ghOSt:快速灵活的用户空间委托 Linux 调度的

Jack Tigar Humphries1, Neel Natu1, Ashwin Chaugule1, Ofir Weisse1, Barret Rhoden1, Josh Don1, Luigi Rizzo1, Oleg Rombakh1, Paul Turner1, Christos Kozyrakis2

1Google, Inc. 2斯坦福大学

摘要

我们介绍了 ghOSt，这是我们将内核调度决策委托给用户空间代码的基础设施。ghOSt 旨在支持我们数据中心工作负载和平台的快速变化的需求。

改进调度决策可以极大地提高重要工作负载的吞吐量、尾部延迟、可伸缩性和安全性。然而，内核调度器很难在大型舰队中高效地实现、测试和部署。最近的研究表明，在自定义数据平面操作系统中定制的调度策略可以在数据中心设置中提供令人信服的性能结果。然而，这些收益已被证明难以实现，因为在应用程序粒度上部署自定义操作系统映像是不切实际的，特别是在多租户环境中，这限制了这些新技术的实际应用。ghOSt 为 Linux 环境中的用户空间进程提供了调度策略的通用委托。ghOSt 提供了状态封装、通信和动作机制，这些机制允许在用户空间代理中复杂地表达调度策略，同时帮助同步。程序员可以使用任何语言来开发和优化策略，这些策略的修改无需重启主机。ghOSt 支持广泛的调度模型，从每 cpu 调度到集中式调度，从运行到完成到抢占式调度，并为调度操作带来较低的开销。在谷歌 Snap 和谷歌搜索等实际工作负载上验证了 ghOSt 的性能。我们表明，通过使用 ghOSt 而不是内核调度器，我们可以在为数据中心工作负载启用策略优化、无中断升级和故障隔离的同时，快速实现相当的吞吐量和延迟。我们开源我们的实现，以使未来基于 ghOSt 的研究和开发成为可能。

CCS 概念•软件及其工程;

关键词操作系统，thread 调度

ACM 参考格式:

Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner 和 Christos Kozyrakis. 2021。ghOSt:Linux 调度的快速灵活的用户空间委托。在 ACM SIGOPS 第 28 届操作系统原理研讨会(SOSP ' 21)上，2021 年 10 月 26 日至 28 日，德国虚拟事件。ACM，纽约，纽约，美国，17 页。https: / / doi.org/10.1145/3477132.3483542

ACM ISBN 978-1-4503-8709-5/21/10。

https://doi.org/10.1145/3477132.3483542

1 介绍

CPU 调度对应用的性能和安全性有着重要的影响。针对特定工作负载类型量身定制策略可以大幅提高延迟、吞吐量、硬/软实时特性、能源效率、缓存干扰和安全性等关键指标[1-26]。例如，新宿请求调度器[25]优化了高度分散的工作负载(混合了短请求和长请求的工作负载)，将请求尾延迟和吞吐量提高了一个数量级。虚拟机工作负载的 Tableau 调度器[23]在多租户场景下证明了吞吐量提高了 1.6×，延迟提高了 17×。Caladan 调 scheduler[21]专注于前台低延迟应用和后台最努力应用之间的资源干扰，将网络请求尾部延迟提高了 11000 ×。为了缓解最近的硬件漏洞[27-32]，运行多租户主机的云平台不得不快速部署新的核心隔离策略，隔离应用程序之间共享的处理器状态。

跨大型机群设计、实现和部署新的调度策略是一项艰巨的任务。它要求开发人员设计能够平衡许多应用程序的特定性能需求的策略。实现必须符合复杂的内核架构，在很多情况下，错误会导致整个系统崩溃，或者由于意想不到的副作用[33]而严重阻碍性能。即使成功了，升级的破坏性本质在主机和应用程序停机时也会带来自己的机会成本。这在风险最小化和进展之间产生了具有挑战性的冲突。

以前通过设计用户空间解决方案来提高性能和降低内核复杂度的尝试有明显的缺点:它们需要对应用程序实现进行大量修改[1,10,21,25,34,35]，

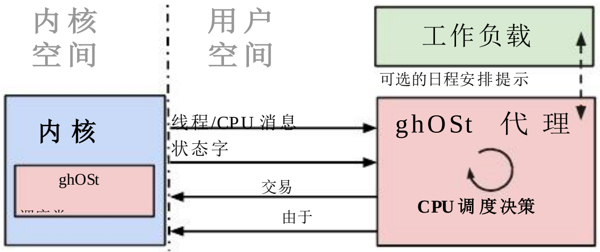


图 1 所示。ghOSt 概述。

需要专门的资源，以便高度响应
Sive[1,21,25]，或者要求特定于应用程序

内核修改[1,2,10,21,25,34 - 37]。

虽然 Linux 支持多种调度实现;为每个应用量身定制、维护和部署不同的机制，在大型舰队上进行管理是不切实际的。我们的目标之一是在一个稳定的内核 ABI 上，确定哪些内核更改需要在用户空间中进行大量优化(和实验)。此外，硬件环境一直在变化，强调现有的调度抽象模型[38]最初是为了管理更简单的系统而设计的。调度器必须不断进化以支持这些快速变化:不断增加的核心计数、新的共享属性(例如 SMT)和异构系统(例如大。小[39])。NUMA 属性即使是单插座平台也在不断增加(如小芯片[40])。AWS Nitro[41]和 dpu[42]等计算卸载以及 gpu 和 tpu[43]等领域特定的加速器都是新型的紧密耦合计算设备，它们完全超出了经典调度器的领域。我们需要一种范式来更有效地支持不断发展的问题域，而不是今天的单片内核实现。

本文介绍了 ghOSt 的设计及其在生产用例和学术用例上的评估。ghOSt 是原生 OS 线程的调度器，它将策略决策委托给用户空间。ghOSt 的目标是从根本上改变调度策略的设计、实现和部署方式。ghOSt 提供了用户空间开发的敏捷性和部署的易用性，同时仍然支持+ s 规模的调度。ghOSt 为用户空间软件提供抽象和接口，以定义复杂的调度策略，从每 cpu 模型到系统范围(集中式)模型。重要的是，ghOSt 将内核调度机制与策略定义解耦。该机制驻留在内核中，很少发生变化。策略定义驻留在用户空间中，变化很快。我们开源我们的实现，以使未来的研究和开发使用 ghOSt[44,45]。通过调度本机线程，ghOSt 无需任何更改就可以支持现有的应用程序。在 ghOSt 中(图 1)，调度策略逻辑运行在正常的 Linux 进程中，称为代理，代理通过 ghOSt API 与内核交互。内核通过异步路径中的消息队列(§3.1)通知代理所有托管线程的状态变化——例如，线程创建/阻塞/唤醒。的

然后，agent(s)在同步路径上使用基于事务的 API 提交线程的调度决策(§3.2)。ghOSt 支持多策略并发执行、故障隔离、CPU 资源分配给不同应用。重要的是，为了实现车队中现有系统使用 ghOSt 的实际过渡，它与内核中运行的其他调度器共存，例如 CFS(§3.4)。

我们证明，ghOSt 使我们能够定义各种策略，从而在学术和生产工作负载上获得与现有调度器相当或更好的性能(§4)。我们描述了关键 ghOSt 操作的间接费用(§4.1)。结果表明，ghOSt 的调度开销很小，从 265 ns 的消息传递开销、几百纳秒的 agent 上下文切换开销和 888 ns 的线程调度开销不等，使得 ghOSt 的调度开销仅比现有的内核调度器略高。通过摊销，这些开销仅允许单个 ghOSt 代理每秒调度超过 200 万个线程(图 5)。我们通过实现集中和抢占策略来评估 ghOSt s-scale 在存在高请求分散[10,25]和对抗[1,21](§4.2)的情况下导致高吞吐量和低尾延迟的工作负载。将 ghOSt 与新宿专用数据平面[25]进行比较，表明 ghOSt 的开销最小且具有竞争力 s-scale 性能(在新宿的 5%以内)，同时支持更广泛的工作负载集，包括多租户。

我们还为 Snap[2](我们在数据中心使用的生产包交换框架)实施了一项策略，在某些情况下，与我们目前使用的软实时调调度程序 MicroQuanta 相比，其尾部延迟可与之相当，在某些情况下比 MicroQuanta 好 5-30%(§4.3)。然后，我们为运行谷歌搜索(§4.4)的机器实现了一个 ghOSt 策略。通过根据机器的拓扑结构定制该策略，在<1000 行代码中，我们证明了 ghOSt 与现有调度器提供的吞吐量匹配，并往往比延迟高出 40-50%。最后，我们实现了一个针对最近发现的微架构漏洞的虚拟机策略(§4.5)[27-32]，并表明 ghOSt 的性能与一个纯内核内策略实现具有竞争力。有了 ghOSt，调度策略-以前需要大量的内核修改-可以在 10 秒或 100 行代码中实现。

2 背景和设计目标

大型云提供商和用户(例如，云客户端)有动力部署新的调度策略，以优化在日益复杂的硬件拓扑上运行的关键工作负载的性能，并通过在单独的物理核心上隔离不可信的线程，提供针对 Spectre[29-32]和 L1TF/MDS[27, 31, 46]等新硬件漏洞的保护。

Linux 中的调度。Linux 支持通过调度类[7]实现多种策略。类按优先级排序:一个线程调度优先级更高

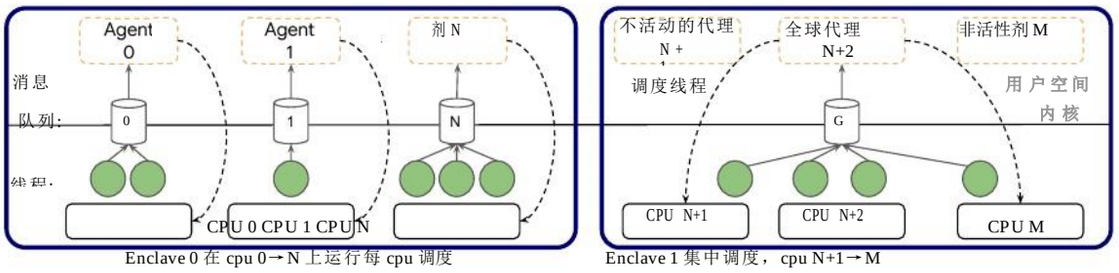
类将抢占用低优先级类调度的线程。为了优化特定应用程序的性能，开发人员可以修改调度类以更好地适应应用程序的需求，例如，优先考虑应用程序的线程以减少网络延迟[21]或使用实时策略调度应用程序的线程以满足数据库查询的截止日期[47]。原则上，云提供商或应用程序开发人员也可以为特定服务创建一个全新的类/策略(而不是修改现有的类/策略)优化，例如云虚拟机[24]或强化学习框架[20]。然而，在内核中实现和维护这些策略的复杂性导致许多开发人员转而使用现有的通用策略，如 Linux 完全公平调度器(CFS[7])。因此，Linux 中的现有类被设计为支持尽可能多的用例。那么，使用这些过于泛型的类来优化高性能应用程序以最大效率使用硬件就具有挑战性了。实现调度器是很困难的。当开发人员着手设计一个新的内核调度器时，他们会发现这些调度器很难实现、测试和调试。调度器通常是用低级语言(C 和汇编)编写的，不能利用有用的库(例如 Facebook Folly[48]和谷歌 Abseil[49])，也不能用流行的调试工具进行介绍。最后，调度器依赖于复杂的同步原语，并与之交互，包括原子操作、RCU[50]、任务抢占和中断，这使得开发和调试更加困难。长期维护也是一个挑战。Linux 很少合并新的调度类，而且特别不可能接受高度调优的非通用调度器。因此，随着上游 Linux 的发展，自定义调度器被维护在树外，并保持一致的合并变动。部署调度器更加困难。部署调度策略的更改需要在大型舰队中部署一个新内核。这对云提供商来说是极具挑战性的。以至于根据我们的经验，内核推出不能很好地容忍在 O(月)粒度以下。要在机器上安装新的内核，云提供商必须迁移或终止机器分配的工作，使机器静默，安装新的系统软件，允许系统守护进程重新初始化，最后，等待新分配的应用程序再次准备好服务。新调度器的初始部署只是一个开始。在部署了第一个候选版本之后，云提供商将频繁进行更改以修复错误和调优性能，重复上述昂贵的过程。例如，在谷歌，我们的磁盘服务器上有一个调度器错误，导致数百万美元的收入损失，直到该错误被发现并修复[51]。ghOSt 支持调度器更新、测试和调优，而无需更新内核和/或重新启动机器和应用程序。

用户级线程是不够的。ghOSt 是一个运行在用户空间的内核调度器，调度本地操作系统线程。相比之下，用户级线程运行时调度用户线程。这些运行时[22,48,52 - 60]在 N 个本机线程上复用 M 个用户线程。这在本质上是不可预测的:虽然用户空间运行时可以控制在给定的本地线程上运行哪个用户线程，但它不能控制何时计划实际运行该本地线程，或者它在哪个 CPU 上运行。更糟糕的是，内核可以取消对持有用户级锁的线程的调度。为了克服这个限制，开发人员有两个选择。(1)将 cpu 分配给运行用户线程的本地线程，从而保证了隐式控制。然而，这个选项在低工作负载利用率时浪费资源，因为专用的 cpu 不能与另一个应用程序共享(见 § 4.2)，并且需要围绕扩展能力进行广泛的协调。或者，开发人员可以(2)任由本机线程调度器摆布，允许共享 cpu，但最终失去对响应时间的控制，他们转向用户级运行时。ghOSt 通过保证对响应时间的控制，同时允许灵活地共享 CPU 资源，实现了两者的最佳结合。

为每个工作负载定制调度器/数据平面 OS 是不切实际的。之前的工作，如新宿、Shenango 和其他[1,10,21,25,34]，实现了高度专业化的数据平面操作系统，具有针对特定网络负载的自定义调度策略和网络栈。虽然这些系统为其目标提供了良好的性能，但它们的内核实现不能轻易更改，并且对其他工作负载提供了较差的性能。新宿[25]是 2,535 行代码，不能与系统中的其他应用程序共存(§ 4.2)。Shenango[1]是 8,399 行代码，只能实现网络负载的单线程调度策略。添加额外的策略需要进行重大的代码修改。此外，这两个系统都无法在没有特定网卡的机器上运行，Shenango 无法调度非网络负载。

通过 BPF 进行自定义调度是不够的。定制内核调度的一种有吸引力的方法是将 BPF[61]程序注入内核调度器，就像对其他内核子系统所做的那样[62]。确实可以实现一个 Linux 调度器类，它的函数指针调用一个 BPF 程序来决定接下来运行哪个线程。不幸的是，BPF 在其表达能力和它可以访问的内核数据结构方面受到了限制。例如，BPF 验证器必须能够确定循环将退出，而 BPF 程序不能使用浮点数。

更重要的是，BPF 程序同步运行，这意味着它们必须对调度事件做出快速反应，阻塞 CPU，直到它们完成。相比之下，异步调度器可以接收调度事件，并在稍后的时间对这些事件做出反应。因此，一个异步模型，比如 § 3.3 中描述的全局调度器，可以基于更广泛的系统视角，由多个调度事件组成，来做出调度决策。这是



Enclave 0 在 cpu 0→N 上运行每 cpu 调度　　Enclave 1 集中调度，cpu N+1→M

图 2。ghOSt 策略管理分配给飞地的 cpu。每个飞地都可以通过不同的 ghOSt 政策来管理。

正如§3.2 所解释的那样，BPF 在 ghOSt 中发挥了很大的作用，以加速快速路径操作，并在内核中的性能关键点提供定制。

2.1 设计目标

ghOSt 的目标是引入一种用于设计、优化和部署调度程序的新范式。我们在设计 ghOSt时考虑了以下要求:

策略应该易于实施和测试。随着新的工作负载、内核和异构硬件进入数据中心，实现新的调度策略不应该强制要求另一个内核补丁。正如 DashFS[63]所演示的，在用户空间而不是内核空间中实现系统简化了开发，并且可以使用流行的语言和工具，从而实现更快的迭代。

调度的表达性和效率。云提供商和用户需要为各种优化目标表达调度策略，包括面向吞吐量的工作负载， s-scale 延迟关键型工作负载、软实时和能效要求。

启用超越每 cpu 模型的调度决策。Linux 调度器生态系统实现了做出每 cpu 调度决策的调度算法，即:，他们使用每 cpu 模型。最近的工作展示了引人注目的尾延迟改进 s-scale 通过集中式、专用轮询调度线程的工作负载[1,21,25]，即。，它们使用集中式模型。其他系统，如 ESXi NUMA[64]和 Minos[15]也展示了在比单个 CPU 更粗的粒度上进行协调的好处，如 per-NUMA-node。集中式调度模型在 Linux 现有的"可插拔调度器"框架中很难得到支持，因为该框架是基于每 cpu 模型的假设实现的。ghOSt 应该支持将调度决策委托给远程 cpu，以支持从每 cpu 到集中式以及两者之间的所有调度模型。

支持多个并发策略。随着数百个核心和新的加速器的加入，机器容量继续水平扩展，在一台服务器上支持多个负载和租户。ghOSt 必须支持分区和共享机器，以便多个调度策略可以并行执行。

不中断更新和故障隔离。大型机群上的操作系统升级会导致昂贵的停机时间。的工作

分配给每个主机的工作必须迁移或重新启动，主机本身必须重新启动。这个耗时的过程会降低计算能力，并阻碍数据中心的容错能力，直到更新完成。对于出于类似原因推出自己策略的用户来说，这甚至是一个挑战:他们必须在手动下架和升级节点的同时维持服务正常运行时间。因此，调度策略应该与主机内核解耦，ghOSt 必须允许部署、更新、回滚新策略，甚至允许在不引起机器重新启动成本的情况下崩溃。

3 设计

我们现在讨论 ghOSt 的设计和实现，并解释它们如何实现§2 中列出的要求。

ghOSt 概览。图 1 总结了 ghOSt 设计。用户空间代理做出调度决策，并指示内核如何在 cpu 上调度本机线程。ghOSt 的内核端被实现为一个调度类，类似于常用的 CFS 类。这个调度类为用户空间代码提供了丰富的 API 来定义任意的调度策略。为了帮助代理做出调度决策，内核通过消息和状态词向代理公开线程状态(§3.1)。然后，代理通过事务和系统调用(§3.2)指示内核进行调度决策。

在本节中，我们将使用两个激励人心的例子:每 cpu 调度和集中式调度。经典的内核调度策略(如 CFS)是每 cpu 调度器。虽然这些策略通常使用负载平衡和工作窃取来平衡整个系统的负载，但它们仍然是从每 cpu 的角度来操作的。集中调度的例子类似于 Shinjuku[25]、Shenango[1]和 Caladan[21]中提出的模型。在这种情况下，有一个单一的全局实体不断地观察整个系统，并为其权限内的所有线程和 cpu 做出调度决策。

线程和 cpu。ghOSt 在 cpu 上调度本机线程。本节中提到的所有线程都是本地线程，与§2 中提到的用户级线程相反。我们将逻辑执行单元称为 cpu。例如，我们认为具有 56 个物理核和 112 个逻辑核(超线程)的机器有 112 个 cpu。

对机器进行分区。ghOSt 使用 enclave 在一台机器上支持多个并发策略。一个系统

可以按 CPU粒度划分为多个独立的飞地，每个飞地运行自己的策略，如图 2所示。从调度的角度来看，这些 enclave是隔离的。分区是有意义的，特别是当在一台机器上运行不同的工作负载时。通过机器拓扑设置这些 enclecle 的粒度通常很有用，例如 per-NUMA-socket 或 per-AMD-CCX[40]。enclave 还有助于隔离错误，限制代理崩溃对它所属的 enclave 的伤害(参见 § 3.4)。

ghOSt 用户空间代理。为了实现我们的许多设计目标，调度策略逻辑是在用户空间代理中实现的。这些代理可以用任何语言编写，并通过标准工具进行调试，使它们更容易实现和测试。为了实现容错和隔离，如果一个或几个代理崩溃，系统将退回到默认的调度器，比如 CFS。然后，在启动新的 ghOSt 用户空间代理时，机器仍然完全正常工作-要么是最后一个已知的稳定版本，要么是带有修复的更新版本。

多亏了崩溃恢复属性，更新调度策略相当于重新启动用户空间代理，而不必重新启动机器。这个属性允许针对各种硬件和工作负载进行试验和快速的策略定制。开发人员可以进行策略调整，并简单地重新启动代理。ghOSt 策略的动态更新在§3.4 中讨论。

不管调度模型是每 CPU 调度还是集中式调度，ghOSt 管理的每个 CPU 都有一个本地代理，如图 2 所示。在每 CPU 情况下，每个代理负责自己 CPU的线程调度决策。在集中式情况下，单个全局代理负责调度飞地内的所有 cpu。所有其他本地代理都是不活动的。每个代理都在 Linux pthread 中实现，所有代理都属于同一个用户空间进程。

3.1 内核到代理的通信

将线程状态暴露给代理。为了让代理为其权限范围内的线程做出调度决策，内核必须向代理公开线程状态。一种方法是将现有的内核数据结构内存映射到用户空间，比如 task_structs，这样代理就可以检查它们来推断线程状态。然而，这些数据结构的可用性和格式因内核和内核版本而异，用户空间策略实现与内核版本紧密耦合。另一种方法是通过 sysfs 文件公开线程状态，在/proc/pid/…时尚。然而，文件系统 api 对于快速路径操作来说效率很低，很难得到支持 s-scale 策略:open/read/fseek，最初是为块设备设计的，太慢且太复杂(例如，需要错误处理和数据解析)。

最终，我们需要一个既快速又不依赖底层内核线程实现的内核用户空间 API。受到分布式系统的启发，我们使用消息作为一种高效而简单的解决方案。

ghOSt 信息。在 ghOSt 中，内核使用表 1 中列出的消息通知用户空间代理线程状态的更改。例如，如果一个线程被阻塞，现在准备运行，内核会发布一条 THREAD_WAKEUP 消息。此外，内核用 TIMER_TICK 消息通知代理计时器滴答。为了帮助代理验证他们正在根据最新的状态做出决策，消息也有序列号，后面会解释。

消息队列。消息通过消息队列传递给座席。在 ghOSt 下调度的每个线程都被分配到一个队列，关于该线程状态变化的所有消息都被发送到该队列。在每 CPU 示例中，每个线程被分配到一个队列，该队列与它打算运行的 CPU 对应(图 2，左)。在集中式的例子中，所有线程都被分配到全局队列(图 2，右)。CPU 事件(如 TIMER_TICK)的消息被路由到与 CPU相关的代理线程的队列。

虽然实现队列的方法有很多，但我们选择在共享内存中使用自定义队列来高效地处理代理唤醒(下文将进行解释)。我们认为现有的队列机制对于 ghOSt 是不够的，因为它们只存在于特定的内核版本中。例如，BPF 系统通过 BPF 环缓冲区将 BPF 事件传递给用户空间[65]，最近版本的 Linux 也通过 io_urings 将异步 I/O 消息传递给用户空间[66]。这些都是同步消费者/生产者访问的快速无锁环缓冲区。然而，旧的 Linux 内核和其他操作系统不支持它们。

Thread-to-queue 协会。ghOSt 飞地初始化后，飞地中只有一个默认队列。代理进程可以使用 create/DESTROY_QUEUE()API 创建/销毁队列。添加到 ghOSt 的线程被隐式分配给将消息发送到默认队列。代理可以通过 ASSOCIATE_QUEUE()来改变这个分配。

Queue-to-agent 协会。队列可以有选择地配置为当消息产生到队列中时唤醒一个或多个代理。代理可以通过 CONFIG_QUEUE_WAKEUP()配置唤醒行为。在每 CPU 示例中，每个队列只与一个 CPU 关联，并被配置为唤醒相应的代理。在集中式的例子中，队列由全局代理连续轮询，因此唤醒是冗余的，因此没有配置。产生消息到队列中并在代理中观察它的延迟在§4.1 中讨论了。

代理唤醒使用标准的内核机制来唤醒阻塞的线程。这包括识别要唤醒的代理线程，将其标记为可运行的，可选地向目标 CPU 发送中断以触发重新调度，以及执行到代理线程的上下文切换。

在队列/CPU 之间移动线程。在我们的每个 CPU 示例中，为了在 CPU之间实现负载平衡和工作窃取，代理可以更改消息的路由

消息	由于
THREAD_CREATED	AGENT_INIT
THREAD_BLOCKED	START_GHOST
THREAD_PREEMPTED	TXN_CREATE()
THREAD_YIELD	TXNS_COMMIT()
THREAD_DEAD	TXNS_RECALL
THREAD_WAKEUP	CREATE_QUEUE
THREAD_AFFINITY	DESTROY_QUEUE
TIMER_TICK	ASSOCIATE_QUEUE()
	CONFIG_QUEUE_WAKEUP()

表 1。幽灵消息和系统调用。

通过 ASSOCIATE_QUEUE()从线程到队列。这取决于代理实现(在用户空间中)，以适当协调跨队列到代理的消息路由。如果一个线程将其关联从一个队列更改为另一个队列，而原始队列中有挂起的消息，则关联操作将失败。在这种情况下，代理必须在重新发出 ASSOCIATE_QUEUE()之前清空原始队列。将代理与内核同步。代理操作通过消息观察到的系统状态。然而，当代理正在做出调度决策时，新的消息可能会到达队列，从而改变该决策。对于每个 CPU 示例和集中式调度示例，这个挑战略有不同(见§3.2 和§3.3)。无论哪种方式，我们都用代理/线程序列号来解决这个挑战:每个代理都有一个序列号，，每当消息被发布到与该代理关联的队列时，该序列号都会递增。的使用 对于§3.2 中的每 CPU 示例。每个线程 都有一个序列号，，每当该线程发布一个新的状态改变消息时，该数字就会递增，。当一个代理弹出队列时，它会收到一条消息和它对应的序列号:(，)。我们解释如何使用 对于§3.3 中的集中式调度示例。通过共享内存公开序列号。幽灵使代理能够通过状态字有效地轮询有关线程和 CPU 状态的辅助信息，这些信息映射到代理的地址空间中。为了简洁起见，我们只讨论我们使用状态词来暴露序列号，和，送给代理商。当内核更新线程或代理的序列号时，它也会更新相应的状态词。然后，代理可以从共享映射中的状态词中读取序列号。

3.2 Agent-to-Kernel 通信

我们现在描述代理如何指示内核下一个调度哪个线程。

通过事务发送调度决策。代理通过提交事务向内核发送调度决策。代理必须能够调度其本地 CPU(每 CPU 情况)和其他远程 CPU(集中式情况)。提交机制必须能够快速支持 s-scale 策略和可扩展到数百个核。对于每 CPU 的示例，理论上一个系统调用接口就足够了。对于集中式调度，代理需要高效地将调度请求发送到多个 CPU 和

```
1  void Agent:: PerCpuSchedule () {
2
3      DrainMessageQueue();//从队列读取消息线程*next=
       runqueue_.;出列();
4
5      If (next == nullptr)返回;//运行队列为空。 //调度
       线程:
6
7      Transaction *txn =TXN_CREATE(next ->tid,
       my_cpu);TXNS_COMMIT ({txn});
8      if (txn ->status != TXN_COMMITTED) {
9
10
11     // ' next '的调度已经成功。
12
13 }
14
```

图 3。以每 CPU 代理代码调度线程。

然后检查这些请求是否成功。因此，共享内存接口更合适。作为题外话，在共享内存中使用事务作为调度接口将允许在未来将调度决策卸载给能够访问该内存的外部设备。

受到事务内存[67]和数据库[68]系统的启发，我们设计了自己的事务 API，通过共享内存实现。这些系统支持具有原子语义的快速、分布式提交操作，并且可以同时针对同一个远程节点进行多个提交。ghOSt 代理需要类似的属性。代理程序使用 TXN_CREATE()helper 函数在共享内存中打开一个新事务。代理写入要调度的线程的 TID 以及要调度线程的 CPU ID。在每 CPU 示例中，每个代理只调度自己的 CPU。当事务被填充时，代理通过 TXNS_COMMIT()系统调用将其提交给内核，这将启动提交过程并触发内核启动上下文切换。图 3 展示了一个简化的例子。

小组提交。在集中式调度示例中，为了允许 ghOSt 扩展到每秒数百个 cpu 和数十万个事务，我们必须降低昂贵的系统调用成本。我们通过引入组提交来摊销事务的成本。组提交也减少了发送到其他 cpu 的中断数量，类似于 Caladan[21]。代理通过将所有事务都传递给 TXNS_COMMIT()系统调用来提交多个事务。这个系统调用将昂贵的开销摊销到几个事务上。最重要的是，它通过使用大多数处理器中存在的批处理中断功能来平摊发送中断的开销。内核没有发送多个中断(每个事务发送一个中断)，而是向远程 cpu 发送单个批处理中断，从而节省了大量开销。

序列号和事务。在每 CPU 示例中，提交事务的代理将其 CPU 放弃给它正在调度的目标线程。在代理运行时发布到队列的消息不会引起唤醒，因为代理已经在运行。然而，队列中的新消息可能来自一个更高优先级的线程，

```
GlobalAgent::centrizedschedule () {
    DrainMessageQueue ();
    map <Cpu, Thread*>赋值;
    //GetIdleCPUs()将返回所有可用的 cpu。for (const
    cpu & cpu:GetIdleCPUs()){
        线程*next= runqueue_.;出列();
        If (next == nullptr) break;//运行队列为空。
        赋值[cpu] = next;//在' cpu '上执行' next '。 }
    //现在为所有分配发送事务:vector <Transaction*>
    txns = Schedule(assignments);for (const Transaction
    *txn: txns) {
        //检查' txn '是否提交成功。 if (txn ->status !=
}}}
```

图 4。一个简化的全局代理示例。

并且会影响调度决策，如果代理意识到这一点。代理只有在下一次唤醒时才有机会检查该消息，这就太晚了。现在，我们将解释如何通过每 CPU 示例的序列号来解决这一挑战。我们将在§3.3 中解释集中式调度的略有不同的情况。

我们使用代理序列号来解决这个挑战，。代理人为其投票 通过检查代理线程的状态词。回想一下， 在将新消息发布到与代理关联的队列时递增。操作顺序为:1)读取 ;2)从队列中读取消息;3)做出调度决策;4)发送 在 TXNS_COMMIT()的事务旁边。如果 与事务一起发送的数据比当前的要旧 由内核观察(即;一个新的消息被发布到代理的队列中)，该事务被认为是“陈旧的”，并将失败，并出现 ESTALE 错误。然后代理耗尽它的队列来检索更新的消息，并重复这个过程。

使用 BPF 加速调度。ghOSt 提供的用户级灵活性并不是免费的:消息传递和组调度最多需要 5 个 s (见§4.1 中的表 3);在集中式调度模型中，线程可能会等待整个集中式调度循环，直到代表它提交调度决策(30 . s 在§4.4)。

鬼允许通过自定义 BPF 程序恢复丢失的 CPU 时间，该程序由代理附加到内核的 pick_next_task()函数。当 CPU 处于空闲状态，并且代理还没有发出事务时，BPF 程序就会发出自己的事务，并选择一个线程在该 CPU 上运行。BPF 程序与代理通信通过共享内存窗口进入代理的地址空间。代理如何使用 BPF 基础设施在 cpu 上调度线程的细节是调度策略的一部分。鬼 BPF 程序本质上是代理本身的扩展，因此 BPF 字节码被嵌入到代理二进制中，使用 libbpf[69]。

3.3 集中式调度器(Centralized Scheduler)

我们现在解释构建集中式调度鬼影策略所需的其他实现细节。

具有单个队列的全局代理。对于集中式调度，一个全局代理轮询一个消息队列，并为鬼下管理的所有 cpu 做出调度决策。如果指定的 CPU 已经运行了一个鬼线程，事务将抢占前一个线程，以优先于新线程。图 4 描述了一个简化的调度器代码示例。直观地看，集中式策略似乎无法支持 s-scale 调度，虽然我们在§4 中展示了鬼在我们的生产工作负载上有相当或更好的整体性能。避免抢占全局代理。支持 s-scale 调度，全局代理必须持续运行，因为任何抢占都会直接导致调度延迟。为了防止更高优先级的内核调度类抢占全局代理，鬼给所有代理分配一个高的内核优先级，类似于实时调度。换句话说，机器中的任何其他线程，无论是鬼还是非鬼，都不能抢占代理线程。然而，这种优先级分配，除非小心处理，否则会破坏系统的稳定。例如，大多数系统都有必须在其指定的 cpu 上运行的每 cpu 守护进程工作线程。

鬼通过以下方式来维护系统的稳定性。所有不活动的代理立即退出，腾出它们的 cpu。当一个非鬼线程需要在全局代理的 CPU 上运行时， ，全局代理执行“热切换”到另一个 CPU 上的非活动代理， 。例如，如果内核 CFS 调度器试图调度一个线程 ，全局代理将首先找到一个空闲的 CPU( )，然后唤醒不活跃的代理 作为新的全局代理。一次 运行全局代理，旧的全局代理产生，允许 CFS 线程在其上运行 。

序号和集中调度。在某些时候，全局代理对线程状态的视图可能不一致。例如，一个线程 可能会发布一条 THREAD_WAKEUP 消息。全局代理接收到这条消息并决定调度 在 。同时，系统中的某些实体调用 sched_setaffinity()，导致 THREAD_AFFINITY 消息，禁止停止运行 。我们需要一种机制来确保交易按时进行在 将会失败。

原则上，我们可以使用代理序列号，如上面的每 cpu 示例所述。然而，全局的

Linux CFS(kernel/sched/fair.c)	6217 LOC
新宿[25](NSDI'19)	3900 LOC
Shenango [1] (NSDI'19)	13161 LOC
鬼调度类鬼用户空间支持库	3,777 loc 3,115 loc .
新宿政策(§4.2)新宿+ Shenango政策(§4.2)	710 loc 727 loc .
谷歌 Snap Policy(§4.3)	855 LOC
谷歌搜索策略(§4.4)	929 LOC
安全虚拟机内核策略(§4.5)安全虚拟机鬼策略(§4.5)	7,164 loc 4,702 loc .

表 2。鬼和比较系统的代码行。

Agent 必须支持成千上万个不断向全局队列发布消息的线程，这使得耗尽队列非常耗时。与每 cpu 示例中的本地代理不同，全局代理不会放弃自己的 CPU。全局代理必须只验证它相对于线程是最新的 正在被调度。我们用线程序列号来解决这个问题。回想一下每个排队的消息 是否标记了线程序列号 (,)。当代理为线程 提交事务时，它将事务连同最近的序列号一起发送 它意识到: 。当内核接收到事务时，它会验证 相对于事务中的线程来说是最新的。否则，事务失败，报错 ESTALE。

3.4 故障隔离和动态升级

与其他内核调度类的交互。ghOSt 的设计目标之一是使现有系统易于采用。因此，即使 ghOSt 策略出错，我们仍然希望鬼管理的线程能够与系统中的其他线程很好地交互。我们希望避免 ghOSt 线程对其他线程造成意想不到的后果，例如饥饿、优先级反转、死锁等。

我们通过在内核的调度类层次结构中为 ghOSt 的内核调度器类分配一个低于默认调度器类(通常是 CFS)的优先级(§2)来实现这一目标。结果是系统中的大多数线程将抢占 ghOSt 线程。ghOSt 的抢占会导致创建 THREAD_PREEMPT 消息，从而触发相关代理(在不同的高优先级调度类中运行)做出调度决策。代理会进一步决定如何处理抢占。

动态升级和回滚。ghOSt 支持快速部署，因为更新了调度策略(即:代理程序)不需要重新启动内核或应用程序。许多生产服务可能需要几分钟到几个小时才能启动，特别是填充内存缓存。同样，我们希望尽量减少对客户端虚拟机的中断。这些长期运行的应用程序在计划中的代理更新或计划外的代理崩溃期间继续正确运行。ghOSt通过(a)替换代理在保持飞地基础设施完整的同时，或者(b)摧毁飞地并从头开始。

替换特工，摧毁飞地。ghOSt 支持在不破坏飞地的情况下“就地”更新代理。用户空间代码可以查询代理是否附属于飞地，并对其进行 epoll。为了升级代理，我们同时运行旧代理和新代理;新代理会阻塞，直到旧代理崩溃或退出，不再连接。新代理从内核中提取 enclave 中所有线程的状态，并恢复调度。如果这个过程失败，内核或用户空间代码都可以破坏这个 enclave。摧毁飞地杀死飞地中的所有代理，保持系统中其他飞地的完整，并自动将被摧毁飞地中的所有线程移回 CFS。此时，线程仍然正常工作，但由 CFS 而不是 ghOSt 调度。

ghOSt 看门狗。ghOSt 或任何其他内核调度器中的调度错误都会对整个系统产生影响。例如，一个 ghOSt 线程可能会在持有一个内核互斥时被抢占，如果它没有被调度太长时间，它可能会传递地暂停其他线程，包括那些在 CFS 或其他 ghOSt 飞地中的线程。类似地，如果垃圾回收器和 I/O 轮询器等关键线程没有被调度，机器就会慢慢停止。作为一种安全机制，《ghOSt》会自动摧毁行为不端的飞地。例如，当内核检测到代理在用户可配置的毫秒数内没有调度可运行线程时，就会销毁一个 enclave。

4 评价

我们对 ghOSt 的评估主要集中在三个问题上:(a)在经典调度器中不存在的特定于 ghOSt 的操作的开销是什么(§4.1);(b)与之前的工作(如新宿[25])相比，ghOSt 实施的调度政策表现如何(§4.2);(c)ghOSt 是大规模低延迟生产负载(包括谷歌 Snap(§4.3)、谷歌搜索(§4.4)和虚拟机(§4.5)的可行解决方案吗?

4.1ghOSt 开销和缩放分析

代码行数:表 2 列出了 ghOSt 和相关工作(如 Linux CFS 调度器)的代码行数(LOC)。ghOSt 可用于生产，并灵活地支持一系列生产工作负载的调度策略，内核代码比 CFS 少 40%。本节中的策略可以很短(只有 100 个 LOC)，因为它们使用了用户空间库中的公共函数。这反映了在高级语言中进行策略定义的优势:更灵活的抽象，使复杂性集中在调度决策上。

实验设置:除非特别说明，实验运行在 Linux 4.15 上，应用了我们的 ghOSt 补丁。我们在 2 插座英特尔至强白金 8173M @ 2GHz 上运行微基准测试，每个插座 28 核，每个逻辑核 2 个。

1.向本地座席发送消息。消息传递到 全局代理	725 ns 265 ns
3.当地时间表(1 txn)	888 纳秒
远程调度(1 个 CPU 对应 1 个 txn)代理 开销 5.;目标 CPU 开销端到端延迟	668 ns 1064 ns 1772 ns
分组远程调度(10 个 cpu 对应 10 个 txn)代理开销 8.;目标 CPU 开销端到端延迟	3964 ns 1821 ns 5688 ns
10.系统调用开销 pthread 最小上下文切换开销 CFS 上下文切换开销	72 ns 410 ns 599 ns

表 3。ghOSt 微基准。由于 agent 和目标并行工作且 IPI 通过系统总线传播，端到端延迟不等于 agent 和目标开销的总和。

表 3 总结了 ghOSt 特有的基本操作的开销。我们还报告了 CFS 下等效线程操作的开销。

消息传递开销(第 1-2 行)。在 per-CPU 示例中，向本地代理的传递包括将消息添加到队列、上下文切换到本地代理以及将消息退出队列。开销(725 ns)主要由上下文切换(410 ns)所支配。在集中式示例中，传递到全局代理(265 ns)包括将消息添加到队列中，并在始终旋转的全局代理内将消息解队列。

本地调度(第 3 行)。在每 CPU 模型中，这是提交事务和在本地 CPU 上执行上下文切换的开销，直到目标线程运行为止。由于事务提交，开销(888 ns)略高于 CFS 上下文切换开销(599 ns)，但仍然具有竞争力。

远程调度(第 4-9 行)。在集中式调度模型中，代理端提交事务并发送 IPI 中断(inter-processor interrupt)。目标 CPU 处理 IPI 并进行上下文切换。代理的开销(668 ns)将每个代理的理论最大吞吐量设置为 109/668 = 1.5M 调度线程/秒。通过摊余 IPI 开销，将不同 cpu 的 10 个事务分组，理论上可以将调度线程的最大值提高到 10 * 109/3964 = 2.52M / s。

给定这些数字，理论上单个代理可以为 100 台 CPU 服务器每秒调度大约 25,200 个线程。如果线程长度为 40⁺ s，代理可以保持 100 个 cpu 忙碌。策略开发人员在设计 ghOSt 策略时应该记住每个代理的可伸缩性限制。随着代理数量的增加，这个限制会相对线性地提高。

全局代理可伸缩性(图 5)。为了展示全局代理是如何扩展的，我们分析了一个简单的轮询策略。该策略管理 FIFO 运行队列中的所有线程，一旦 cpu 空闲，就在 cpu 上调度它们。代理在每次提交时将尽可能多的事务分组。我们跑

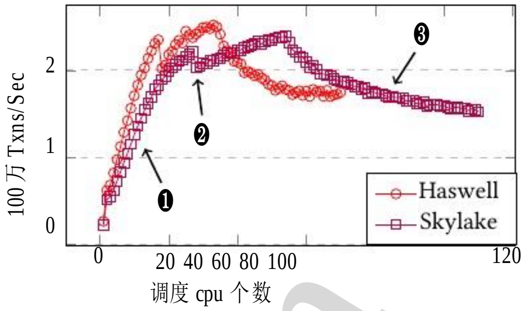


图 5。全局代理的可伸缩性。实验在使用 Skylake 处理器的默认 microbenchmark 机器上进行，以及使用 Haswell 处理器的 2 插槽机器上进行(每个插槽 18 个物理核，每个插槽 2 个逻辑核，2.3 GHz)。

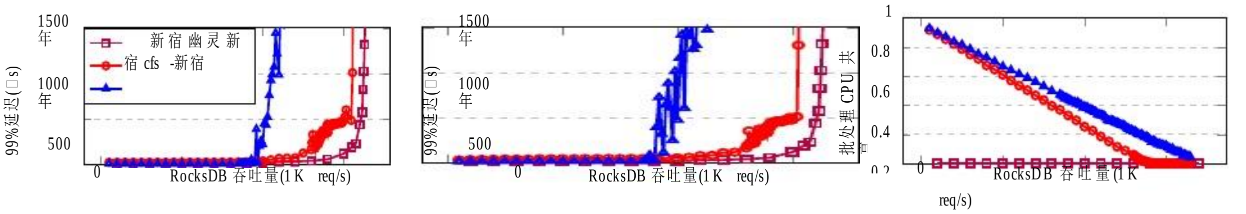
实验结果如图 5 所示。这两行都遵循类似的模式，我们在图中注释了 Skylake 线:陡峭的 ramp-up 注释:当有更多的 cpu 可用来调度工作时，全局代理每秒调度更多的事务。当我们将全局代理作为执行工作的 ghOSt 线程共同定位在同一个物理核上时，就会发生丢弃的情况。超线程在物理核心的管道中争夺资源，这降低了全局代理的性能。最后，性能下降的原因来自于全局代理在远程套接字中调度 cpu。调度这些 cpu 需要内存操作和跨 NUMA 套接字发送中断，从而产生更高的开销。

我们现在将 ghOSt(一种通用的调度框架)与最近学术研究中使用集中式策略来调度需求的高度专业化的调度系统进行比较 s-scale 工作负载。实验在 Intel Xeon CPU E5-2658 的单插槽上运行(每个插槽 12 核，每个插槽 24 逻辑核，2.2 GHz)。

对比下的系统。我们在新宿[25]中比较了该调度方法的三种实现，它们都服务于 RocksDB 工作负载[70]。我们使用一个物理核心对所有系统进行负载生成。新宿系统运行在 Linux 4.4 上，因为它的沙丘[35]驱动程序无法为新版本编译。所比较的其他系统(ghOSt- shinjuku 和 CFS-Shinjuku)运行在 Linux 4.15 上，并应用了我们的 ghOSt 补丁。

(1)使用原新宿系统[25]。它使用 20 个旋转的工作线程固定在 20 个不同的超线程和一个旋转的调度线程上，运行在一个专用的物理核心上。旋转的线程阻止任何其他线程在其 cpu 上运行(图 6c)。dispatcher 在 FIFO 中管理到达的请求，并将它们分配给工作线程。每个请求在被抢占并添加到 FIFO 后面之前，都会运行到一个有限的运行时。

(2)在幽灵中，利用集中式调度模型在 710 行用户空间中实现了新宿调度策略



(a)分散负载的尾部延迟。 (b)RocksDB 与批处理应用程序同处。 (c)批处理应用程序的 CPU 份

图 6。幽灵实现现代 s-scale 抢占策略和共享 CPU 资源，而不损害尾部延迟。

代码。幽灵新宿全球代理开始拥有物理核心但可以自由移动(§3.3)。我们保持一个由 200 个工作线程组成的线程池，负载生成器将其作为 - 将请求签名到。全局代理维护一个 FIFO 队列的可运行工作线程，并将它们调度到重新维护 20 个 cpu。注意，幽灵允许其他负载在系统使用任何空转 cpu，如图 6b-c 所示。

(3) 作为参考，我们还实现了非抢占在 Linux CFS 上运行的新宿版本。这个 CFS-新宿版本不受益于新宿的特殊 - 化了数据平面的功能，比如虚拟化的使用特性和用于抢占的发布中断。

单个工作负载比较。就像新宿 [25] 一样 Paper，我们生成一个工作负载，其中每个请求在 - 包含对内存中 RocksDB 键值的 GET 查询 Store[70](大约 6 . s) 并执行少量 pro- 消费类电子展。我们分配了以下处理时间:99.5% 请求的 - 4 s, 0.5% 的请求 - 10ms。分配的时间 - 切片每个工作线程，在强制抢占和回到 FIFO，是 30 s. cfs - 新宿不是先发制人，所以所有请求都运行到完成。

我们的结果如图 6a 所示。《幽灵》很有竞争力新宿是 s-scale 尾工作量，即使它新宿策略的实现减少了 82% 的代码行数比自定义新宿数据平面系统。幽灵高负载时尾延迟略高于新宿距离新宿饱和吞吐量不到 5%。不同 - ence 反映了幽灵在调度 a 时的额外开销线程处理每个请求，而新宿通过请求旋转线程之间的描述符。cfs - 新宿 satu 速率比其他两个系统快 30% 左右，原因是它缺乏抢占。

多工作负载比较。在生产环境中 nario，当 RocksDB 负载较低时，它很有吸引力闲置计算资源来服务低优先级的批处理应用程序 tion[1,21,71] 或运行无服务器函数。最初的 Shin-Juku 系统调度请求，不能管理其他的本机线程。图 6c 显示了当我们共定位一个批应用程序，RocksDB 工作负载由新宿管理，批处理应用程序甚至不能获得任何 CPU 资源当 RocksDB 负载很低时。

Shenango 的集中调度程序监视网络的负载工作应用程序，当应用程序处于轻负载时，调度器将空闲的 CPU 周期提供给批处理应用程序。Shenango 不适合执行变化的请求次数，所以 RocksDB 的工作负载会更糟糕尾部延迟比新宿的延迟多。

我们扩大了幽灵新宿政策的实施范围 shenango 风格的调度，只有 17 行代码，使该政策总共达到 727 行 (新宿 + 表 2 中的 Shenango 策略)。该策略监视负载到 RocksDB，并将空闲周期给批处理应用程序表明我们修改后的幽灵策略产生了相同的尾巴延迟作为我们最初的幽灵政策。主要的好处是如图 6c 所示。同时保持 RocksDB 的尾延迟完好无损，幽灵现在与批量共享备用 CPU 周期批处理应用程序可以利用的计算量是类似于它在 CFS 下运行时可以达到的效果 nice 值为 19，而 RocksDB 的 nice 值为 -20。一个《幽灵》中的几行代码结合了新宿 [25] 的优点和 Shenango[1]，没有任何应用程序更改。

4.3 谷歌 Snap

我们现在评估幽灵作为我们的软现实 - 的替代品 time 内核调度器 MicroQuanta[2]，它管理 Snap[2] 的工作线程，我们的用户空间数据包处理框架。

类似 DPDK [72]， Snap 维护轮询 (worker) 负责与 NIC hard- 交互的线程 Ware 和运行自定义网络和安全 pro- Tocols 代表重要服务。Snap 可能会决定随着网络负载的变化，衍生/加入工作线程。

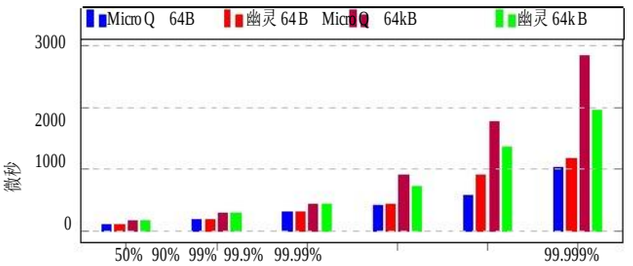
今天的工作线程是如何安排的？吸附主要 - 容纳至少一个工作线程不断轮询。随着爆发大量的网络负荷到来时，Snap 可能会醒来并失去控制 Quently 让其他工作线程进入睡眠状态。这些主要 Quent 唤醒 / 睡眠需要快速调度程序的干预避免增加延迟。试图保证低延迟通过现有的实时调度器，如 SCHED_FIFO、destabi- lize 系统，因为它可能会饿死其他应用程序在同一台机器。因此，我们在生产中部署 Micro- Quanta，一个定制的软实时调度器，保证对于任何一段时间，例如 1 毫秒，最多是一个时间量子，例如，

其他线程。然而，它也会导致高达 0.1 毫秒的网络瘫痪。

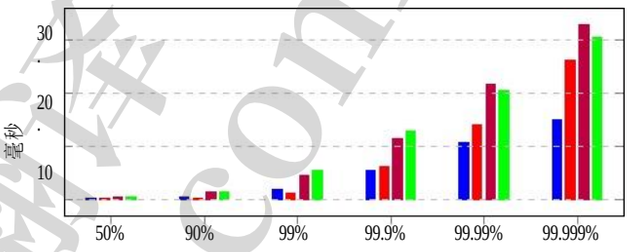
实验设置。我们使用了两台机器，每台机器都有两个 Intel Xeon Platinum 8173M 处理器(每个插槽 28 个物理核，每个插槽 2 个逻辑核，2GHz)、383gb DRAM、100Gbps NIC 和匹配的开关。我们的测试使用了单个插座，即。，每台机器有 56 个逻辑 cpu。

测试工作量。我们的测试工作负载由 6 个客户端线程组成，向另一台机器上的 6 个服务器线程发送 10k 条消息/秒，并接收一个大小对称的回复。该测试旨在强调线程调度，而不是 NIC。一个客户端线程发送 64 字节的消息，这代表了最坏的情况，而不是现实的负载。其他 5 个客户机线程都发送 64kB 的消息。在所有情况下实现的总带宽为 51.86Gbps。在所有实验中，客户端线程和服务器线程都使用 Linux CFS 进行调度。在幽灵实验中，工作线程使用幽灵而不是微量子进行调度。我们以两种模式运行测试。在安静模式下，客户端/服务器线程是它们机器上唯一显式的工作负载。在加载模式下，机器还运行批处理工作负载的另外 40 个拮抗线程，当网络流量较低时，这些线程试图使用空闲的 CPU 资源，这些资源被服务器或 Snap 线程使用。

幽灵政策。我们将微量子替换为以下幽灵策略。该策略管理 Snap 的工作线程和对抗线程。这是一个简单而有效的集中式 FIFO 策略。全局代理试图找到一个空闲的 CPU 来调度它的线程，给 Snap worker 线程比拮抗线程严格的优先级。在安静模式测试中，工作线程经常被客户机/服务器线程和 CFS 调度的其他本机线程(例如重要但周期性的守护进程)抢占。在加载模式测试中，Snap 工作线程将抢占拮抗线程，但不能抢占由 CFS 管理的线程。拮抗线程只在 CFS 线程和 Snap 线程都有空闲资源时运行。我们没有使用任何专用的内核。尾延迟比较。图 7 比较了机器中给定两个调度器的客户端请求的往返尾延迟。我们分别给出了 64B 和 64kB 消息的尾部延迟。对于 64B 消息，当我们考虑高达 99.9 百分位数的延迟时，幽灵的性能与基线相似或比基线提高 10%。在 99.99%以上的情况下，幽灵的延迟要严重 1.7 倍。64B 消息需要很少的计算来进行分组处理。因此，当突发流量中包含大量 64B 消息时，幽灵调度事件的开销是显著的。对于 64kB 的消息，对于第 99 个百分位数的延迟(在任何方向上都在 15%以内)，幽灵的性能与基线相似。对于 99.9 百分位数及以上，幽灵导致尾部延迟降低 5%至 30%。64kB 的消息需要更多的处理(用于复制数据)，因此导致更少的调度事件。在某些情况下，幽灵比微量子性能更好的原因是，当 CPU 变成时，它可以重新定位工作线程



(a)仅组网负载(静音测试)。



(b)带附加载荷(加载试验)。

图 7。64B 和 64kB 消息的谷歌 Snap 延迟。由于服务器进程线程繁忙。另一方面，微量子必须等待停电期。

这些结果非常令人鼓舞。一个非常简单的 ghOSt 策略的执行与自定义内核调度器类似，而不需要修改 Snap。ghOSt 允许快速实验和性能优化，这在内核调度器中要难得多。我们期待额外的改进，比如包括来自工作线程的调度提示，可以进一步优化性能。

4.4 谷歌搜索

我们现在评估 ghOSt 作为在提供谷歌搜索查询的机器上 CFS 调度器的替代品[73]。

测试工作负载。该基准测试包括一个在独立机器上运行的查询生成器(没有 ghOSt)，发送三种不同的查询类型，分别表示为 a、B 和 c。查询类型 a 是一个 CPU 和内存密集型查询，由工作线程提供服务，需要时唤醒这些线程。查询类型 B 需要很少的计算，但需要访问 SSD，并由一组短期工作线程提供服务，也会在需要时被唤醒。查询类型 C 是 cpu 密集型负载，由长寿的工作线程提供服务。

在数据包进入时，所有查询首先由几个服务器线程中的一个处理，这些服务器线程创建子查询，由上面引用的工作线程处理。一些子查询必须由绑定到 NUMA 节点的特定工作线程处理，以利用数据局部性。由于此工作负载受内存和 CPU 限制，如果查询由在它们访问的数据所在的套接字上运行的工作线程处理，那么查询的服务速度会快得多。

实验设置。我们评估 ghOSt 的机器有两个 AMD Zen Rome 处理器，总共 256 个 CPU(2 个插槽，每个插槽 64 个物理核，2 个逻辑核)

每个)。AMD 的架构带来了新的挑战，因为它将 4 个物理核(8 个逻辑核)集群成 CPU 核 complex (CCX)，每个 CCX 有自己的 L3 cache。

ghOSt政策。利用 ghOSt 的集中式调度模型和一个全局代理实现了对所有 256 个 CPU 的调度策略。在启动时，全局代理首先生成一个系统拓扑的模型，使用 sysfs。然后，全局代理使用拓扑信息根据其 NUMA 首选项来调度线程，并且在可能的情况下，优先在属于线程所运行的最后一个 CCX 的 CPU 上运行线程。全局代理维护一个按线程运行时排序的 min-heap，其中运行时最少的线程会在其他线程之前被选中执行。线程运行到完成或被 CFS 线程抢占为止。

由于 ghOSt 灵活的事务 API 和 c++标准库的使用，用于选择下一个空闲 CPU 的 NUMA 和 ccx 敏感的启发式代码只有 57 行。当产生一个新的工作线程时，它的 cpumask(通过 sched_setffinity())被设置为它的查询数据所在的套接字中的 CPU 集。这个 cpumask 被包含在发送给全局代理的 THREAD_CREATED 消息中。当 ghOSt 全局代理想要运行其运行队列前面的下一个线程时，它将线程的 cpumask 与空闲 CPU 集相交。如果交集为空，代理会跳过该线程并调度运行队列中的下一个线程，在其调度循环的下一次迭代中重新访问被跳过的线程。

当每个线程在具有预热 L1、L2 或 L3 缓存的 CPU 上运行时，搜索查询的性能会提高。对于每个线程调度事件，我们的 ghOSt 策略将线程分配给一个空闲的 CPU 目标，这个目标最接近线程最后运行的地方。该策略首先在线程最近运行的 CPU 的 L1 和 L2 缓存域中搜索可用的 CPU。如果没有发现空闲的 CPU，该策略将搜索扩展到 CCX (L3 cache)域。如果这也失败了，那么它会对线程上次运行的 CCX 的最近邻居进行扇出搜索，以避免由于 CCX 之间的高通信延迟而导致的昂贵的线程迁移成本。CFS vs.ghOSt。图 8 比较了使用 CFS 和 ghOSt 策略的基准搜索在 60 秒内的规范化查询延迟和吞吐量。图 8a-c 显示，ghOSt 提供了与 CFS 相当的吞吐量。CFS 和 ghOSt 都考虑 NUMA 插座和 CCX 放置。NUMA 和 CCX 优化在实现与 CFS 的均等方面至关重要，因为它们分别提供了 27%和 10%的吞吐量改进。在短 ghOSt 策略中迭代优化 NUMA 和 CCX 放置比在内核 CFS 代码中试验更改要容易得多。对 ghOSt 代理的每次修改只需要重新启动代理进程，而对 CFS 的任何修改都将要求安装内核并重新启动。

尾巴延迟。由图 8d-f 可知，对于查询类型 A 和 B，ghOSt 使得尾部延迟降低了约 40-45%，与 CFS 和类似的查询类型 C 的尾延迟相比，在 socket 和 ccx 感知的优化之前，ghOSt 策略导致了查询类型 A 的近 2 倍的延迟，并且在查询类型 B 和 C(即:查询类型 C)上与 CFS 相当。，在 10%之内)。查询类型 A 受内存限制，从拓扑优化中受益最多。查询类型 B 同时访问内存和 SSD，而查询类型 C 主要是计算绑定的。对于 B 和 C，ghOSt 策略从一开始就有优势，因为全局代理旋转并对整个系统容量的变化做出快速反应，以微秒量级在 cpu 之间重新平衡线程。另一方面，CFS 只会以毫秒量级的周期间隔在 cpu 之间重新平衡线程，从而损害查询尾延迟。

我们可以通过进一步细化策略来提高查询 C 在 ghOSt 上的延迟。工作负载将 nice 值分配给线程，以表示对 CFS 的相对优先级排序，这对于确保工作线程以比低优先级后台线程更高的优先级运行(例如，用于垃圾回收)很重要。CFS 通过使用这些良好的值做出了更优的决策，初步实验表明，将它们与 ghOSt 的策略相结合，可以使 ghOSt 在查询 C 的尾部延迟方面优于 CFS。

我们在快速实验方面的经验。ghOSt 是我们生产车队的可行解决方案，能够调度大型机器和现实的工作负载，同时支持快速开发和推出调度策略。

在开发内核调度器时，写-测试-写周期包括(a)编译内核(最多 15 分钟)，(b)部署内核(10-20 分钟)，以及(c)运行测试(重启后由于数据库初始化需要 1 小时)。因此，热情的内核开发人员每天要试验 5 个变种。使用 ghOSt，编译、部署和启动新代理可以在一分钟内轻松完成。事实上，由于 ghOSt 能够在不重启的情况下升级，测试继续不间断地运行。这也有一个重要的次要影响，因为这个测试的规模和复杂性可以在重启期间贡献非微不足道的运行-运行方差，混淆优化开发。

这种实验的便便性使我们能够试验定制的优化，否则很难发现。例如，由于在 Rome 架构中，CCX 内部和 CCX 之间的延迟存在很高的方差，我们发现，如果一个线程的首选 CCX 集群不可用，那么暂时保持线程挂起 100 会更有效 s 而不是立即将其迁移到另一个 CCX。

4.5 保护虚拟机免受 L1TF/MDS 攻击

我们现在评估了一种保护虚拟机免受跨超线程推测执行攻击(如 L1TF 和 MDS[27-32])的 ghOSt 策略。在这些攻击中，恶意 VM 利用微架构缺陷从运行在同级超线程上的不同 VM 窃取数据。通过确保每个物理核心只运行虚拟，可以缓解攻击

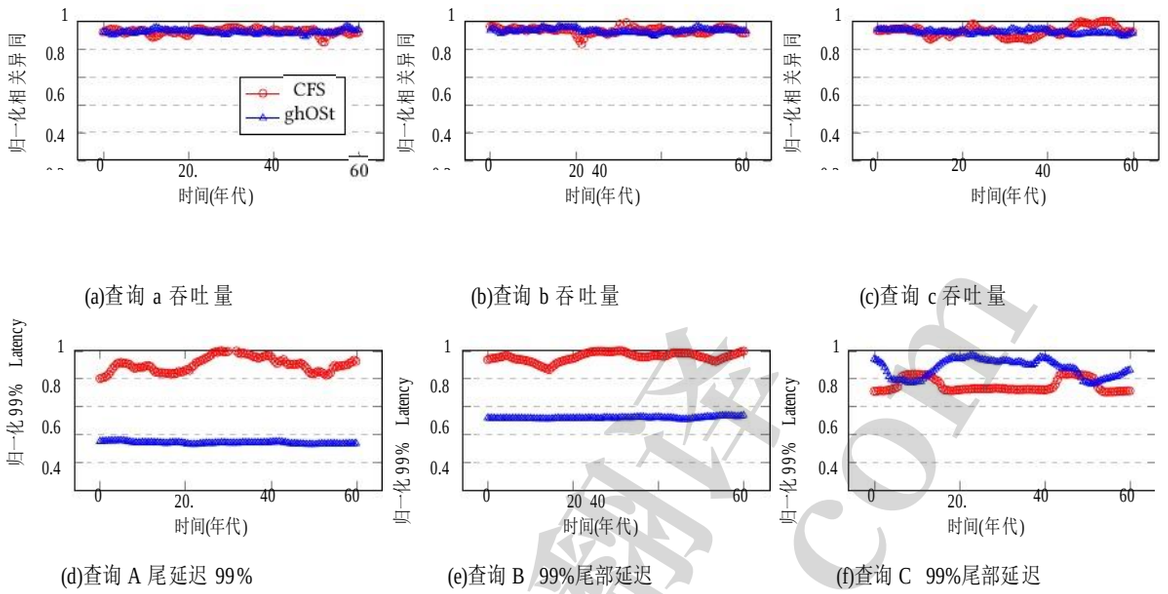


图 8。使用 CFS 和 ghOSt 调度的谷歌基准搜索结果。

同一虚拟机的 cpu 个数。当一个新的 VM 被调度到物理核心上时，微架构缓冲区会被刷新。

减轻交叉超线程攻击需要调度物理核，每个物理核都有两个逻辑 cpu。在每 CPU 模型中实现核心调度具有挑战性，因为调度器代码只能在其当前执行的 CPU 上运行线程。相比之下，ghOSt 代理可以通过对每个物理核执行同步的组提交来轻松地调度整个核。，向一个核心的两个 cpu 发出提交，要么全部成功，要么全部失败。

我们的每核调度模型如图 9 所示。在每个核中，两个相邻的 cpu 将消息发送到同一个队列。在任何时候，在给定的物理核心上只有一个代理处于活动状态。生成该消息的 CPU 唤醒其对应的代理为活动代理;另一个代理在必要时变为非活动。活动代理为两个 cpu 做出调度决策，并以一种队列中没有等待消息的方式提交一个组事务，如§3.2 所解释的那样。在每物理核调度模型中，ghOSt 策略确保在物理核上运行的线程属于同一个虚拟机。VM 的线程可能占用(a)两个兄弟线程，或者(b)只占用一个兄弟线程，而另一个兄弟线程运行一个空闲线程。

安全虚拟机核心调度策略。在内核和 ghOSt 中实现了一种类似于 Tableau[23]的虚拟机调度策略。我们设计了这种策略，以保证前进的进度，绑定尾部延迟，并在所有虚拟机之间提供良好的平均延迟。该策略通过每个周期 p 为每个可运行的 VM 线程调度 c 个时间单位来确保前两者。具体来说，我们使用分区 EDF 方案，其中每个物理核心为每个线程的运行分配有保证的时间，限制尾延迟。任何多余的时间都在可运行线程之间公平地共享，提高了平均延迟。运行队列跨单个 NUMA 节点;当一个物理内核处于空闲状态并寻找一个新的线程来运行时，它喜欢在 NUMA-local 运行队列中选择一个线程。然而，在高系统负载下，该策略允许跨运行队列溢出，提供了 NUMA 优先级，而没有 CPU 亲和掩码强加的硬边界。

评估。实验设置与§4.3 相同。我们在 25 个物理核和 50 个逻辑 cpu 上调度 32 个 vcpu。在 SPECCPU 2006 的 bwaves 测试中，我们使用了三种调度策略:1)CFS，对推测执行攻击没有提供安全性;2)核内安全虚拟机核心调度;3)ghOSt 安全虚拟机核心调度。结果如表 4 所示。CFS 提供了更好的整体性能，但没有安全性。ghOSt 策略减少了跨超线程攻击，执行与内核版本的核心调度相似，即使考虑到 ghOSt 中额外的上下文切换开销。

5 未来的工作

利用 BPF 加速调度。通过将 agent 的部分职责委托给同步 BPF 回调来加速 ghOSt 是一个开放的研究领域。§4.4 中的全局代理调度循环需要 30 s, 产生潜在的调度间隙。事实上，我们系统中的一些线程只运行 5-30 个线程 s 在它们阻塞之前，在这些间隙期间让 cpu 空闲。我们可以使用集成的 BPF 程序(见§3.2)来减少这些调度差距。

BPF 程序通过共享内存与多个多生产者、多消费者环缓冲器与用户空间进行通信。代理将可运行线程插入缓冲区，BPF 尝试运行它们。代理可以在 BPF 调度线程之前撤销线程。例如，全局代理可以为每个 NUMA 节点使用一个环形缓冲区;然后，全局代理可以跟踪每个线程的首选 NUMA 节点，并在两个环之间对线程进行负载平衡。

Tick-less 调度。当 ghOSt 处于集中式模式时，可以在 cpu 之间禁用计时器滴答，以避免虚拟机工作负载中昂贵的虚拟机退出。在一个典型的每 cpu

调度策略	bwaves 率	总时间
CFS(无安全性)	489	888 秒
内核内核心调度	464	937 秒
ghOSt 核心调度	468	929 秒

表 4。安全虚拟机核心调度性能。CPU 2006 bwaves，在 50 个真实/逻辑 cpu 上调度 32 个 vcpu。速率-越高越好。时间越低越好。调度器，ticks 每毫秒触发调度器，以确保所有虚拟机之间的轮询抢占。不幸的是，这些 tick 会导致 vm 退出到宿主内核上下文。由于全局代理在不断地旋转并做出调度决策，因此不需要这些 tick。消除所有 cpu 上的这些节拍将大大减少客户的抖动。这种类型的优化在 CFS 中不可能实现。CFS 中最接近的选项是启用 CONFIG_NO_HZ_FULL，但这只会在给定 CPU 上只有一个可运行线程的情况下禁用滴答，而在高利用率的情况下通常不是这样。

6 相关工作

我们简要讨论与 ghOSt 相关的其他先前工作。调度器激活和用户级线程。调度器激活[74]为单个应用程序提供了一个 API，以协调内核分配给它的 CPU 的调度。与 ghOSt 相比，调度器激活允许应用程序对 CPU 的分配或删除做出反应，但不允许将 CPU 分配给应用程序。人们可以使用激活使应用程序与 ghOSt 调度决策同步。类似地，用户空间线程库和调度器[13,22,48,52-57,59,60,75]在内核线程之上复用用户空间上下文，但不能控制内核线程运行的时间和位置。

智能网卡调度;最近的工作探索了将调度和应用程序从主机卸载到智能网卡的正确策略和机制[19,76 - 78]。ghOSt 的共享内存队列和事务 api 被设计为与新的连贯互连技术(如 CXL[79])无缝工作，允许 ghOSt 部分或全部卸载到智能网卡。

微内核的返回。一个新兴的趋势是将内核组件卸载到用户空间，类似于 microker- nels[80]。DPDK[72]、IX[34]和 Snap[2]将网络驱动程序和堆栈卸载到用户空间。SPDK [81]， ReFlex [82]， FUSE[83]，和 DashFS[63]卸载存储和文件系统操作。Linux 用户空间 I/O (UIO)[84]有助于卸载内核驱动程序。甚至有人提出了一种新的 CPU 设计来加速微内核[85]。ghOSt 延续了这一趋势。之前的工作也建议将调度器移动到用户空间。Stoess 为 L4 微内核开发了一个层次化的用户级调度器[86]。然而，与 ghOSt 不同，Stoess 需要修改应用程序来实现自定义的调度策略。

CPU 继承调度。Ford 和 Susarla 提出的 CPU 继承调度[87]是一种用户级调度系统

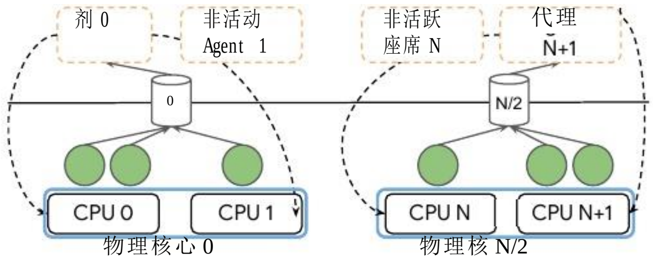


图 9。每核 ghOSt 调度。每一对兄妹-CPU 共享一个消息队列。对于每个核，只有一个

单 agent 一次激活。

其中调度是由线程提供它们的运行时完成的。与 ghOSt 的交易相比，捐赠模型效率较低，表达能力也较差。每个 CPU 的根调度器将把它的运行时间捐赠给另一个线程。这个线程也可以是一个调度器，允许调度器的层次结构，或者是一个应用程序线程——优雅地解决优先级反转。与 ghOSt 代理一样，这些调度程序接收关于线程和系统事件的消息。系统的调度程序，类似于 ghOSt 内核代码，处理机制:执行捐赠和发送消息。捐赠类似于运行本地 CPU 的 ghOSt 事务，但它不能快速调度远程 CPU。要做到这一点，它必须唤醒远程 CPU 上的调度程序，并让它捐赠。此外，与 ghOSt 不同，每个调度器一次只能调度一个 CPU。

7 结论

本文提出了 ghOSt，一个用于评估和实现现代数据中心线程调度策略的新平台。ghOSt 将调度器开发从单一内核实现领域转换为更灵活的用户空间设置，允许应用广泛的编程语言和库。虽然直观地说，将调度决策转移到用户空间可能会带来过多的协调开销，但我们已经在 api 中最小化了同步成本，我们的特征表明大多数操作都是 circa s. 以前需要花费大量精力来优化和部署的事情，现在通常只需要不到一千行代码就可以实现。ghOSt 允许我们快速为生产软件开发策略-快速测试和迭代-与现状调度程序相比，导致具有竞争力的性能。我们将 ghOSt 开源，作为未来用户驱动资源管理新时代的研究方向。

致谢

我们感谢 Eric Brewer、大卫 Culler、Hank Levy、Amin Vahdat、Kostis Kaffes、Adam Belay、Josh Fried、大卫 Mazieres、我们的 shepherd Irene Zhang、谷歌系统基础设施、斯坦福平台实验室和匿名 SOSP 评审员的有用反馈。Christos Kozyrakis 得到了斯坦福平台实验室的支持。

参考文献

Amy Ousterhout, Joshua Fried, Jonathan Behrens，Adam Belay 和 Hari Balakrishnan。Shenango:为延迟敏感的数据中心工作负载实现高 CPU 效率。第 16 届 USENIX 网络系统设计和实现研讨会(NSDI 19)，第 361-378 页，波士顿，MA，2019 年 2 月。USENIX 协会。

Snap:一种微核方式的主机联网。在 ACM SIGOPS 第 27 届操作系统原理研讨会上，纽约，NY，USA，2019。

[3] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, Kunle Olukotun。弹性 rss:使用可编程网卡共同调度数据包和核心。《第三届亚太网络研讨会论文集 2019》，APNet ' 19，第 71-77 页，纽约，NY，USA，2019 年。计算机协会。

[4] Amy Ousterhout, Adam Belay, Irene Zhang。及时交付:利用操作系统知识更好地控制数据中心拥塞。第十一届 USENIX 云计算热点研讨会(HotCloud 19)，WA 伦顿，2019 年 7 月。USENIX 协会。

[5] Esmail Asyabi, Azer Bestavros, Renato Mancuso, Richard West, Erfan Sharafzadeh。秋田:虚拟化云的 cpu 调度器，2020 年。

[6] Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, Timothy Zhu。Peafowl:应用程序内 cpu 调度，以降低内存中键值存储的功耗。在第 11 届 ACM 云计算研讨会论文集中，SoCC’20，第 150-164 页，纽约，NY，USA，2020 年。计算机协会(Association for Computing Machinery)。

[7] SCHED(7) Linux 程序员手册，2020 年 9 月。

[8]贾伟伟，单建臣，Tsz On 李，尚晓伟，崔和明，丁晓宁。vsmt-io:提高虚拟化云中 SMT 处理器的 i/o 性能和效率。2020 年 USENIX 年度技术会议(USENIX ATC 20)，第 449-463 页。USENIX 协会，2020 年 7 月。

[9] Tim Harris、Martin Maas、Virendra J. Marathe。Callisto:协同调度并行运行时系统。《第九届欧洲计算机系统会议论文集》，EuroSys ' 14，纽约，NY，USA，2014 年。计算机协会。

[10] George Prekas, Marios Kogias, Edouard Bugnion。Zygos:实现微秒级网络任务的低尾延迟。《第 26 届操作系统原理研讨会论文集》，SOSP ' 17，325-341 页，纽约，NY，USA，2017。计算机协会(Association for Computing Machinery)。

[11] Kostis Kaffes, Dragos Sbirlea, Yiyan Lin，大卫 Lo 和 Christos Kozyrakis。利用应用程序类来节省高利用率数据中心的电力。在第 11 届 ACM 云计算研讨会论文集中，SoCC’20,134-149 页，纽约，NY，USA，2020 年。计算机协会(Association for Computing Machinery)。

[12] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, Edouard Bugnion。ix 操作系统:在受保护的数据平面中结合了低延迟、高吞吐量和效率。ACM 翻译。第一版。系统。，34(4)， 2016 年 12 月。

秦亨利，钱李，Jacqueline Speiser, Peter Kraft 和 John Ouster- hout。Arachne:核心感知线程管理。第 13 届 USENIX 操作系统设计和实现研讨会(OSDI 18)，第 145-160 页，卡尔斯巴德，CA，2018 年 10 月。USENIX 协会。

[14]林铉泽、韩东洙、大卫安德森、迈克尔卡敏斯基。MICA:一种快速内存键值存储的整体方法。第 11 届 USENIX 网络系统设计和实现研讨会研讨会(NSDI 14)，第 429-444 页，WA 西雅图，2014 年 4 月。USENIX 协会。

[15] Diego Didona 和 Willy Zwaenepoel。用于改善内存中键值存储中的尾部延迟的大小感知分片。第 16 届 USENIX 网络系统设计和实现研讨会(NSDI 19)，第 79-94 页，波士顿，MA，2019 年 2 月。USENIX 协会。

[16] Kostis Kaffes, Neeraja J. Yadwadkar, Christos Kozyrakis。无服务器功能的集中式核心粒度调度。ACM 云计算研讨会论文集，SoCC’19，第 158-164 页，纽约，NY，USA，2019 年。计算机协会。

[17]朱航、Kostis Kaffes、陈子旭、刘振明、Christos Kozyrakis、Ion Stoica、金鑫。Racksched:用于机架级计算机的微秒级调度器。第 14 届 USENIX 操作系统设计和实现专题讨论会(OSDI 20)，第 1225-1240 页。USENIX 协会，2020 年 11 月。

[18] Marios Kogias、George Prekas、Adrien Ghosn、Jonas Fietz 和 Edouard Bugnion。R2p2:让 rpcs 成为一流的数据中心公民。2019 年 USENIX 年度技术会议(USENIX ATC 19)，第 863-880 页，WA 伦顿，2019 年 7 月。USENIX 协会。

[19]杰克·泰格·汉弗莱斯，科斯蒂斯·卡夫斯，大卫·Mazières 和克里斯托·科兹拉基斯。注意间隙:nic 上的知情请求调度案例。《第 18 届 ACM 网络热点专题研讨会论文集》，HotNets ' 19, 60-68 页，USA 纽约 NY，2019 年。计算机协会。

[20] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Yang Zongheng, William Paul, Michael I. Jordan, Ion Stoica。Ray:新兴 AI 应用的分布式框架。第 13 届 USENIX 操作系统设计和实现研讨会(OSDI 18)，第 561-577 页，卡尔斯巴德，CA，2018 年 10 月。USENIX 协会。

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, Adam Belay。Caladan:减轻微秒级时间尺度上的干扰。第 14 届 USENIX 操作系统设计和实现研讨会专题讨论会(OSDI 20)，第 281-297 页。USENIX 协会，2020 年 11 月。

[22] Barret Rhoden, Kevin Klues, David Zhu, Eric Brewer。用新的操作系统抽象来提高数据中心的每个节点效率。第二届 ACM 云计算研讨会论文集，SOCC ' 11，纽约，NY，USA，2011。计算机协会。

[23] Manohar Vanga, Arpan Gujarati, Björn B. Brandenburg。Tableau:高密度工作负载的高通量和可预测的 vm 调度器。在 2018 年 USANY 纽约 EuroSys ' 18 第十三届 EuroSys 会议的会议记录中。计算机协会。

[24] e2 虚拟机性能驱动的动态资源管理。https://cloud.google.com/blog/products/compute/understanding-dynamic-resource-management-in-e2-vms。最后访问时间:2020-11-11。

[25] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, Christos Kozyrakis。新宿:抢先调度 for second-scale 尾巴延迟。第 16 届 USENIX 网络系统设计和实现研讨会(NSDI 19)，第 345 - 360 页，波士顿，MA，2019 年 2 月。USENIX 协会。

[26] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, Thomas F. Wenisch。Q-zilla:容尾微服务的调度框架和核心微架构。2020 年 IEEE 高性能计算机体系结构(HPCA)国际研讨会，第 207-219 页，2020。

[27] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, Yuval Yarom。伏笔 - ng:用瞬态乱序执行打破虚拟内存抽象。技术报告，2018 年。另见 USENIX 安全文件预示[28]。

[28] Jo Van Bulck、Marina Minkin、Ofir Weisse、Daniel Genkin、Baris Kasikci、Frank Piessens、Mark Silberstein、Thomas F. Wenisch、Yuval Yarom 和 Raoul Strackx。伏笔:利用瞬态乱序执行，提取 intel SGX 王国的钥匙。第 27 届 USENIX 安全研讨会(USENIX 安全 18)，第 991-1008 页，巴尔的摩，马里兰州，

2018年8月。USENIX协会。

[29] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, Daniel Gruss。Zombieload:跨权限边界的数据采样。2019年ACM计算机和通信安全SIGSAC会议论文集，CCS'19，第753-768页，USA纽约NY，2019年。计算机协会。

[30]Zombieload:跨权限边界数据泄漏。https://www。cyberus-technology.de/文章/2019-05-14-zombieload.html。最近访问:2020-09-02。

[31] S. van Schaik、A. Milburn、S. Osterlund、P. Frigo、G. Maisuradze、K. Razavi、Bos和C. Giuffrida。Ridl: Rogue空中数据加载。在

2019 IEEE安全与隐私研讨会(SP)，页88-105,2019。

[32] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, Yuval Yarom。辐射:从用户空间读取内核写入。CoRR, abs/1905.12701, 2019。

[33] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, Alexandra Fedorova。linux调度器:十年浪费的核心。第11届欧洲计算机系统会议论文集，EuroSys'16，纽约，NY，USA，2016年。计算机协会。

[34] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, Edouard Bugnion。IX:高吞吐量、低延迟的受保护数据面操作系统。第11届USENIX操作系统设计和实现研讨会(OSDI 14)，第49-65页，Broomfield, CO, 2014年10月。USENIX协会。

亚当 Belay, Andrea Bittau, Ali Mashtizadeh，大卫 Terei，大卫 Maz- ieres和Christos Kozyrakis。沙丘:安全的用户级访问特权CPU功能。第10届USENIX操作系统设计和实现研讨会(OSDI 12)，第335-348页，CA好莱坞，2012年10月。USENIX协会。

[36]郝明哲、李怀诚、Michael Hao Tong、Chrisma Pakha、Riza O. Suminto、Cesar A. Stuardo、Andrew A. Chien、Haryadi S. Gunawi。Mittos:支持毫秒尾容差，具有快速拒绝慢速感知的操作系统界面。《第26届操作系统原理研讨会论文集》，SOSP'17, 168-183页，纽约，NY，USA，2017。计算机协会(Association for Computing Machinery)。

[37]杨婷，刘彤萍，Emery D. Berger, Scott F. Kaplan, J. Eliot B. Moss。Redline:商品操作系统中对交互性的一级支持。第8届USENIX操作系统设计和实现会议论文集，OSDI'08，第73-86页，USA，2008。USENIX协会。

[38]单一舟、黄雨桐、陈一伦、张怡颖。乐高:一种用于硬件资源分解的分散式、分布式OS。第13届USENIX操作系统设计和实现研讨会(OSDI 18)，第69-87页，CA卡尔斯巴德，2018年10月。USENIX协会。

[39]。小臂。https://www.arm.com/why-arm/technologies/big-little。最后访问:2020-11-27。

[40]凯文·勒帕克。下一代amd企业服务器产品架构。IEEE热芯片年度研讨会论文集，2017年8月。

[41] Aws硝基体系。https://aws.amazon.com/ec2/nitro/。最后的访问:

2020-11-29 。

[42] dpu是什么?https://blogs.nvidia.com/blog/2020/05/20/whats-a-dpu- data-processing-unit /。最近访问:2020-11-29。

[43] Norman P. Jouppi、Cliff Young、Nishant Patil、大卫帕特森、Gaurav Agrawal、Raminder Bajwa、Sarah Bates、Suresh Bhatia、Nan Boden、Al Borchers、Rick Boyle、Pierre luc Cantin、Clifford Chao、克里斯Clark、Jeremy Coriell、Mike Daley、Matt Dau、Jeffrey Dean、Ben Gelb、Tara Vazir Ghaemmaghami、Rajendra Gottipati、William Gulland、罗伯特Hagmann、C. Richard Ho、Doug Hogberg、John Hu、罗伯特Hundt、Dan Hurt、Julian Ibarz、Aaron Jaffey、Alek Jaworski、亚历山大Kaplan、Harshit Khaitan、安迪Koch、Naveen Kumar、Steve Lacy、詹姆斯Laudon、詹姆斯劳、Diemthu Le、克里斯Leary、Adriana Maggiore、Maire Mahony、Kieran Miller、Rahul Nagarajan、Ravi Narayanaswami、Ray Ni、Kathy Nix、Thomas Norrie、Mark Omernick、Narayana Penukonda、安迪Phelps和乔纳森罗斯。张量处理单元的数据中心内性能分析。2017.

[44]幽灵内核代码。https://github.com/google/ghost-kernel。

[45]幽灵用户空间代码。https://github.com/google/ghost-userspace。

[46] Claudio Canella、Daniel Genkin、Lukas Giner、Daniel Gruss、Moritz Lipp、Marina Minkin、Daniel Moghimi、Frank Piessens、Michael Schwarz、Berk Sunar、Jo Van Bulck、Yuval Yarom。辐射:防熔毁cpu上的数据泄露。在计算机和通信安全(CCS)ACM SIGSAC会议的会议记录中。ACM，

2019.

[47]达里奥·法焦利、迈克尔·特里马基、法比奥·切科尼、马尔科·贝尔托尼亚、安东尼奥·曼奇纳。linux中最早期限优先算法的实现。2009 ACM应用计算研讨会论文集，SAC'09, 1984-1989页，纽约，NY，USA，2009。计算机协会。

[48] Folly: Facebook开源库。https://github.com/facebook/愚蠢。最后访问:2020-11-10。

[49]下降。https://abseil.io/。最后访问:2020-11-29。

[50]保罗·e·麦肯尼。利用延迟破坏:操作系统内核中的读-拷贝-更新技术分析。博士论文，俄勒冈健康与科学大学OGI科学与工程学院，2004年。可得:http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf[2004年10月15日查看]。

[51]迪克网站。数据中心计算机:cpu设计的现代挑战。https://www.youtube.com/watch?v=QBu2Ae8-8LM。最后的访问:

2020-11-10 。

P. Gadepalli, R. Pan, G. Parmer。Slite: Os对接近零成本、可配置调度的支持*。2020年IEEE实时和嵌入式技术和应用研讨会(RTAS)，第160-173页，CA，USA Los Alamitos, 2020年4月。IEEE计算机学会。

[53] Martin Karsten和Saman Barghi。用户级线程:鱼与熊掌兼得。ACM Meas。分析的第一版。系统。，4(1)，2020年5月。

[54] go编程语言。https://golang.org/。最后访问:2020-11-10。

[55] 增强光纤。html/纤维/overview.html https://www.boost.org/doc/libs/1_68_0/libs/fiber/doc/。最后访问时间:2020-11-10。

Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula和Eric Brewer。Capriccio:用于互联网服务的可伸缩线程。第十九届ACM操作系统原则研讨会论文集，SOSP'03，页268-281，纽约，纽约，USA，2003。计算机协会(Association for Computing Machinery)。

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall和Yuli周。一个高效的多线程运行系统。在关于并行程序设计的原则和实践的第五届ACM SIG-计划专题讨论会的会议记录中，PPOPP'95, 207-216页，纽约，NY，USA，1995。计算机协会(Association for Computing Machinery)。

[58] Kevin Alan Klues。并行应用中高效管理核心的操作系统和运行时支持。博士论文，加州大学伯克利分校，USA，2015。

[59]柔软。http://lithe.eecs.berkeley.edu/。最后访问:2020-11-10。

[60]海星。https://github.com/scylladb/seastar。最后访问:2020-11-10。

[61]详细介绍ebpf。https://lwn.net/Articles/740157/。最后访问:2020-12-10。

[62]吴宇建，王洪毅，钟玉红，Asaf Cidon, Ryan Stutsman, Amy Tai，杨俊峰。用于存储的Bpf受exkernel启发的一个

的方法。在第十八届 ACM 操作系统热点专题研讨会的会议记录中，HotOS 的 21。计算机协会，2021年。

[63]刘静，刘国强，刘国强。文件系统作为进程。2019 年 7 月，第 11 届 USENIX 存储和文件系统热点专题研讨会(HotStorage 19)，西澳大利亚州伦顿。USENIX协会。

[64] esxi numa 调度如何工作。https://docs.vmware.com/en/ vmware vsphere / 7.0 / com.vmware.vsphere.resmgmt 。 doc / GUID - bd4a462d 5 - 疾 病 预 防 控 制 中 心 - 4483 - 968 b - 1 - dcf103c4208.html。最后的访问:

2020-11-02 。

[65]Bpf环形缓冲器。https://nakryiko.com/posts/bpf-ringbuf。最后的访问:

2021-04-26 。

[66] io_uring的快速增长。https://lwn.net/Articles/810414/。最后访问:2021-04-26。

[67] Nir Shavit, Dan Touitou。软件事务内存。关于分布式计算原则的第十四届 ACM年会论文集，PODC ' 95, 204-213页，纽约，NY，USA，1995。计算机协会(Association for Computing Machinery)。

[68]李志强，李志强。数据库系统中的并发控制和恢复。addison - wesley，1987年。

[69] libbpf。https://github.com/libbpf/libbpf。最后访问:2020-08-25。

[70] Rocksdb。https://rocksdb.org。最后访问:2020-11-27。

[71]陈立群，李志强，陈志强，等。李志强。赫拉克勒斯:大规模提高资源效率。载于 Deborah T. Marr和李志强 H. Albonesi，编辑，第四十二届年度国际计算机架构研讨会论刊，波特兰，OR，USA，2015 年 6 月 13-17日，页 450-462。ACM，2015。

[72]数据平面开发工具包。http://www.dpdk.org/。最后访问:2019-06-26。

[73]路易斯·安德烈·巴罗佐，杰弗里·迪恩，乌尔斯 Hölzle。网络搜索一颗行星:谷歌星团架构。IEEE学报，2003,22(4):562 - 562。

[74] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy。调度器激活:对并行的用户级管理的有效内核支持。第十三届 ACM操作系统原则研讨会论文集，SOSP' 91，页 95-109，纽约，纽约，USA，1991。计算机协会。

[75] Upthread。http://akaros.cs.berkeley.edu/parlib/upthread/。最后审议:2020-11-10。

[76] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, Thomas Anderson。Floem:一种用于 nic 加速网络应用的编程系统。第 13 届 USENIX 操作系统设计与实现研讨会(OSDI 18)，第 663-679 页，卡尔斯巴德，CA，2018 年 10 月。USENIX 协会。

[77]刘明，崔天一，Henry Schuh, Arvind Krishnamurthy, Simon Peter, Karan Gupta。使用 ipipe卸载分布式应用到 smartnics上。ACM 数据通信特别兴趣小组会议论文集，SIGCOMM’19，页 318-333，纽约，NY，USA，2019年。计算机协会。

[78]李志强，李志强，李志强，等。星云 rpc 优化架构。2020 年 ACM/IEEE 第 47 届年度国际计算机架构国际年会(ISCA)，第 199-212 页，2020 年。

[79] Compute Express Link https://docs.wixstatic.com/ugd/0c1418_d9878707bbb7427786b70c3c91d5fbd1.pdf。最后访问:2019-06-26。

[80] D。王志刚，王志刚。Exokernel:用于应用级资源管理的操作系统架构。在关于操作系统原理的第十五届 ACM专题讨论会的会议记录，SOSP ' 95，页 251-266，纽约，NY，USA，1995。计算机协会(Association for Computing Machinery)

[81]存储性能开发工具包。https://spdk.io/。最后访问:2020-08-20。

[82] Ana Klimovic, Heiner Litz, Christos Kozyrakis。反射:远程闪光≈局部闪光。《第 22 届国际编程语言和操作系统体系结构支持会议论文集》，ASPLOS ' 17, 345-359 页，纽约，NY，USA，2017。计算机协会(Association for Computing Machinery)。

[83] libfuse。https://github.com/libfuse/libfuse。最后访问:2020-08-15。

[84]用户空间 i/o howto。https://www.kernel.org/doc/html/v4.14/ driver-api/uio-howto.html。最后访问:2021-01-11。

[85] Jack Tigar Humphries, Kostis Kaffes, David Mazières, Christos Kozyrakis。A Case against (Most) Context Switches，第 17-25页。计算机协会，纽约，NY，USA，2021年。

[86]杨淑珍。面向基于微内核的系统的有效用户控制调度。ACM SIGOPS Oper。系统。Rev. 41(4): 59-68, 2007。

[87] Bryan Ford 和 Sai Susarla。CPU 继承调度。Karin Petersen 和 Willy Zwaenepoel，编辑，《第二届 USENIX操作系统设计和实现研讨会论文集》，USA华盛顿西雅图，1996年 10月 28日至 31日，第 91-105 页。ACM，1996年。