华中科技大学

研 究 生（数 据 中 心 技 术）课 程 论 文（报 告）

题目：网络任务尾延迟优化研究综述

学　　号　　M202173765

姓　　名　　宋诚

专　　业　　电子信息

课程指导教师　　施展、童薇

院（系、所）　计算机科学与技术学院

2021 年 12 月 20 日

网络任务尾延迟优化研究综述

宋诚

摘要 为了满足服务水平的目标(SLO)，网络规模的在线数据密集型应用程序(如搜索、电子商务和社交应用程序)依赖于现代仓库规模的数据中心的向外扩展架构。在这种部署中，单个应用程序可以由数百个软件组件组成，部署在数千个服务器上，这些服务器组织在多层中，并通过商用以太网交换机相互连接。这样的应用程序必须支持高并发连接数，并使用面向用户的 SLO 进行操作，通常以尾延迟来定义以满足业务目标。为了满足这些目标，大多数这样的应用程序将所有关键数据(例如社交关系图)分布在数百个数据服务的内存中。由于每个用户请求通常涉及数百个数据服务，必须等待延迟完成，因此数据服务的 SLO 必须考虑请求的延迟分布的长尾。单个任务的大小通常只需要少量的用户级执行。因此，理想情况下，这些服务将以最高吞吐量执行，有效地使用所有系统资源(CPU、网卡和内存)，并交付一个仅为典型任务服务时间的很小倍数的尾延迟 SLO。寻找致命的微秒需要研究人员重新审视跨网络和计算堆栈的假设，这些假设的策略和实现在加剧问题中扮演了重要角色。我们的工作集中在这些非常细粒度的内存服务的多核系统上的高效调度。

关键词 尾延迟 数据仓库 数据库 微秒级计算 计算机网络

Review on Research on Optimization of Network Task Tail Delay

Cheng Song

Abstract In order to meet service level objectives (SLO), network-scale online data-intensive applications (such as search, e-commerce, and social applications) rely on the scale-out architecture of modern warehouse-scale data centers. In this deployment, a single application can be composed of hundreds of software components and deployed on thousands of servers, which are organized in multiple layers and connected to each other through commercial Ethernet switches. Such applications must support a high number of concurrent connections, and use user-oriented SLO operations, usually defined in terms of tail delay to meet business goals. To meet these goals, most of these applications distribute all critical data (such as social graphs) in the memory of hundreds of data services. Since each user request usually involves hundreds of data services and must wait for the delay to complete, the SLO of the data service must consider the long tail of the delay distribution of the request. The size of a single task usually requires only a small amount of user-level execution. Therefore, ideally, these services will execute at the highest throughput, effectively use all system resources (CPU, network card, and memory), and deliver a tail-delay SLO that is only a small multiple of the typical task service time. The search for deadly microseconds requires researchers to re-examine assumptions across networks and computing stacks. The strategies and implementations of these assumptions have played an important role in exacerbating the problem. Our work focuses on the efficient scheduling of these very fine-grained memory services on multi-core systems.

Key words: tail latency, data warehouse, database, microsecond-level computing, computer network

1 引言

为了满足服务水平的目标(SLO)，网络规模的在线数据密集型应用程序(如搜索、电子商务和社交应用程序)依赖于现代仓库规模的数据中心的向外扩展架构。在这种部署中，单个应用程序可以由数百个软件组件组成，部署在数千个服务器上，这些服务器组织在多层中，并通过商用以太网交换机相互连接。这样的应用程序必须支持高并发连接数，并使用面向用户的 SLO 进行操作，通常以尾延迟来定义以满足业务目标。为了满足这些目标，大多数这样的应用程序将所有关键数据(例如社交关系图)分布在数百个数据服务的内存中，例如内存驻留的事务型数据库、NoSQL 数据库、键值存储，或专门的图存储。这些内存中的数据服务通常服务于来自数百个应用服务器的请求。由于每个用户请求通常涉及数百个数据服务，必须等待延迟完成，因此数据服务的 SLO 必须考虑请求的延迟分布的长尾。单个任务的大小通常只需要少量的用户级执行。因此，理想情况下，这些服务将以最高吞吐量执行，有效地使用所有系统资源(CPU、网卡和内存)，并交付一个仅为典型任务服务时间的很小倍数的尾延迟 SLO。寻找致命的微秒需要研究人员重新审视跨网络和计算堆栈的假设，这些假设的策略和实现在加剧问题中扮演了重要角色。我们的工作集中在这些非常细粒度的内存服务的多核系统上的高效调度。理论上的答案很容易理解:(a)单队列、多处理器模型比并行单队列、单处理器模型提供更低的尾延迟，(b) FCFS 为低分散任务提供最佳的尾延迟，而处理器共享在高分散服务时间分布中提供更好的结果。

计算机体系结构正处于一个拐点。仓库规模的计算机的出现，以 Web 搜索、社交网络、软件即服务等形式，将大型在线服务带到了最前沿。这些应用程序每天为数百万个用户查询提供服务，分布在数千台机器上运行，除了高吞吐量之外，还关心用户请求的尾部延迟(例如第 99 个百分位)这些特征与以前的系统有很大的不同，以前的系统关注的性能指标仅仅是吞吐量，或者最多是平均延迟。对尾部延迟的优化已经改变了我们构建操作系统、集群管理器和数据服务的方式。本文研究了尾延迟如何影响硬件设计，包括构建哪种类型的处理器内核，在缓存结构上投资多少芯片区域，服务之间有多少资源干扰，如何在多核芯片上调度不同的用户请求，以及这些决定如何与最小化芯片或数据中心层面的能源消耗的愿望相互作用。

流行的云应用程序，如网络搜索、社交网络和机器翻译，向运行在数千台机器上的数百个通信服务发送请求。然后端到端响应时间由响应最慢的机器支配。在几十毫秒内响应用户操作，要求每个参与的服务进程请求的尾部延迟在 10 到几百微秒之间。不幸的是，现代操作系统(如 Linux)中的线程管理并不是为微秒级的任务而设计的，并且经常会产生长时间的、不可预测的调度延迟，导致毫秒级的尾延迟。为了弥补这一点，研究人员开发了网络栈、数据平面和完全的应用程序，它们绕过了操作系统。这些系统中的大多数以类似的方式运行:NIC 使用接收端扩展(RSS)，以流一致的方式将传入的请求分发到多个队列;一个轮询线程以先到先服务的方式(FCFS)处理每个队列中的请求，而不会中断调度;诸如零复制、运行到完成、自适应批处理以及缓存友好和线程私有数据结构等优化减少了开销。产生的请求调度被称为分布式排队和 FCFS 调度，或 d-FCFS。当请求服务时间显示出低离散度时，d-FCFS 是有效的，就像 get/put 请求到简单的内存键值存储(KVS)(如 Memcached)的情况一样。d-FCFS 在高离散度或重尾请求分布(例如，双峰、对数正态、Zipf 或 Pareto 分布)下表现很差，因为短请求被卡在分配到相同队列的较老的长请求后面。d-FCFS 也不是工作节约，基于 RSS 的流一致哈希的实现加剧了这种影响，它只有在大量的客户端连接均匀分布在队列上时，才近似于真正的 d-FCFS。ZygOS 改进了 d-FCFS，实现了低开销的任务窃取:完成短请求的线程从被长请求绑定的

线程窃取工作。它近似于集中式的 FCFS 调度(c-FCFS)，即所有线程都为一个队列服务。偷工作不是免费的。它需要扫描缓存在非本地内核上的队列，并将系统调用转发回请求的主内核。但是，如果服务时间分散程度较低，并且有足够的客户端连接让 RSS 将请求均匀地分布在队列中，那么窃取就不会频繁发生。

大型分布式系统显示出不可预测的高百分比(尾部)延迟变化，这损害了性能的可预测性，并可能会赶走用户。许多因素可能会导致这些变化，如组件故障、复制开销、负载不平衡和资源争用。当系统运行在收获的资源上时，这个问题就会加剧。资源收集数据中心将对延迟敏感的服务(如搜索引擎)与批处理作业(如数据分析、机器学习)放在一起，以提高资源利用率。在这些数据中心中，当服务需要批处理作业时，性能隔离机制限制甚至拒绝将资源提供给批处理作业。

本文将介绍最近三年来在尾延迟优化方面的一些工作，包括改进的方案和取得的性能表现。

2 ZygOS 的调度器优化

2.1 调度器优化

目标是为尾延迟服务级别目标提供高吞吐量(每秒数百万个远程过程调用)，该目标是任务大小的一小倍。本文介绍了一种在多核服务器上为μs 级内存计算进行优化的 ZygOS 系统。它在专门为高请求率和大量网络连接而设计的操作系统中实现了一个工作节约的调度器。ZygOS 结合使用共享内存数据结构、多队列网卡和处理器间中断来重新平衡内核间的工作。对于以 99 小时百分位表示的主动服务水平目标，ZygOS 在 10 个μs 任务上达到了理论零开销模型(带 FCFS 的集中式排队)所确定的最大可能负载的 75%，在 25 个μs 任务上达到了 88%。我们使用网络版本的 Silo 来评估 ZygOS，Silo 是最先进的内存事务数据库，运行 TPC-C。对于 99μs 百分位的 1000μs 延迟的服务水平目标，ZygOS 可以在 Linux 上提供 1.63×加速(因为它的数据平面架构)，在最先进的数据平面 IX 上提供 1.26×加速(因为它的工作保存调度程序)。

2.2 内核旁路

数据平面方法，如 IX ， Arrakis 和用户级堆栈绕过内核，依靠 I/O 轮询来增加吞吐量和减少延迟抖动。例如，IX 将 memcached 的吞吐量提高了 6.4 倍，超过 Linux。

虽然这些全面的简化提供了实质性的吞吐量改进，但在资源效率方面却付出了关键的代价:数据平面的无同步特性迫使每个线程只处理 NIC 硬件定向到它的包。假设一个均衡的、高连接计数的扇入模式，这样的设计不会对吞吐量产生实质性的影响，甚至不会意味着延迟，因为所有的核平均会得到相同的工作量。然而，当负载低于饱和时，它会对尾部延迟产生巨大的影响，因为一些内核可能是空闲的，而另一些内核可能有积压。依赖历史信息来平衡 NIC 未来流量的数据飞机只能解决持续的不平衡和资源分配问题。对于显式设计用于统计地将负载分配到所有核上的应用程序(如其 CREW 和 CRCW 执行模型中的 MICA)，也存在同样的限制。虽然这种设计可以防止任何持续的不平衡，但将请求映射到内核的随机选择过程并不能防止内核之间的暂时不平衡。

2.3 调度策略

2.3.1 d-FCFS

策略比较:为了量化不同调度策略之间的差异，我们开发了一个离散事件模拟器。模拟器允许我们配置诸如调度策略、主机核数、系统负载、服务和到达间隔时间分布以及各种与系统相关的开销等参数。图 1 比较了调度策略的理想版本——即。，没有窃取或抢占头顶-使用模拟器。服务时间的轻尾指数分布，平均μ= 1μsec，代表了诸如内存

中 key-value 存储的 get/set 请求等工作负载。在这样简单的工作负载下，d-FCFS 可以说是可以忍受的，但在中等和高负载时，由于请求没有在工人之间完美地分配，d-FCFS 就会受到影响。c-FCFS 在这种工作负载下是最优的，而 PS 稍差一些，因为它甚至会抢占短请求。PS 时间切片用于所有模拟是 0.1μs。

对于重尾请求分布来说，d-FCFS 是一个糟糕的选择，如在搜索引擎中发现的，或由垃圾收集或压缩等活动引起的。重尾对数正态分布下的性能，平均μ- = 1μ- sec，标准差σ = 10μ- sec。在 d-FCFS 中，任何长请求都会阻塞分配给相同队列的每个短请求。c-FCFS 的性能明显更好，因为工人可以服务任何请求;只有当大多数工人同时处理较旧的长请求时，短请求才会延迟，这在对数正态分布中并不常见。c-FCFS 在轻尾双峰分布下的性能明显更差，这种分布通常出现在对象存储和数据库中，这些对象存储和数据库混合了简单的 get/put 请求和复杂的范围或关系查询。图(c)显示了这样的分布，99.5%的请求需要 0.5μs, 0.5%需要 500μs。与重尾情况相比，双峰分布的长请求没有那么长，但频率要高得多。

2.3.2 c-FCFS

通过抢占长请求来交错执行短请求。通过分离性能的短期和长期请求与服务时间双向负载均匀地分成 1μ秒和 100μ秒。这接近一个变电站中一半的 get / put 请求请求和另一半是范围查询。两种请求类型的尾部延迟，所有请求的请求放缓的第 99 个百分比，这是请求的总延迟与其服务时间的比值。这个比率是一个有用的度量指标，用来衡量我们在多大程度上实现了减少所有请求类型的排队时间的目标:如果这个比率很小，则意味着所有类型的排队时间都很小，并且没有请求受到不同类型请求的影响。图 2a 显示，d-FCFS 和 c-FCFS 都严重惩罚 1-μs 请求。图 2b 显示，c-FCFS 对于 100μg sec 的请求略好于 PS，因为它有效地优先处理可能被 PS 抢占的较老的、较长的请求。相对而言，c-FCFS 对较短的请求施加的惩罚远远超过了对较长请求的任何好处。

2.4 抢占式调度

通过集中式调度、快速抢占和上下文切换机制，Shinjuku 可以实现抢占式调度策略。我们开发了两种策略，它们在是否可以区分请求类型的先验方面存在差异。这些策略依赖于频繁抢占，为任何工作负载提供接近最优的尾部延迟，对于低分散的工作负载接近 c-FCFS，对于所有其他情况接近 PS。

2.4.1 单队列(SQ)策略

该策略假设我们不预先区分请求类型，并且尾延迟只有一个服务水平协议(SLO)。例如，在搜索服务中，我们无法预先知道哪些请求的服务时间更长。所有传入的请求都放在一个 FCFS 队列中。当一个 worker 空闲时，调度程序将队列头的请求分配给它。如果请求被快速处理，则该策略作为集中式的 FCFS 操作。调度程序使用时间戳来识别任何运行时间超过预定义量(在我们的实验中为 5 - 15 秒)的请求，并假设队列不是空的，就抢占它。请求被放回队列中，worker 被分配到队列的当前头部。通过模拟评估的 c-PRE-SQ 策略就是这个单一队列策略。

2.4.2 多队列(MQ)策略

该策略假设网络子系统能够识别不同的请求类型。例如，它可以解析 KVS 如 Redis 和 RocksDB 的请求头，并将简单的 get/put 请求从复杂的范围查询请求中分离出来，或者对不同的请求类型使用不同的端口。Linux 已经支持用 ebbpf 窥视数据包。每个请求类型可以有不同的尾部延迟 SLO。dispatcher 为每个请求类型维护一个队列。如果只有一个队列有挂起的请求，该策略的操作方式与上面描述的单队列策略类似。如果有多个非空队列，dispatcher 必须在一个 worker 空闲或一个请求被抢占时选择一个

队列来服务。一旦队列被选中，调度程序总是在请求的头部接受请求。

队列选择算法受到 BVT 的启发，BVT 是一种针对延迟敏感任务的进程调度算法。在 BVT 中，每个过程都有一个扭曲因子，用来量化其与其他过程相比的优先级。对于 Shinjuku，我们需要类似的翘曲因素，在短期内支持目标延迟较小的请求，但也要考虑目标延迟较长的请求的老化。由于 Shinjuku 调度请求，而不是像 BVT 这样的长时间运行的进程。对于每个队列前面的请求，该算法使用时间戳来计算它在系统中花费的时间(排队时间)与该请求类型的 SLO 目标延迟的比率。该比率最高的队列将被选中。该算法最初偏爱只能容忍较短排队时间的短请求，但最终会选择可能等待了一段时间的长请求。每个队列的 SLO 是一个用户设置参数。在我们的实验中，我们使用单队列策略分别运行每个请求类型，并使用观察到的 99%的延迟来设置它。这捕获了请求类型的性能不应受到具有不同服务时间分布的请求的影响的需求。

1 Queue Selection Policy

```
1:  procedure QUEUESELECTION(QUEUES):
2:      max ← 0
3:      max_queue ← −1
4:      time ← timestamp()
5:      for queue in queues do
6:          cur_ratio ← (time−queue[0].timestamp) / queue.SLO
7:          if cur_ratio > max then
8:              max ← cur_ratio
9:              max_queue ← queue
10:     return max_queue
```

2.5 硬件限制

2.5.1 实验环境

简单的硬件属性将极大地提高分派器的可伸缩性，将是一个低开销的消息传递机制在不同的核之间。理想情况下，这种机制将提供两种变体，一种是用于调度的抢占式机制，另一种是非抢占式机制，将消息添加到每个核心队列中，并用于工作分配。

2.5.2 线程同步

在线服务被设计成在多核上运行良好。它们跨请求进行同步，但为了实现可伸缩性，同步时间很短且不频繁。无论我们是否在读写锁周围禁用或允许抢占，可伸缩的应用程序都将在 Shinjuku 执行。我们目前在任何非线程安全的代码中禁用中断，使用一个调用安全(fn) API 调用来简化应用程序移植。用于禁用中断的指令的运行时开销只有几个时钟周期，而且它们不会影响 Linux 内核回收核心的能力。内存分配代码是一种特殊情况，它经常使用线程本地存储来优化锁。我们预先加载了自己版本的 C 和 c++库，这些库在执行分配函数时禁用了中断(因此也禁用了抢占)。如果这些功能花费的时间较长，就会影响 Shinjuku 观察到的尾潜伏期。

2.5.3 跨处理器的中断

ZygOS 设计消除了 shuffle 队列本身的首行阻塞问题。在 ZygOS 的纯协作实现中，内核对彼此的数据结构进行轮询，这导致在网络处理之前和应用程序执行之后都出现首行阻塞情况，因为网络处理显式地发生在主内核中。首先，考虑这样一种情况:在硬件网卡队列中，数据包可用来进行网络处理，但 shuffle 队列为空。这是图 4 中步骤(1)所示的队列。只要该核心正在执行应用程序代码，就没有远程核心可以窃取该任务。空闲核轮询软件和硬件的远程包队列。如果等待的包存在，它会向远程内核发送一个 IPI，从而强制执行网络堆栈，从而补充 shuffle 队列。第二,远程批处理系统调用由远程队列的核心执行在国内核心。在合作模型中,这些系统调用只是执行完成后的应用程序代码,而不幸的是直接影响 RPC 延迟一些系统调用在套接字写的反应。这里，IPI 也确保了这些远程系统调用的及时执行。因此，共享 IPI 处理器在中断用户级执行时执行两个简单的任务:(1)如果 shuffle 队列为空，则处理传入的包;(2)执行所有远程系统

调用，并在线路上传输出的包。IPI 只中断用户级的执行，因为内核处理很短而且有限制。内核在中断被禁用的情况下执行，从而避免了 TCP/IP 堆栈中的饥饿或重入问题。当该连接处于“就绪”状态时，它在 shuffle 队列中只出现一次，否则永远不会出现。

2.6 小结

SLO 的选择是由应用程序需求和规模驱动的，直观的理解是，更严格的 SLO 会降低系统的交付能力。我们证明了 SLO 的选择也会影响底层操作系统和调度策略的选择。合成基准在服务时间指数(¯S =10μS)时，通过延迟与吞吐量曲线的权衡。图 11a 和 11b 实际上是同一实验的结果，但在两个不同的 y 轴上，对应两个不同的 SLO。ZygOS 的 SLO 更严格，为 100μs(10 ×¯s)，工作保存调度器控制了尾部延迟，IX 取消了批处理功能。对于这个 SLO，IX(启用批处理)始终提供最高的尾部延迟,并以最低的吞吐量违反 SLO。然而，对于一个更宽松的 SLO(100 ×¯S)，IX 的自适应批处理在违反 SLO 之前提供略高于 ZygOS 的吞吐量。

3. Shinjuku

3.1 优化思路

最近提出的微秒级应用的数据平面，如 IX 和 ZygOS，使用非抢占策略来调度请求到内核。在许多现实场景中，请求服务时间遵循具有高离散度或较大尾部的分布，它们允许短请求被长请求阻塞，从而导致较低的尾部延迟。Shinjuku 是一个单地址空间操作系统，它使用硬件支持虚拟化，以实现微秒级的抢占。这使得 Shinjuku 可以实现集中调度策略，每 5μs 抢占一次请求，并且可以很好地处理轻尾请求和重尾请求服务时间分布。我们证明，在各种工作负载场景中，Shinjuku 提供了比 IX 和 ZygOS 更显著的尾延迟和吞吐量改进。在 RocksDB 服务器同时处理点查询和范围查询的情况下，Shinjuku 实现了高达 6.6 倍的吞吐量和 88%的低尾时延。

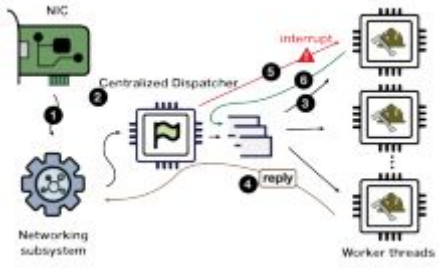

图 1：Shinjuku 系统设计

3.2 相对于 ZygOS 的改进

当请求服务时间显示出低离散度时，d-FCFS 是有效的,就像 get/put 请求到简单的内存键值存储(KVS)(如 Memcached)的情况一样。d-FCFS 在高离散度或重尾请求分布(例如，双峰、对数正态、Zipf 或 Pareto 分布)下表现很差，因为短请求被卡在分配到相同队列的较老的长请求后面。d-FCFS 也不是工作节约，基于 RSS 的流一致哈希的实现加剧了这种影响，它只有在大量的客户端连接均匀分布在队列上时，才近似于真正的 d-FCFS。ZygOS 改进了 d-FCFS，实现了低开销的任务窃取:完成短请求的线程从被长请求绑定的线程窃取工作。它近似于集中式的 FCFS 调度(c-FCFS)，即所有线程都为一个队列服务。偷工作不是免费的。它需要扫描缓存在非本地内核上的队列，并将系统调用转发回请求的主内核。但是，如果服务时间分散程度较低，并且有足够的客户端连接让 RSS 将请求均匀地分布在队列中，那么窃取就不会频繁发生。

我们将 Shinjuku 与 IX 和 ZygOS 这两种最先进的数据平面操作系统进行比较。使用合成负载，我们发现 Shinjuku 在轻尾负载方面的性能与 IX 和 ZygOS 相当，而在重尾和多模态分布方面，Shinjuku 支持高达 5 倍的负载。通过使用 RocksDB(一个流行的键值存储，也支持范围查询)，我们发现 Shinjuku 在 99 个百分点的延迟下，吞吐量

比 ZygOS 提高了 6.6 倍。我们发现,Shinjuku 可以很好地扩展可用的核数,可以饱和高速网络连接,即使连接数很少,也很有效。本文的其余部分组织如下。2 激发了对微秒级优先调度的需求。3 讨论 Shinjuku 的设计与实现。4 给出了一个全面的定量评价,5 讨论了相关的工作。

3.3 其他做法

优化的网络栈:网络栈的优化工作非常重要,包括基于轮询的处理(DPDK)、多核可伸缩性(mTCP)、模块化和专门化(Sandstorm)以及 OS 旁路(Andromeda)。Shinjuku 与此工作正交,因为它在网络协议处理后优化了请求调度。数据平面操作系统:一些最近的系统通过将操作系统数据平面与操作系统控制平面分离来优化吞吐量和尾部延迟,这一想法起源于 Exokernel。IX、Arrakis、MICA、Chronos 和 ZygOS 都属于这类。Shinjuku 改进了这些系统,引入了抢占式调度,允许短请求避免过多排队。任务调度:Li et al.通过减少用于违反 SLO 的长时间运行请求的资源数量来控制尾部延迟。哈克等人则采取相反的方法,为掉队的人投入更多的资源,这样他们就能完成得更快。有趣的是,这两种方法都很有效。但是,这些方法适用于毫秒级的工作负载,并且需要动态可并行的工作负载。Shinjuku 允许开发高效的调度策略,以应对比这一行能处理的短 3 个数量级的请求。流量调度:PIAS 是一种网络工作流调度机制,它使用交换机中可用的硬件优先级队列来近似最短作业优先(SJF)调度策略,并将短流优先于长流。我们在 Shinjuku 没有采用类似的方法,因为 SJF 在最小化平均延迟方面是最优的,但不是尾延迟。此外,为了有效,PIAS 需要某种形式的拥塞控制来保持队列长度短。这在非网络设置中是不实际的,因为运行时不能控制应用程序。无退出中断:在 ELI 中引入了安全、低开销中断的概念,以便将中断快速交付给 vm。ZygOS 使用处理器间中断来窃取工作,但不实现抢占式调度。Shinjuku 使用 Dune 来优化处理器到处理器的中断。用户空间线程管理:从调度程序激活开始,已经有一些努力来实现高效的用户空间线程库。它们都专注于协同调度。Shinjuku 表明,抢占式调度在微秒尺度上是可行的,可以实现低尾时延和高吞吐量。

3.4 快速抢占

为了在微秒延迟使用抢占式调度,Shinjuku 需要快速抢占。一个简单的方法是让调度程序使用 Linux 信号通知工作人员。然而,正如我们在表 1 中所示,信号对发送方和接收方都产生了很高的开销(在 2GHz 的机器上大约是 2.5μs)。它们需要用户到内核空间的转换,以及一些内核处理。

Preemption Mechanism	Sender Cost	Receiver Cost	Total Latency
Linux Signal	2084	2523	4950
Vanilla IPI	2081	2662	4219
IPI Sender-only Exit	2081	1212	2768
IPI Receiver-only Exit	298	2662	3433
IPI No Exits	298	1212	1993

表 1:周期内的平均抢占开销。发送方/接收方成本指的是发送/接收核心消耗的周期,包括调用空中断处理程序的接收方开销。总成本包括通过系统总线的中断传播。因此,它不等于发送方和接收方的开销之和。对于“IPI send -only Exit”和“Vanilla IPI”,接收方在发送方从虚拟机退出返回之前开始中断处理。

3.4.1 通过中断抢占

直接使用处理器间中断(IPIs)可能比信号更快。x86 处理器使用高级可编程中断控制器(APIC)实现 api。每个核心都有一个本地 APIC 和一个 I/O APIC 附加到系统总线上。为了发送一个 IPI,发送内核在其本地 APIC 中写入寄存器,该寄存器通过 I/O APIC 将中断传播到目标内核的 APIC,而后者将执行向量传递给一个中断处理器。我们通过虚拟化本地 APIC 寄存器来扩展 Dune 以支持 api。当核心 a 上的一个非根线程写入它的虚拟 APIC 发送中断号 V 到核心 B 时,这将导致一个 VM 退出到运行在根模式下的

Dune。Dune 将 V 写入核心 B 发布的中断描述符，然后使用真正的 APIC 向核心 B 发送中断 242。这会导致核心 B 向 Dune 中的一个中断处理程序执行 VM exit，在恢复应用程序时将中断号 V 注入到非根模式中。

如表 1 所示，这种使用 IPIs 的普通抢占实现略快于 Linux 信号，但由于发送方和接收方的 VM 退出成本，仍然存在巨大的开销。

Mechanism	Linux process	Dune process
swapcontext	985	2290
No signal mask	140	140
No FP-restore	36	36
No FP-save	109	109

表 2:普通 Linux 进程和 Shinjuku 使用的 Dune 进程在不同上下文切换机制下的平均时钟周期开销。

3.4.2 优化中断传递

我们首先关注在接收核心 B(Shinjuku 工人)上使用张贴中断来移除虚拟机退出，这是一个 x86 特性，用于接收中断而不退出虚拟机。为了启用发布中断，Dune 在 B 上配置其硬件定义的 VM 控制结构(VMCS)，以识别中断 242 作为特殊的发布中断通知向量。B 也向 VMCS 注册它发布的中断描述符。内核 A 在写入虚拟 APIC 时仍然会退出虚拟机。A 上的 Dune 代码将 V 写入 B 发布的中断描述符中，并将 interrupt 242 发送给 B。然而，B 直接注入中断 V 而不需要 VM 退出。表 1 显示,消除接收方 VM 出口可以减少 54%的接收方开销(从 2662 周期到 1212 周期)。这允许频繁抢占工作线程，而不会显著降低有用的工作线程吞吐量。这种接收器开销包括对硬件结构的修改，如果不进行硬件更改(例如支持轻量级用户级中断)，就无法显著改进它。

3.4.3 优化中断发送

最后，通过让 Shinjuku 调度程序直接访问真实的(非虚拟的)APIC，我们删除了发送核心(调度程序线程)上的 VM 出口。使用扩展页表(EPT)，我们将其他核的已发布的中断描述符和本地 APIC 寄存器映射到 Shinjuku 调度程序的客户物理地址空间。因此，调度程序可以直接发送 IPI，而不会导致 VM 退出。表 1 显示，消除发送方 VM 退出将发送方开销降低到 298 个周期(在 2GHz 系统中为 149ns)。

这改进了 Dispatcher 可伸缩性，允许它每秒处理更多的请求和/或更多的工作线程(核心)。表 1 给出了结合发送端和接收端优化中断交付的结果，用于支持 Shinjuku 的抢占。较低的发送端开销(298 个周期)使得构建一个集中式的、先发制人的调度程序能够每秒处理数百万个调度操作。较低的接收端开销(1212 个周期)使得它可以每 5µs 抢占一次请求，以便在不浪费超过 10%的工作人员吞吐量的情况下调度更长的请求。总代价范围从 36 到 109 周期(对于 2GHz 系统，18 到 55ns)

3.4.4 低开销上下文切换

当一个请求被调度到一个空闲的内核或被抢占时，我们会在每个 worker 的主上下文和请求处理上下文之间进行上下文切换。直接的方法是使用 Linux ucontext 库中的 swapcontext 函数。根据表 2，这个开销在普通的 Linux 进程中是非常大的，在 Dune 进程中使用这个开销会翻倍。swapcontext 需要一个系统调用来设置交换期间的信号掩码，这需要在 Dune 中退出一个 VM。在 swapcontext 中剩下的工作——即。，保存/恢复寄存器状态和堆栈指针，不需要系统调用。评估上下文切换优化。首先，我们跳过设置信号掩码，这消除了系统调用，并使 Dune 与普通 Linux 相当。这引入了一个限制，即属于同一个应用程序的所有任务都需要共享同一个信号掩码。接下来，我们利用主工作上下文不使用浮动(FP)指令。当从请求上下文切换到工作上下文时，我们必须保存 FP 寄存器，因为它们可能在请求处理中使用过，但我们不需要为工作上下文恢复它们。当从工作上下文切换到请求上下文时，我们跳过保存 FP 寄存器，

只是为请求上下文恢复它们。Shinjuku 使用表 2 中的最后两个选项来进行工人核心的上下文切换。

3.4.5 缓存优化

重新利用现有缓存来减轻后端延迟变化的影响，而不仅仅是缓存流行数据。我们的解决方案 RobinHood 将缓存资源从缓存丰富的(不影响请求尾延迟的后端)动态地重新分配给缓存不足的(影响请求尾延迟的后端)。我们在一个拥有 50 台服务器、20 个不同后端系统的集群上，通过生产跟踪来评估 RobinHood。令人惊讶的是，我们发现即使工作集比缓存大小大得多，RobinHood 也可以直接处理尾延迟。在出现负载峰值时，RobinHood 在 99.7%的情况下达到了 150ms 的 P99 目标，而下一个最佳策略在 70%的情况下达到了这个目标。

3.4.6 罗宾汉方案

针对这些挑战，我们提出了一种减少请求尾延迟的新思路，这种思路与后端服务的设计和功能无关。我们建议在多层系统中重新利用现有的缓存层，通过动态分区缓存来直接处理请求尾部延迟。将缓存空间分配给那些负责高请求尾延迟的后端(缓存不足的后端)，同时从不影响请求尾延迟的后端窃取空间(缓存丰富的后端)。在这样做的过程中，RobinHood 做出了一些看似违反直觉的妥协(例如，显著增加了某些后端的尾部延迟)，但最终改善了整个请求的尾部延迟。由于许多多层系统已经合并了一个能够动态分区的缓存层，所以可以在很少的额外开销或复杂性的情况下部署 RobinHood。

3.4.7 罗宾汉架构

RobinHood 架构由应用服务器及其缓存、后端服务和一个 RBC 服务器组成。RobinHood 需要一个可以动态调整大小的缓存系统。在我们的测试台中，我们使用现成的 memcached 实例来在每个应用服务器上形成缓存层。实现 RobinHood 需要两个额外的组件，这些组件目前还没有被生产系统(如 OneRF)使用。首先，我们向每个应用服务器添加一个轻量级缓存控制器。控制器实现了 RobinHood 算法，并向本地缓存的分区发出调整大小的命令。每个控制器的输入为 RBC。

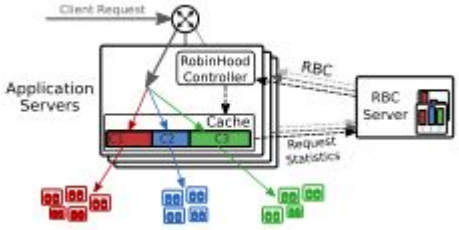

图 2:RobinHood 为每个应用服务器和延迟统计(RBC)服务器添加了一个分布式控制器

为了避免在控制器之间的全对全通信模式中，我们添加了一个集中的 RBC 服务器。RBC 服务器聚合来自所有应用服务器的请求延迟，并为每个后端计算 RBC。在我们的实现中，我们修改了应用服务器缓存库，以发送(每秒钟分批发送)每个请求的延迟和来自请求最长查询的后端 ID。在 OneRF 生产系统中，实时日志框架已经包含了计算 RBC 所需的所有指标，因此 RobinHood 不需要更改应用程序库。这些信息也已经存在于其他生产系统中，如 Facebook。控制器每次运行 RobinHood 算法时，都会轮询 RBC 服务器以获取最近的 RBC。

4. 未来发展和研究

我们介绍了 ZygOS，这是一个工作节约的操作系统，专为具有高连接扇入、高请求率和短任务执行时间的内存中延迟关键应用程序设计。ZygOS 在执行环境的框架中应用了一些经过充分验证的工作窃取思想，但避免了连接静态分区的数据平面设计的基本限制。我们在一系列综合微基准测试(具有已知的理论边界)和最先进的内存事务数据库上验证了我们的想法。ZygOS 证明，在多核系统上调度μ-s 级任务是可能的，可以

提供高的贯穿性和低的尾延迟，几乎达到饱和点。

Shinjuku 使用虚拟化的硬件支持来实现微秒级的频繁抢占。因此，它的调度策略可以避免非抢占策略的常见陷阱，即短请求被长请求阻塞。Shinjuku 提供了低尾延迟和高吞吐量的广泛分布和请求服务时间，而不管客户端连接的数量。对于 RocksDB KVS，我们发现 Shinjuku 的吞吐量比最近发布的 ZygOS 系统提高了 6.6 倍，尾部延迟提高了 88%。

我们已经看到，即使在后端同时超载的挑战性条件下，RobinHood 也能够满足 OneRF 工作负载 150ms 的 SLO。许多其他系统，例如 Facebook、谷歌、Amazon 和 Wikipedia，都使用类似的多层架构，其中一个请求依赖于多个查询。但是，与 OneRF 相比，这些其他系统可能有不同的优化目标、更复杂的工作负载或系统架构上的微小变化。在本节中，我们将讨论将 RobinHood 合并到这些其他系统时可能出现的一些挑战。非凸曲线小姐。之前的工作已经观察到了一些工作负载的非凸脱靶率曲线(也称为性能悬崖)。我们在与其他公司的讨论中也经常提到这个话题。虽然我们实验中的漏比值曲线大部分是凸的，但 RobinHood 从根本上并不依赖于凸性。具体来说，RobinHood 从来不会被困住，因为它忽略了漏失率斜率。然而，非凸性会导致 RobinHood 的分配效率低下。如果缺失率曲线非常不规则(阶跃函数)，我们建议使用 Talus 和 Cliffhanger 等现有技术来凹凸缺失率。

5. 结束语

本文以综述的角度对数据中心服务的尾延迟优化研究工作进行了分析，具体包括 ZygOS。对于传统数据中心，深入剖析了如何从以及对其的改进 Shinjuku 系统，他们从负载均衡的角度以调度器优化的方式对尾延迟做出分析，以大刀阔斧的方式撇开操作系统内核的网络栈，以负载均衡系统的专用系统的角色实现了更高的性能。RobinHood 方案则以缓存优化的方式动态感知尾延迟并提高缓存利用效率。本文最后对数据中心服务的尾延迟优化和发展进行了回顾和展望。

参考文献

[1] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17). Association for Computing Machinery, New York, NY, USA, 325–341. DOI:https://doi.org/10.1145/3132747.3132780

[2] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: preemptive scheduling for μsecond-scale tail latency. In Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19). USENIX Association, USA, 345–359.

[3] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Mor Harchol-Balter, and Siddhartha Sen. 2018. RobinHood: tail latency-aware caching—dynamically reallocating from cache-rich to cache-poor. In Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI'18). USENIX Association, USA, 195–212.