基于内核旁路网络和大小敏感分片技术改善尾延迟

胡悦晨[1)]

[1)](华中科技大学 机械科学与工程学院, 武汉市 430074)

摘 要 数据中心应用程序需要微秒级的尾延迟和来自操作系统的高请求率，而且大多数应用程序处理的负载在多个时间尺度上有很高的变化。以 CPU 高效的方式实现这些目标是一个有待解决的问题。由于当今内核的高开销，实现微秒级延迟的最佳解决方案是内核旁路网络，它将 CPU 内核专门用于对网卡进行旋转轮询的应用程序。但是这种方法会浪费 CPU。Shenango 实现在类似的延迟下，更高的 CPU 效率。Perséphone 作为一个新的内核旁路操作系统调度器，它实现了应用程序感知，非工作保存的 DARC 策略，为较短的请求保持了良好的尾延迟，并且与 Shenango 和 Shinjuku 相比，可以用相同数量的内核处理更高的负载，更好地利用了数据中心的资源。在 Minio 键值存储中实现大小敏感的分片，是一种将大小请求分配到分离的内核集的新技术，大小敏感的分片通过避免首行阻塞来改善尾延迟，确保了小请求不会因为与长请求的并置而等待。与最近的键值存储技术相比，当给定的第 99 个百分位延迟值等于平均服务时间的 10 倍时，Minio 的吞吐量最高可达平均服务时间的 7.4 倍。大小敏感的分片改善了 Minio 内存中键值存储的尾部延迟。

关键词 尾延迟 吞吐量 内核旁路网络 程序感知 大小敏感分片 键值存储

Optimizing tail-latency based on kernel bypass network and size-aware sharding

Hu Yue-Chen[1)]

[1)](Department of School of Mechanical Science & Engineering, Huazhong University of Science and Technology, Wuhan, 430074)

Abstract Data center applications require microsecond tail delays and high request rates from the operating system, and most applications handle loads that are highly variable over multiple time scales. Achieving these goals in a CPU-efficient manner is an open question. Due to the high overhead of today's kernels, the best solution for microsecond latency is a kernel-by-pass network, which uses the CPU core exclusively for applications that poll the network card for rotation. But this method wastes CPU. Shenango achieves higher CPU efficiency with similar delays. As a new kernel bypass operating system scheduler, Perséphone implements the application aware, non-work save DARC strategy, maintains good tail-latency for shorter requests, and can handle higher loads with the same number of cores than Shenango and Shinjuku, which make better use of data center resources. The implementation of size-sensitive sharding in Minio key and value stores is a new technique for allocating size requests to separate kernel sets. Size-sensitive sharding improves tail-latency by avoiding first-row blocking, ensuring that small requests do not wait due to juxtaposition with long requests. When a given 99th percentile delay value is equal to 10 times the average service time, Minio throughput is up to 7.4 times the average service time compared to recent key-value storage technologies. Size-sensitive sharding improves the tail-latency of key-value storage in Minio memory.

Key words tail-latency; throughput; kernel-bypass; network programs ; size-aware sharding; key-value stores

1.引言

在许多数据中心应用程序中，响应单个用户的请求需要数千个软件服务的响应。为了向用户提供快速响应，必须支持高请求率和微秒级的尾延迟(例如，99.9 百分位)。这对于服务时间只有几微秒的请求(例如 memcached 或 RAMCloud)来说尤为重要。同时，随着摩尔定律的变慢和网络速率的上升，CPU 效率变得至关重要。在大规模的数据中心中，即使是 CPU 效率的微小改进(执行有用工作的 CPU 周期的一部分)也可以节省数百万美元。

不幸的是，现有的系统在实现高 CPU 效率方面做得很差，因为它们还需要保持微秒级的尾延迟。Linux 通常支持微秒延迟，当利用率保持在较低时，留下足够的空闲核来快速处理传入的请求。或者，内核旁路网络堆栈，比如 ZygOS，可以通过绕过内核调度器来支持微秒延迟和更高的吞吐量。然而，这些系统仍然浪费了大量的 CPU 周期;它们依靠旋转轮询网络接口卡(NIC)来检测包到达，而不是中断，因此即使没有包需要处理，CPU 也总是在使用。此外，它们缺乏在应用程序之间快速重新分配内核的机制，因此必须为它们提供足够的内核来处理峰值负载。低尾延迟和高 CPU 效率之间的这种紧张关系由于当今数据中心工作负载的突发性到达模式而加剧，提供的负载不仅在分钟到小时的长时间尺度上变化，而且在几微秒的短时间尺度上也变化。为此，文章[1]提出的 Shenango 内核旁路网络能在满足与 Linux 和 ZygOS 类似延迟的基础上，实现更高的 CPU 效率。

最新的内核旁路调度器通过共享队列和工作窃取提高了利用率，但这些技术只适用于均匀和轻尾的工作负载。对于响应时间分布广泛的工作负载，新宿利用中断实现处理器共享; 然而，新宿的中断对一位数微秒的请求造成了不可忽略的延迟，而且频繁运行代价太高(实验发现每一个中断≈2us,每 5μs 一次的抢断对可持续负载造成很大的影响)。此外，新宿对硬件虚拟化特性的非标准使用使其难以在数据中心和公共云(如谷歌 Cloud、Microsoft Azure、AWS 等)中使用。类似地，最新的拥塞控制方案通过近似最短剩余处理时间(SRPT)来优化网络利用率并减少数据流完成时间，SRPT 是最小化平均等待时间的最优方法。但与 CPU 调度不同的是，交换包调度程序有一个物理抢占单元，在最坏的情况下是 MTU; 它们处理包含实际消息大小的数据包报头;并利用可以根据数据包所属的流大小对数据包进行优先级排序的流量类，从而做出调度决策并实施策略更容易。但 CPU 调度器无法提前知道每个请求将占用 CPU 多长时间，而且对执行时间没有上限，这使得类似 srpt 的策略，或者通常对短请求进行优先级排序的策略，难以在微秒级实现。为了优化重尾部数据中心负载下的尾延迟，文章[2]引入了一个应用程序感知的内核旁路调度器 Perséphone，它实现了一个新的调度策略，动态的、应用程序感知的预留核心(DARC)。通过实验实验中，DARC 以 5%的吞吐量成本优先处理短请求，最适合那些看重微秒响应的应用程序。

许多分布式应用程序使用内存中的键值(KV)存储作为缓存或(非持久化)数据存储库。它们的性能，包括吞吐量和延迟，通常对整体系统性能至关重要。这些应用程序中有许多表现出高度扇出的模式，也就是说，它们在并行中发出大量请求。从应用程序的角度来看，总的响应时间是由对这些请求最慢的响应决定的，因此，尾部延迟对于 KV 存储至关重要考虑到它们的重要性，键值(KV)存储的性能已经成为最近许多研究的主题，无论是在软件方面还是在硬件方面。软件优化包括:零拷贝的用户级网络栈、轮询、运行到完成的处理，以及核之间的请求分片。硬件优化主要围绕 RDMA、可编程网卡或 gpu 的使用展开。许多键值(KV)存储的工作负载包括对小物品的大量请求和对大物品的少量请求。但是，由于它们的服务时间较长，处理较大条目的请求会消耗大量可用资源。因此，处理这些大型项目会增加首行阻塞的可能性。为了解决这个问题，本文引入了大小感知分片的概念。大小感知分片的原理是，通过隔离小项目的请求使其不会经历任何行首阻塞，相应的延迟分布百分比得到了改善。

2.原理和优势

理想的系统应该在每个请求的持续时间内旋转一个核心，并达到完美的效率，因为应用程序级别的工作将与 CPU 周期一一对应。Shenango 接近于这种理想状态，尽管在现实世界中会产生上下文切换、同步等开销，但对于一个速度较慢的分配器来说，它的效率比理论上的上限有显著提高。

Shenango 通过两个关键理念来应对这些挑战。首先，Shenango 将线程和包排队延迟视为计算拥塞

的信号，并引入了一种有效的拥塞检测算法，该算法利用这些信号来决定应用程序是否会从更多的核中受益。该算法要求对每个应用程序的线程和包队列进行细粒度、高频率的可见性。因此，Shenango 的第二个关键思想是将一个繁忙的核心专用于一个被称为 IOKernel 的集中软件实体。IOKernel 进程具有根权限运行，为应用程序和网卡硬件队列提供服务。通过忙碌旋转，IOKernel 可以以微秒级检查线程和包队列，以协调核心分配。此外，它可以提供低延迟的网络访问，并允许将包转向软件中的内核，允许在内核真正分配时快速重新配置包转向规则。结果是，核心重新分配只需要 5.9 秒完成，并且需要不到 2 微秒的 IOKernel 计算时间来进行编排。这些开销支持一个足够快的核心分配率，既能适应负载的变化，又能快速纠正拥塞检测算法中的任何错误预测。

程序开发人员的高生产率。

实验证明 Perséphone 相较 Shenango 和 Shinjuku，可以用相同数量的内核处理更高的负载。Perséphone 调度器由三个部分组成，如图 2 所示。这些组件作为一个事件驱动的管道运行，并按如下方式处理数据包。1.在进入路径上，网络工作端从网卡获取数据包并将其推送给调度器；2.调度器使用一个用户定义的请求分类器对传入的请求进行分类；3.调度器将分类后的请求存储在类型化队列中（针对单一请求类型的缓冲区）；4.调度器运行 DARC 的调度程序选择一个请求，并将其推送给应用程序端 5.应用程序端处理请求，格式化一个响应缓冲区；6.应用程序端将指向该缓冲区的指针推到 NIC；7.应用程序端通知调度器它已经完成了请求。

Perséphone 的架构。一个或多个网络工作端将数据包从网卡中取出，调度程序应用请求分类器并

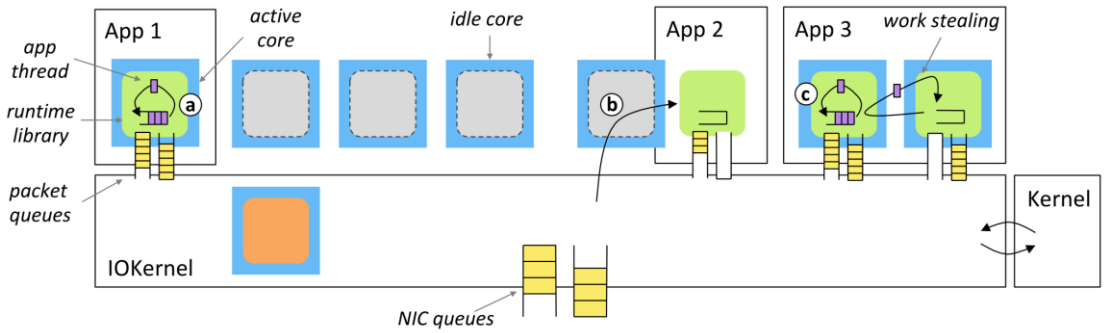


图 1　Shenango 架构。(a)用户应用程序作为单独的进程运行，并与我们的内核旁路运行时链接。(b)IOKernel 运行在一个专用的核心上，转发数据包并将核心分配给运行时。(c)运行时在每个核心上调度轻量级的应用程序线程，并使用工作窃取来平衡负载。

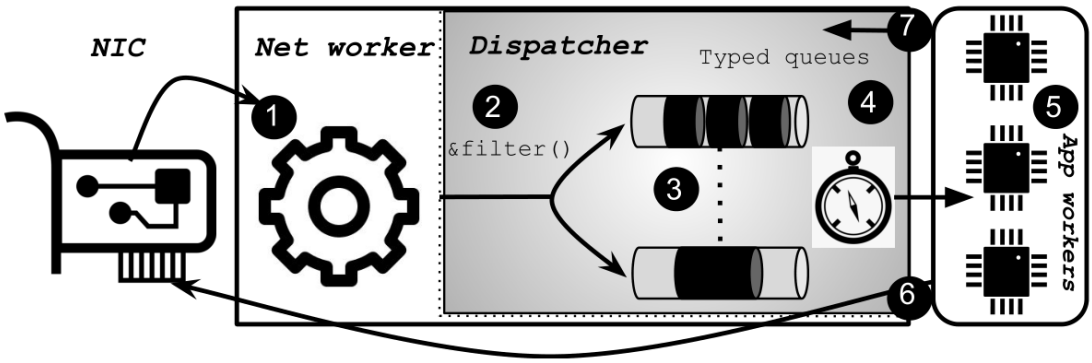


图 2　Perséphone 的架构

基于此架构，Shenango 实现了三个目标：(1)微秒级的端到端尾延迟和数据中心应用的高吞吐量;(2)多核机器上应用程序的 cpu 高效打包;(3)应用

执行 DARC 调度，应用程序端执行应用程序处理(例如，从键值(KV)存储中获取值)

Perséphone 整合的 DARC 向应用程序工作者提

供了一个单一的队列:队列按平均服务时间排序，其中的请求以先到先服务的方式将它们从队列中取出。给定类型的请求不仅可以在其预留的核上调度，还可以从分配给更长类型的内核中窃取周期。在实践中，因为可以有选择地通过对较短请求的工作窃取来启用工作保存，所以 CPU 浪费更小，DARC 能够更好地容忍短请求的突发。同时调度程序使用排队延迟和 CPU 需求的变化作为性能信号来限制长请求的处理。如果前者超出了目标减速 SLO，而后者明显偏离当前需求，则调度程序继续更新预订并过渡到下一个窗口。

通过使用 DPDK 对 Perséphone 进行原型化，并将其与 Shenango 和 Shinjuku 这两个最先进的内核旁路调度程序进行比较，文章[2]表明 Perséphone 和 DARC 可以极大地提高请求延迟，在目标 SLO 上分别比 Shenango 和 Shinjuku 多 2.3 倍和 1.3 倍的负载。此外，与 Shinjuku 的抢占技术相比，这些改进以更低的成本实现了长请求，突出了传统操作系统调度技术在微秒级的挑战。

文章[3]实现大小敏感分片的 Minio 键值(KV)存储。大小敏感的分片通过避免首行阻塞来改善尾延迟。其他大小无关的分片方法，如基于键哈希的分片、请求分发和窃取，都不能避免首行阻塞，因此会出现更糟糕的尾延迟。Minio 为小项目的所有请求使用硬件调度，这构成了所有请求的绝大部分。它通过调整处理大小项目请求的内核数量，以适应它们在工作负载中的相对存在，从而实现负载平衡。我们将 Minio 与三种最先进的内存 KV 存储设计相比较。与最接近的竞争对手相比，Minio 实现了 99 个百分点的延迟，最高降低了两个数量级。换句话说，当给定的第 99 个百分位延迟值等于平均服务时间的 10 倍时，Minio 的吞吐量最高可达平均服务时间的 7.4 倍。

3.性能评估

在测试 Shenango 性能时，文章[1]主要从以下几个方面入手：1)比较 Shenango 和其他跨不同工作负载和服务时间分布的系统的延迟和 CPU 效率；2)Shenango 对突发负载的反应；3)Shenango 的单个机制对其可观测性能的贡献。

文章[2]根据 Shenango 和 Shinjuku 提供的策略评估 DARC 调度：1)对于短长请求离散度为 100 倍的工作负载，与 Shenango 和 Shinjuku 相比，Perséphone 可以分别维持 2.35 倍和 1.3 倍的吞吐量；2)Perséphone 可以比 Shenango 多维持 1.4 倍的吞吐量，并在短请求方面比新宿提高 1.4 倍的速度；3)对于以 TPC-C 基准为模型的工作负载，Perséphone 比 Shenango 降低了 4.6 倍的减速，比新宿降低了 3.1 倍的减速；4)对于 RocksDB 应用，DARC 的吞吐量分别比 Shenango 和 Shinjuku 高 2.3 倍和 1.3 倍。此外，文章[2]还演示了 Perséphone 可以处理工作负载迅速变化和程序员提供错误请求分类器的不利情况。

文章[3]测试的重点如下:1)Minio 实现了低延迟和高吞吐量，与最接近的竞争对手相比，Minio 实现了 99 个百分点的延迟，比它低一到两个数量级，当指定的 99 个百分位数是平均服务时间的 10 倍时，其吞吐量比第二最佳方法高 7.4 倍；Minio 优于现有的设计，在给定的工作负载和给定的 SLO 上始终实现更高的吞吐量；在线程密集型和写密集型的工作负载下，Minio 都取得了良好的性能；Minio 随着可用网络带宽(6.4)的数量而扩展；Minio 实现了内核之间的负载平衡；Minio 可以适应不断变化的工作负载条件。

3.1 测试Shenango性能

3.1.1 比较Shenango与其他系统的延迟和CPU效率

在本节中，文章[1]评估 memcached、spin-server 和 gdnsd 的 CPU 效率和延迟。使用 6 个客户端服务器来生成负载，足以将客户端排队延迟降到最低。每个客户端使用 200 个持久连接(总数 1200 个)，逐渐增加负载，并在几秒钟内测量每一个提供的负载，因此突发负载只来自泊松到达过程。

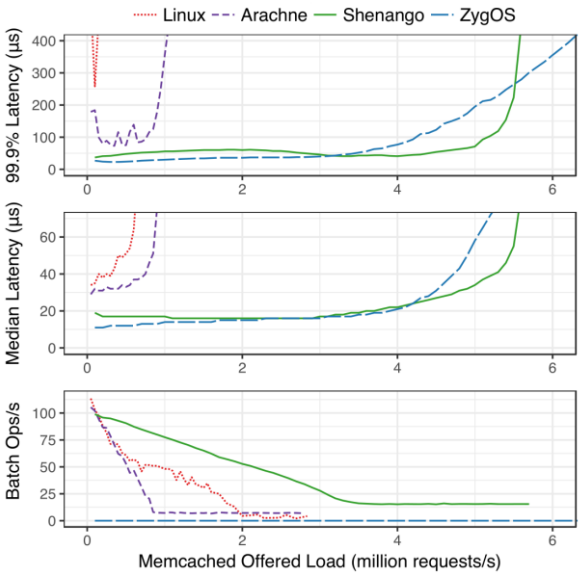


图3 Shenango始终保持低中值和99.9%的延迟，与ZygOS相当，同时允许批处理应用程序使用未使用的周期。

文章[1]使用 USR 工作负载:请求遵循泊松到达流程，由 99.8%的 GET 请求和 0.2%的 SET 请求组成。对于 Shenango，限制 memcached 最多使用 12 个超线程，因为这样 memcached 的性能最好。

如图 3 所示，Shenango 始终保持低中值和 99.9%的延迟，与 ZygOS 相当，同时允许批处理应用程序使用未使用的周期。

为了评估 Shenango 在存在批处理应用程序的情况下处理服务时间变化的能力，运行 spin-server。spin-server 有三个服务时间分布，每个分布的平均值为 10μs，这里 constant 表示其中所有请求花费相同的时间; exponential；bimodal 表示其中 90%的请求需要 5μs，10%需要 55μs。

吞吐量。

文章[1]通过 Linux 和 Shenango 同时运行 gdnsd 和交换来评估 UDP 性能;我们没有将 gdnsd 移植到 ZygOS 或 Arachne。Linuxgdnsd 可以每秒驱动 900,000 个请求，中位延迟为 41 秒，在开始丢弃包之前延迟为 99.9 个百分点。Shenango gdnsd 能够扩展到每秒 570 万个请求(提高了 6.33)，其中位延迟为 36 秒，99.9 个百分位延迟为 73 秒。由于空间限制，在此省略了一个图。

3.1.2 Shenango 对突发负载的反应

在本实验中，文章[1]以 1 s 的模拟工作生成 TCP 请求，并测量负载突然增加对尾延迟的影响。实验提供的基线负载为每秒 100,000 个请求，持续 1 秒，然后立即增加到一个更高的速率。在以新的

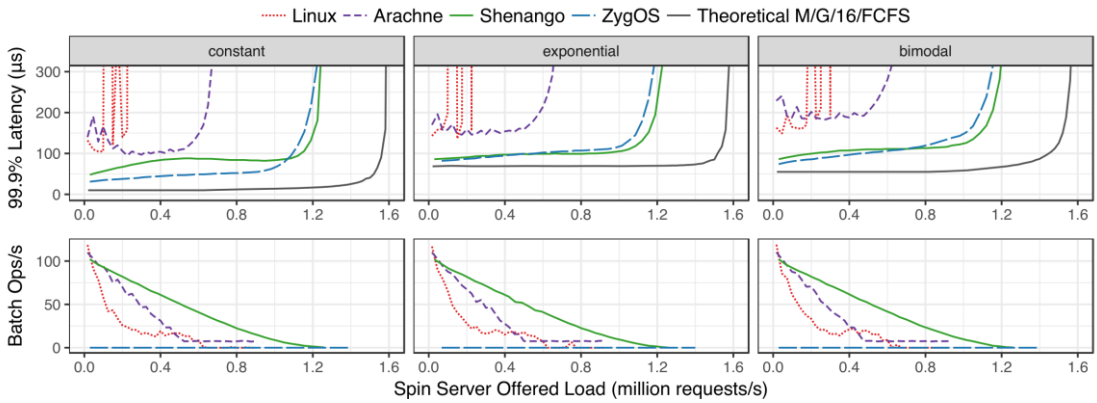


图 4　显示了当改变自旋服务器上的负载时，产生的 99.9 百分位延迟和批处理吞吐量。

如图 4 所示，由于数据包处理等开销，所有系统的吞吐量都低于 M/G/16/FCFS 模拟所能达到的理论最大吞吐量。与 zygos 相比，shenango16 个超线程中有 2 个用于运行 IOKnel，但 spin 服务器的吞吐量却略高。Shenango 的尾部延迟类似于 ZygOS，但因为 ZygOS 必须为 spin 服务器提供所有的核心才能达到峰值吞吐量，所以它不能实现任何批处理

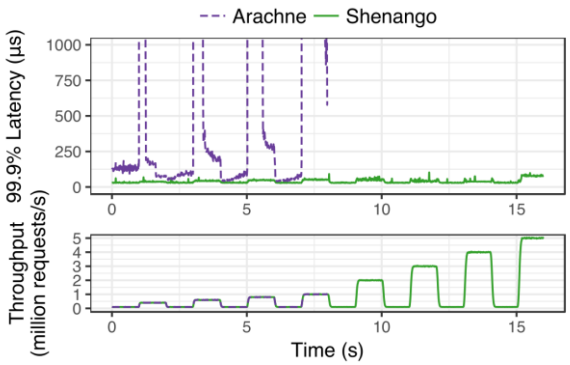


图 5　Shenango 对突发负载的反应

速率增加一秒后，负载下降到基线。未使用的核被分配到批处理中，保持 CPU 利用率在 100%。

相比之下，Arachne 最终能够满足实验中所提供的负载，即每秒 100 万个请求。然而，由于它的内核分配速度很慢，在负载转移之后，它可能需要超过 500 毫秒的时间来添加足够的内核来适应，这导致它积累了一个等待请求的积压。因此，即使是在相对温和的负载转移之后，Arachne 也会经历几毫秒的尾部延迟。相比之下，shenangoa 的响应速度非常快，几乎没有额外的延迟，即使在处理每秒 10 万到 500 万个请求的极端负载变化时也是如此。

3. Shenango 的单个机制对其可观测性能的贡献

	pthreads	Go	Arachne	Shenango
Uncontended Mutex	30	24	55	37
Yield Ping Pong	593	109	79	52
Condvar Ping Pong	1,900	281	203	100
Spawn-Join	12,996	462	595	148

图 6　执行普通线程操作的纳秒数(最快用绿色高亮显示)

Shenango 在互斥对象之外的所有对象上表现最好。在 Go 中，互斥对象要快一些，因为它的编译器可以内联它们。sHenango 在除了一个基准测试之外的所有测试中都优于这三种系统，这是因为它的预分配堆栈、无原子唤醒，以及注意避免保存可以安全地被破坏的寄存器。

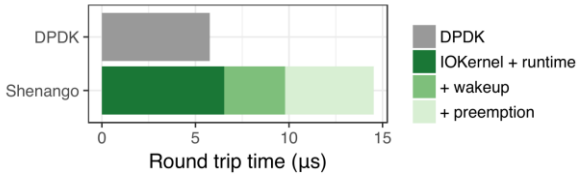


图 7 网络堆栈和核心分配开销

在 Shenango 中，遍历网络栈、唤醒 kthread 和抢占 kthread 只会给数据包的 RTT 增加一些开销，如图 7 所示。

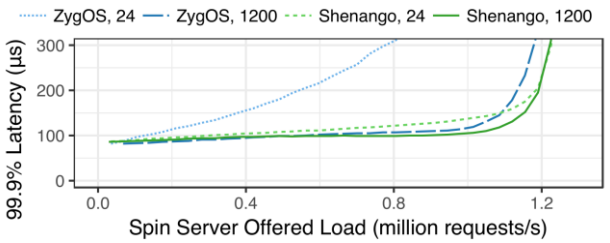


图 8 数据包负载平衡

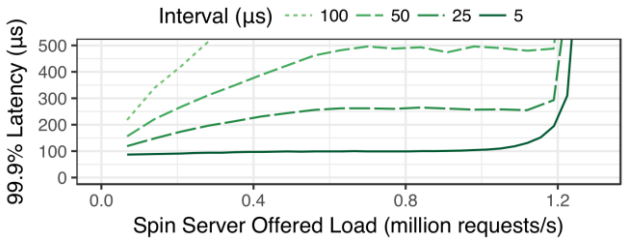


图 9 核心配置时间间隔

通过工作偷窃数据包处理，Shenango 可以比 zygo 更有效地负载平衡，并且在 24 个客户端连接下保持几乎和在 1200 个连接下一样好的性能，如图 8 所示。如图 9，Shenango 的尾部延迟随着更大的核心分配间隔而降低

3.2 DARC调度评估

在 100 倍离散度（High Bimodal)的工作负载下对于 20 倍的减速目标，DARC 可以分别比 Shenango 和新宿多维持 2.35 倍和 1.3 倍的流量，如图 10 所示。

在 1000 倍离散度（Extreme Bimodal)的工作负载下对于 50 倍的减速目标，DARC 和新宿可以承受比 Shenango 多 1.4 倍的负载。此外，与新宿相比，DARC 将短请求的减速降低了 1.4 倍。注意不同的 Y 轴表示减速和长请求尾延迟，如图 11 所示。

如图 12 所示与 Shenango 的 c-FCFS 相比，DARC 为支付、订单状态和新委托单量事务分别提供高达 9.2 倍、7 倍和 3.6 倍的吞吐量。代价是来自较长请求的投递和存储级事务遭受了更高的尾延迟。因为 DARC 从 14 个核中排除了 8 个较长的 Delivery 和 StockLevel 事务，这些事务与 Shenango 的 c-FCFS 相比遭受了更高的尾延迟。然而，有趣的是，由于 DARC 基于优先级的调度，交付事务在高负载时经历了与 c-FCFS 的竞争的尾部延迟。此外，虽然受益于 Payment 和 OrderStatus 请求，但对于中等缓慢的 NewOrder 请求，Shinjukus 提供了类似于 c-FCFS 的性能，因为它在一半的程度上抢占了它们，以保护较短的请求。同样，交付和存储级请求也会遭受重复抢占。相比之下，DARC 能够为 NewOrder 请求保持良好的尾部延迟，在高负载下为 Delivery 交付和 Stocklevel 存储级提供了更好的权衡(后者没有显示在图中)，并且与新宿相比降低了高达 3.1x 的减速。

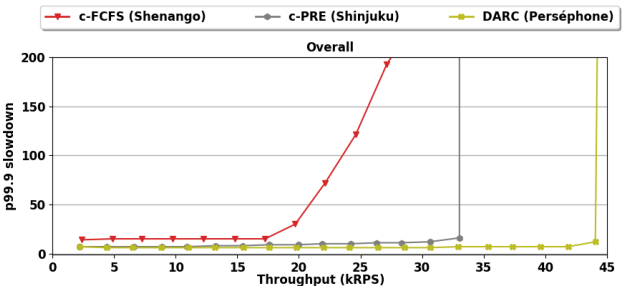


图 13 RocksDB 环境下测试

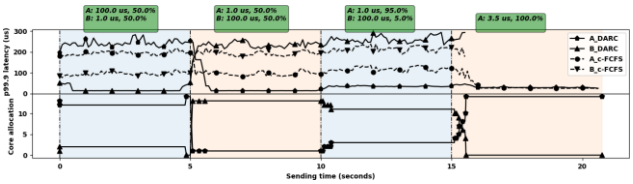


图 14 处理工作负载的变化

RocksDB (Facebook 使用的数据库引擎)下对于 20 倍的减速目标，DARC 可以比 Shenango 和新宿分别维持 2.3 倍和 1.3 倍的高吞吐量。

在 c-FCFS 和 DARC 的 4 个阶段中，p99.9 延迟

和两种请求类型 A 和 B 的保证核心。顶部框描述了各个阶段(服务时间和比率)。在转换期间，Perséphone 的分析器为每种类型挑选新的服务时间和比率，并相应地调整核心分配。核心分配行的标记表示预订更新事件。

最后，我们评估了当用户不能提供正确的请求分类器时，DARC 的行为。我们修改调度程序，将传入的请求推送到一个随机类型的队列。我们期望每个类型化的队列都持有相同份额的每种请求类型，从而收敛到 c-FCFS。我们在两个节点设置上运行 High-bimodal(一个服务器有 8 个工作线程和一个客户端，都运行在 Silver 4114 Xeon cpu 和 Mellanox Connectx-5 卡上)。如我们所期望的，DARC-random 和 c-FCFS 表现出相似的行为。

3.3 Minio测试

3.3.1 设计原理

Minio 的目标是提高第 99 个百分位数。为此，我们确定了最小的 99%的请求。我们将这些请求的处理与较大请求的处理隔离开来，这样就不会出现首行阻塞。此外，我们分配了足够数量的内核来处理这些请求，这样这些内核就不会出现太长的请求队列。

使用随机化和 hash 值来决定一个请求的目标 RX 队列，可以在 RX 队列之间实现合理的负载平衡。在 MICA 的背景下也进行了类似的观察。由于小核处理到达自己 RX 队列的请求，以及到达大核 RX 队列的同等比例的请求，所以总体上小核之间的负载是平衡的。通过对小请求使用纯硬件调度，

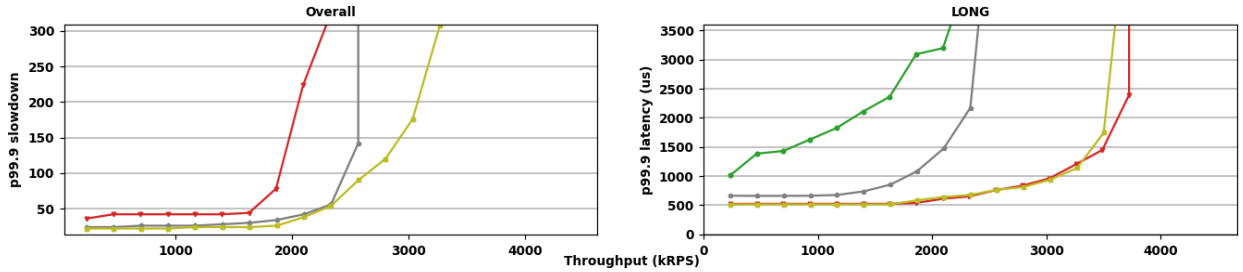


图 10　High Bimodal 工作负载下，不同调度器吞吐量比较

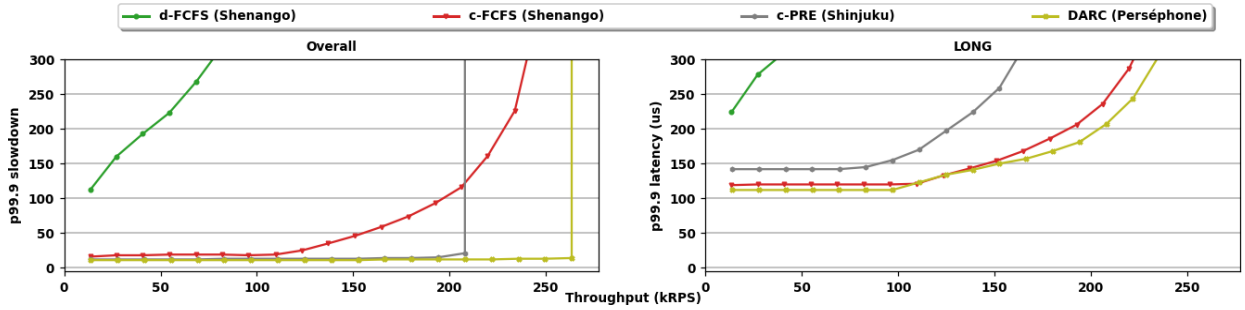


图 11　Extreme Bimodal 工作负载下，不同调度器吞吐量比较

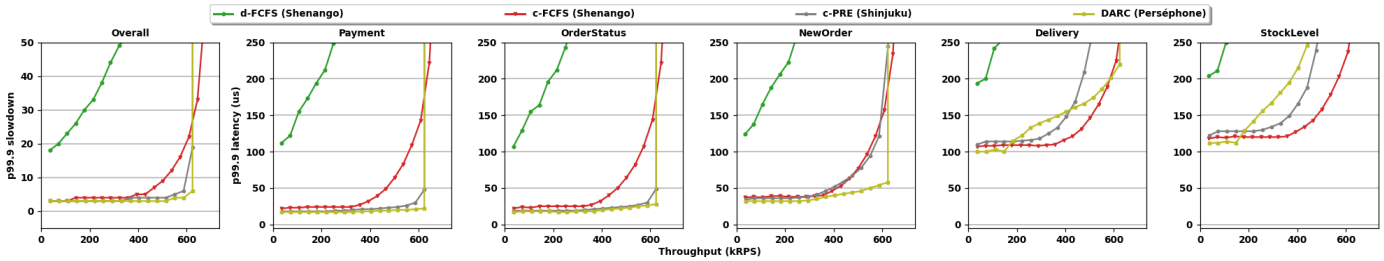


图 12 TPC 模型下，不同调度器吞吐量比较

我们消除了处理过程中任何不必要的开销，例如，软件调度。当每次抛出更大的请求时，我们都会得到这个结果，因为总有至少一个内核可以处理更大的请求。

与像 MICA 这样的纯硬件调度解决方案相比，唯一的开销是:1)软件调度非常小的大的请求数量,2)同步 RX 队列和软件队列的大型核心,我们发现竞争低,3)一些小损失在当地的小请求到达大核的 RX 队列。

3.3.2 性能评估

Minio 实现了低延迟和高吞吐量。与最接近的竞争对手相比，Minio 实现了 99 个百分点的延迟，比它低一到两个数量级。当指定的 99 个百分位数是平均服务时间的 10 倍时，其吞吐量比第二最佳方法高 7.4 倍。

图 15 显示了第 99 百分位延迟(99p)作为默认工作负载下的吞吐量函数。Minio 的最大吞吐量(6.2 Mops)和最低的延迟(≤50µs, 90%的最大吞吐量)。

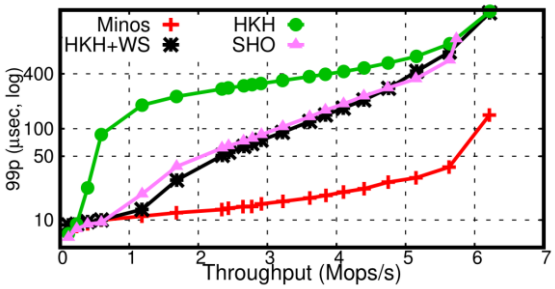


图 15 Minio 与不同键值存储技术比较

吞吐量与默认工作负载下的 99 个百分位延迟。通过有效地分离大小请求，Minio 能够提供最高的吞吐量和最低的尾延迟。Minio 的吞吐量与纯基于硬件的设计相匹配，并实现了比软件切换设计更低的尾部延迟

在处理可变大小的物品时，Minio 克服了现有

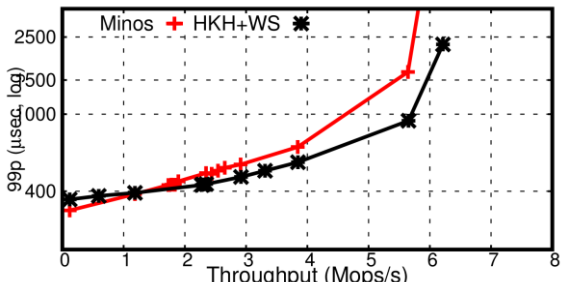


图 16 重尾延迟下 Minio 的吞吐量

设计的局限性，实现了最佳性能。即使在相对较低的负荷下，HKH 的早期绑定也会导致线首阻塞。在低或中等负载下，工作窃取减轻了 HKH 中的前端阻塞，并使 HKH +WS 延迟接近于后期绑定的延迟

图 16 报告了 Minio 和 HKH+WS(最佳替代方案)中大型请求的第 99 个百分位延迟。随着负载的增加，偷盗发生的次数越来越少，并且 HKH+WS 的性能下降到 HKH 的水平。在 SHO 中，延迟绑定在很大程度上避免了排队头阻塞，但大量请求的突然激增会伤害到高百分比的请求

吞吐量与默认工作负载下的大请求的 99 个百分位延迟。Minio 用它在整体的 99 个百分比上的巨大利益换取了对少数大型请求的适度惩罚，这些请求已经代表了一小部分工作负载。

图 17 和图 18 报告了与其他设计相比，Minio 实现的吞吐量增加(y 轴以日志尺度表示)。

图 17 对于给定的 99 个百分比的延迟 SLO，可以实现的最大吞吐量与不同百分比的大请求(y 轴以日志尺度表示)。每个条代表 Minio 在其他设计上的加速(越高越好)。

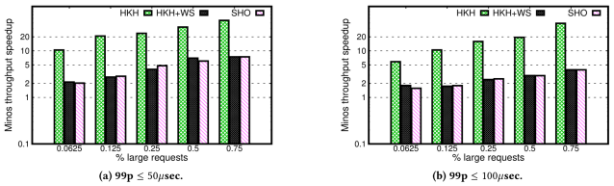


图 17 请求占用时间不同对 Minio 吞吐量加速的影响

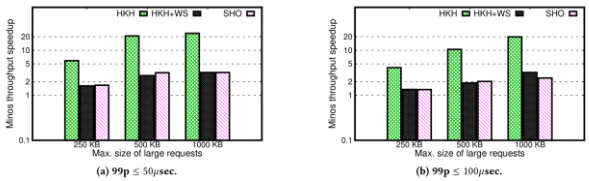


图 18 请求大小的不同对 Minio 吞吐量加速的影响

图 18:对于给定的 99 个百分点的延迟 SLO，对于不同的大请求的最大大小(以日志为单位的 y 轴)，可以实现的最大吞吐量。每个条代表米诺斯在其他设计上的加速(越高越好)。

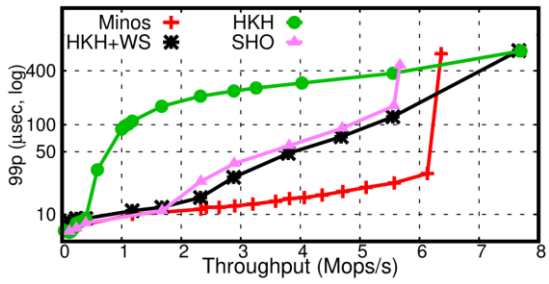


图 19 在线程密集型和写密集型的工作负载下，Minio 均取得了良好的性能。

Minio 优于现有的设计，在给定的工作负载和给定的 SLO 上始终实现更高的吞吐量。吞吐量加速随着 pL 和 sL 的增加而增加，因为大(r)请求的增加会对小请求的延迟产生负面影响，因此会影响第 99 个百分位数的延迟。正如预期的那样，SLO 越严格，吞吐量收益就越高:性能目标越松散，Minio 设计的影响就越小。对于更严格的 SLO，Minio 对 HHK+WS(对应 pL = 0)的加速达到 7.4，即第二好的设计。对于较宽松的 SLO，加速范围从 1.34(sL= 250KB)到 3.9 (pL = 0.75)。

我们现在研究写的强度对米诺斯的影响。图 19 报告了所有四个系统和一个 50:50 的 GET:PUT 工作负载的响应时间。

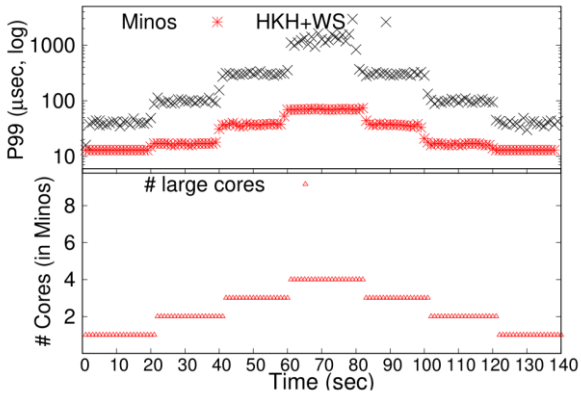
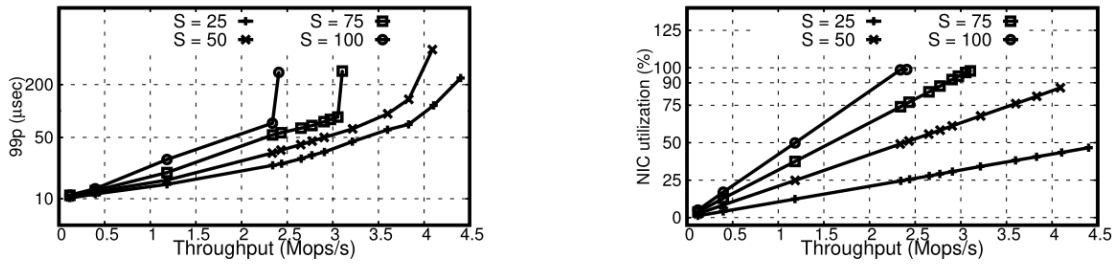


图 22 Minio 可以适应不断变化的工作负载条件

写密集型工作负载将瓶颈从网卡转移到 CPU。Minio 使 CPU 饱和的时间早于 HKH 和 HKH+WS，这是因为分析工作负载并周期性地在核心 0 上聚合它们以计算项大小的第 99 百分位所产生的开销。我们目前正在研究减少这种开销的技术，例如: 只对请求的一个子集进行抽样。或者，如果目标工作负载的跟踪可用于脱机分析(在生产工作负载中很典型)，则可以静态设置大请求和小请求之间的阈值。有了这个变体，Minio 能够匹配 HKH 和 HKH+WS 的吞吐量。

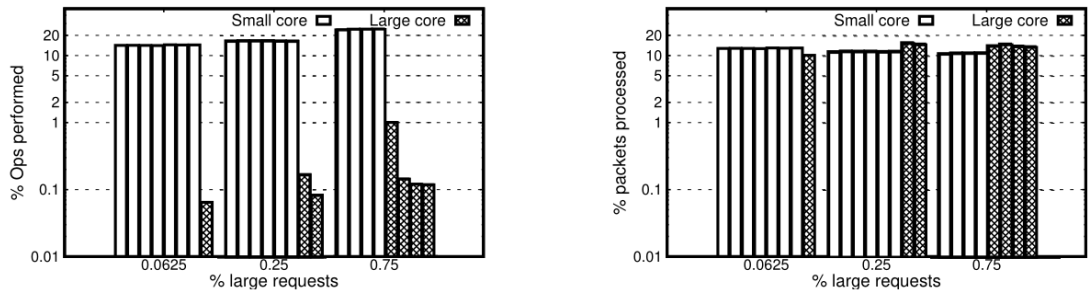
图 20 报告了实验结果。左边的图显示了吞吐



(a) Throughput vs. 99th percentile latency

(b) Throughput vs. NIC utilization.

图 20 Minio 随着可用网络带宽的数量而扩展



(a) Operations per second.

(b) Packets per second.

图 21 Minio 实现了内核之间的负载平衡

量 vs. 99 个百分点的延迟(y 轴以对数为单位)。右边的显示了 NIC 的利用率作为吞吐量的函数。随着 S 的降低，Minio 可以承受更高的负载，因为瓶颈逐渐向 CPU 转移。Minio 能够充分利用可用资源，总能达到吞吐量值，使网卡(S = 100、75、50)或 CPU (S = 25)接近饱和。

图 21(a)报告了执行的请求的百分比，图 21(b)报告了每个核处理的包的百分比(以日志为单位的 y 轴)。由此可以得出两个结论。首先，所有的核处理的数据包数量大致相同，因此执行的工作量大致相同。显然，小内核每秒处理的请求更多，因为这些请求涉及的工作更少。大型内核每秒在彼此之间处理不同的请求，这是 Minio 在大型请求中也实现了大小敏感的分片的结果。其次，Minio 将大小核的数量作为工作负载的函数来变化，这样就可以在所有核之间平衡工作。

我们最后演示了 Minio 适应不断变化的工作负载的能力。为此，我们的工作负载是每 20 秒变化一次的大型操作的百分比。它首先从 0.125 逐渐增长到 0.75，然后收缩到 0.125。我们将请求到达率固定在 2.25 Mops，对应于 pL= 0 时的高负载。75. 图 10(上)比较了 Minio 和 HKH+WS 的性能，即第二好的设计。每个点表示在 1 秒窗口内测量的第 99 个百分位延迟(y 轴以对数表示)。图 10(底部)显示了 Minio 在一段时间内为大型请求分配了多少内核。Minio 的延迟比 HKH+WS 低 2 个数量级(p L = 0 时，70 秒 vs 1 毫秒)。75)。Minio 通过编程方式按照大小请求对应的负载比例分配内核来实现这一结果。

4.总结与展望

4.1总结

本文介绍了 Shenango 系统，它可以在处理多个延迟敏感和批处理应用程序的机器上同时保持 CPU 效率、低尾延迟和高网络吞吐量。Shenango 通过它的 ioknel 实现了这些好处，这种设计允许 Shenango 通过恢复繁忙旋转中由于最小负载和峰值负载之间的供应缺口而浪费的循环，大大改进了以前的内核旁路网络堆栈。

Perséphone 作为一个新的内核旁路操作系统调度器，它实现了应用程序感知，非工作保存的 DARC 策略。在重尾工作负载中，DARC 将核心用于短请求，以保证它们不会被长请求阻塞，从而为较短的请求保持了良好的尾延迟，并且与 Shenango 和 Shinjuku 相比，可以用相同数量的内核处理更高的负载。

在 Minio 实现大小敏感的分片，是一种将大小请求分配到分离的内核集的新技术，这确保了小请求不会因为与长请求的并置而等待。与最近的键值存储技术相比，当给定的第 99 个百分位延迟值等于平均服务时间的 10 倍时，Minio 的吞吐量最高可达平均服务时间的 7.4 倍。

4.2工作展望

1）Shenango 硬件拓展。文章[1]中对 Shenango 的评估没有考虑多插座、NUMA 机器。一个选项可能是运行多个 IOKernel 实例，每个套接字一个实例。每个 IOKernel 实例都可以与其他实例交换消息，可能会在套接字之间实现粗粒度的负载平衡。这样的设计将使 ioknel 能够进一步扩展。同时，ioknel 的大部分开销是在转发数据包上，而不是在编排核心分配上。因此，其硬件负载有待进一步探索，比如新的，可以有效地向 ioknel 公开关于队列堆积信息的网卡设计。

2）Perséphone 的网络模型有待进一步优化.目前网络工作端是一个第 2 层转发器，对以太网和 IP 报头执行简单的检查；应用程序端处理第 4 层及以上的转发器，直接执行 TX。该设计旨在最大化调度程序的性能，但也 Perséphone 的主要瓶颈。

3）在微秒尺度上，实现和协调抢占式系统仍具有挑战性。虽然理论上是可取的，但微秒级的中断很难实现。在微秒级下，对于短请求，放缓目标为 10，即使是 1 微秒的开销也会导致 30%可持续负载减少。

参 考 文 献

[1] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay,and Hari Balakrishnan, MIT CSAIL. Shenango: achieving high CPU efficiency for latency-sensitive datacenter workloads.//Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation. Boston, MA, USA, Boston, MA, USA: 361-377

[2] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, B. T. Loo, L. T. Phan, Irene Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone// Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. Virtual Event, Germany, 2021: 621–637

[3] Didona, Diego and Willy Zwaenepoel. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. .//Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation. Boston, MA, USA, Boston, MA, USA: 79–93.