测试（Mercurial 修订版 9bd900888753）。从记录中排除线束 （使用 mach mochitest -f plain --debugger R R dom/html/test 来自实际使用情况。大约 30% 的用户空间 CPU 时间在利用。sambatest 通过 mak e test TESTS=samba4.echo.udp 运行 Samba（git 修订版 9ee4678b）UDP echo 测试。这是一个真实世界使用的例子。所有测试都在 Dell XPS15 笔记本电脑上运行 ，该笔记本电脑配备四核 Intel Skylake CPU（8 个 SMT 线程）、16GB RAM 和 51 2GB SSD，在 Fedora Core 23 Linux 中 4.2 开销 表 1 显示了各种配置的挂钟运行时间，标准化为 基线配置的运行时间。octane 旨在运行固定的 时间长度并报告分数，因此我们报告基线分数与 被测配置分数的比率 - 重放测试除外，其中报告 的分数必然与录音时的分数。对于辛烷值回放测 试，我们报告基线分数与记录分数的比率，乘以回 放运行时间与记录运行时间的比率。每个测试运 行六次，丢弃第一个结果并报告其他五个结果的 几何平均值。因此，结果代表暖缓存性能。“单核” 报告了使用 Linux 任务集仅将所有线程限制为 单核的开销。“Record no-intercept”和“Rep lay no-intercept”报告了禁用进程内系统调用 拦截（也禁用块克隆）的开销。“Record no-clo ning”报告仅禁用块克隆的开销。“DynamoRio- null”报告在 DynamoRio [8]（版本 6.1.0）“ null 工具”下运行测试的开销，以估计使用动态 代码检测作为实现技术的开销的下限。（据报道， DynamoRio 是最快的动态代码检测引擎之一。

4.3 观察 make 的开销明显高于其他工作负载。将 mak e 强加到单个内核上会导致严重的减速。此外 ，使 fork 和 execs 2430 进程，大多是短暂 的。（下一个最多产的工作负载是 89 的 sam batest。）进程内系统调用拦截仅在加载拦截库 后才开始在进程中工作，但在完成之前至少执行 了 80 次系统调用，因此它的有效性仅限于短 暂的过程。图 4 显示了除 make 之外的工作负 载的整体记录和重放。图中的误差线显示 95 % 的置信区间；这些结果在运行中高度稳定。

不包括制作，RR 的录制速度低于两倍。不包括 make，RR 的重播开销低于其录制开销。在 c p 中，重放甚至可以比正常执行更快，因为系统 调用做的工作更少。对于此处未表示的交互式应 用程序，由于消除了空闲期，因此重播所需的时 间比原始执行要少得多。 octane 是这里唯一的工作负载，而不是大量使 用多个内核，这占了 RR 在 octane 上的大部分 开销。 图 5 显示了系统调用拦截和阻塞克隆对录制 的影响。系统调用拦截优化大大减少了记录（ 和重放）开销。克隆文件数据块是 cp 记录的一 项重大改进，但对其他工作负载基本上没有影 响。 图 6 将 RR 记录开销与 Dy namoRio 的“空 工具”进行了比较，该工具通过 DynamoRio 检测引擎运行所有代码，但不会修改代码，超出 维持监督执行所需的任何内容；这代表了最小 开销的代码检测配置。DynamoRio 在辛烷值 1 上坠毁。cp 执行的用户空间代码非常少，并 且 1 我们报告了 DynamoRio 在我们的“辛烷值”工作上的崩溃—— 在 https://github.com/DynamoRIO/dyn amorio/issues/1930 向开发人员加载。 图 5：优化的影响 DynamoRio 在该工作负载上的开销很低 。在 make 和 sambatest DynamoRio RR 录制，即使在 make DynamoRio 上也可以 利用多个核心。在 htmltest 上，DynamoRi o 的开销非常高，可能是因为该测试运行了大 量带有动态生成和修改的机器代码的 Java script。在动态检测之上实现记录和重放会产 生大量的额外开销，因此我们预计最终系统的 开销会比 RR 高得多。 4.4 存储空间使用情况 RR 跟踪包含三种数据：克隆（或硬 linked) 用于内存映射操作、克隆文件块和所有 其他跟踪数据的文件，尤其是事件元数据和一般 系统调用的结果。 内存映射文件主要是跟踪加载的可执行文件和 库。虽然原始文件不会更改或删除，这在实践中 通常是正确的，但它们的克隆不会占用额外的空 间，也不需要写入数据。RR 没有尝试合并重复 的文件克隆，因此大多数跟踪都包含许多重复项 ，并且在实践中报告这些文件的有意义的空间使 用情况既困难又不重要。克隆文件块也是如此。

5.1 硬件 如第 2.4.1 节所述，RR 需要一个“确定的”硬 件性能计数器来衡量应用程序进度。出于我们的 目的，理想的性能计数器将计算在用户空间中观 察到的退休指令的确切数量（例如，计算一次中 断并重新启动的指令）。虚拟机应该支持可靠的 性能计数器虚拟化。目前 RR 在 KVM 和 VMw are 下工作，但 VMware 的 VM 退出集群优化 [4] 在实施时打破了 RCB 计数器的确定性， 必须手动禁用。一些 x86 CPU 指令是不确定的 。第 2.6 节讨论了我们当前的解决方法。公开对 捕获 CPUID 的硬件支持对于长期控制这些指令 很重要。我们希望支持使用硬件事务内存 (XB EGIN/XEND) 记录和重播程序。如果硬件和操作 系统可以配置为在任何失败的事务上发出信号 就足够了。捕获所有其他非确定性指令（例如 RDRAND）会很有用。将 RR 移植到 ARM 失败， 因为所有 ARM 原子内存操作都使用“加载链接 /存储条件”方法，这种方法本质上是不确定的。 条件存储可能由于非用户空间可观察活动而失 败，例如硬件中断，因此执行原子内存操作的代 码的退休指令或条件分支的计数是不确定的。 这些操作被内联到非常多的代码位置，因此看起 来修补它们是不可行的，除非通过普遍的代码检 测或编译器更改。在 x86(-64) 上，原子操作（ 例如比较和交换）在用户空间状态方面是确定性 的，因此不存在此类问题。 5.2 软件 如第 2.5 节所述，RR 依赖于配置应用程序 以避免与未记录的进程共享内存。我们描述了 RR 性能如何取决于现代 Linux 功能：se ccomp-bpf 选择性地捕获系统调用，PERF COUNT SW CONTEXT SWITCHES 性能事件以 处理阻塞系统调用，以及写时复制文件和块克 隆 API 以减少 I/O高架。有效的记录和重放 取决于清楚地识别代码在其中确定性地重放 的边界，并将所有输入的时间和内容记录和重 放到该边界。在 RR 中，该边界主要是 ker - nel 和用户空间。这适合 Linux：大多数 Li nux 用户/内核接口在不同操作系统版本之间 都是稳定的，相对简单且文档齐全，并且很容易 计算边界内发生的硬件性能事件（即特定进程 的所有用户空间事件））。这在其他操作系统中 不太正确。例如，在 Windows 中，用户/内核界 面没有公开记录，并且显然比 Linux 更复杂 且更不稳定。在 Windows 上实施和维护 RR 方法将比在 Linux 上更具挑战性，至少对于 操作系统供应商以外的任何人来说都是如此。 6 相关工作 6.1 全系统回放 ReVirt [17] 是一个早期项目，它记录和回放 整个虚拟机的执行。VMware [28] 曾一度使用 相同的方法来支持 VMware Workstation 中 的记录和重放调试，但后来停止了该功能。全系 统模拟器 Simics 通过确定性重新执行 [2 0] 支持反向执行调试。已经努力为 QEMU [ 15, 16, 38] 和 Xen [18, 10] 添加一些记 录和回放支持。全系统记录和回放可能很有用 ，但将应用程序提升到虚拟机中通常不方便。 许多记录和重放应用程序都需要廉价的检查点 ，而对 VM 映像进行检查点通常比对一个或几 个进程进行检查点更昂贵。 6.2 使用内核重放用户空间支持 Scribe [26]、dOS [6] 和 Arnold [14] 通过 扩展具有记录和重放功能的 OS 内核来重放一 个进程或一组进程。内核更改使维护和部署更加 困难——除非将记录和重放集成到基本操作系统 中。但是向内核添加侵入性的新特性是有风险的 ，因此如果可以在内核之外很好地实现记录和重 放，那么将其移入内核可能是不可取的。 6.3 纯用户空间重放 至少从 MEC [11] 以及后来的 Jockey [36 ] 和 liblog [21] 开始，就已经存在纯用户 空间记录和回放系统。这些系统不处理异步事 件计时和其他操作系统功能。PinPlay [34] 、iDNA [7]、UndoDB [1] 和 TotalView Re playEngine [22] 使用代码检测来记录和重 放异步事件时序。与 UndoDB 和 RR 不同，P inPlay 和 iDNA 在 strument 中全部加载 386 2017 USENIX 年度技术大会 USENIX 协会

数据竞争的存在并避免计算系统调用的影响，但 这给它们带来了比其他系统更高的开销。与支持 异步事件的其他系统相比，RR 通过避免代码检测 来实现更低的开销。 6.4 更高级别的回放 记录和回放功能已集成到语言级虚拟机中。De jaVu [12] 为 Jalape ～ no Java VM 添加了 记录和回放功能。Microsoft IntelliTrace [2] 检测 CLR 字节码以记录高级事件以及函 数调用的参数和结果；它不会产生完整的回放。 诸如 Chronon [13] 之类的系统用于 Java 工具字节码，以收集足够的数据以提供重放执 行的外观，而无需实际进行重放。Do los [9] 为 Webkit 中的 JS 应用程序提供记录和重 放，方法是在浏览器中记录和重放非确定性输入 。R2 [23] 通过检测库接口提供记录和回放； 处理数据竞争或异步事件需要用户努力隔离非 确定性。这些系统的范围都比重放一般用户空间 执行的能力要窄得多。 6.5 并行回放 以低开销记录在多个内核上并发运行的应用 程序线程，具有数据竞争的可能性，极具挑战 性。PinPlay [34] 和 iDNA/Nirvana [7] 检测共享内存负载并报告高开销。SMP-ReV irt [18] 使用硬件页面保护跟踪页面所有权 ，并在具有大量共享的基准上报告高开销。D oublePlay [39] 运行应用程序的两个实例 ，因此当单独的应用程序可能使可用内核饱和 时具有很高的开销。ODR [5] 具有低记录开销 ，但重放可能非常昂贵，并且不能保证重现相 同的程序状态。Castor [29] 通过修改编译 器和运行时系统来检测同步代码，这为易于部 署创造了障碍，并且在存在数据竞争的情况下 无法可靠地重放。 一般的、低开销并行记录的最大希望似乎是 硬件支持。FDR [41]、BugNet [31]、Reru n [24]、DeLorean [30] 和 QuickRec [35] 等项目已经探索了低开销并行记录硬 件。 7 未来工作 RR 干扰执行，尤其是通过强制所有线程 到单个核心上，因此可能无法重现 出现在 RR 之外的错误。我们通过引入“混沌 模式”解决了这个问题，该模式智能地为调度 决策添加随机性，使我们能够重现更多错误， 但这项工作超出了本文的范围。有更多机会来 增强记录器以发现更多错误。 将记录和重放支持放入内核具有性能优势， 例如减少记录上下文切换的成本。我们也许能 够找到可重用的原语，这些原语可以添加到内 核中，以提高用户空间记录和重放的性能，同 时比完整的内核实现侵入性更小。 如果它们不共享内存，那么记录在多个内核上 并行运行的多个进程似乎是可行的——或者， 如果它们共享内存，则受 SMP-ReVirt [18 ]、dthreads [27] 或 Castor [29] 启发的 技术可能会起作用对于一些工作负载。 记录和回放的应用可能比基础技术更有趣、 更重要。例如，可以在重放期间执行高开销动 态分析 [14,15,34]，可能会在执行的多个部 分并行化。和 RR 的无工具方法，可以收集每个- 记录期间的采样堆栈和性能计数器值等形式 数据，并将该数据与重放期间生成的丰富分析 相关联（例如缓存模拟）。始终在线的记录和回 放将使在现场查找和修复错误变得更加容易 。演示用于记录和回放的引人注目的应用程序 将为构建对商品硬件和软件的支持奠定基础 。 8 结论 Linux 在商品 x86 CPU 上的当前状态能够以 低开销实现单核用户空间记录和重放，而无需普 遍的代码检测——但仅此而已。这是偶然的；我们 将软件和硬件功能用于其并非旨在服务的目的 。这也是最近的发展；五年前 seccomp-bpf 和 Linux 文件克隆 API 还不存在，具有可从用 户空间使用的确定性硬件性能计数器的商品架 构才刚刚出现（英特尔 Westmere） 2 。通过确 定这些功能在记录和回放方面的实用性，我们希 望它们将得到越来越广泛的未来系统的支持。 通过提供一个开源、易于部署、生产就绪的记录 和回放框架，我们希望能够使这项技术的应用更 加引人注目。