**μ秒级尾延迟的抢占式调度**

**摘要**

**现有方案不足之处**

最近为微秒级应用程序(如IX和ZygOS)提出的数据平面使用非抢占策略将请求调度到核心。对于请求服务时间遵循高分散或重尾部分布的许多现实场景，它们允许短请求被阻塞在长请求之后，这导致较差的尾部延迟。

**方案简介**

Shinjuku是一个单地址空间操作系统，它使用虚拟化硬件支持来实现微秒级的抢占。这使得Shinjuku可以实现集中式调度策略，每5µ秒抢占一次请求，并且可以很好地用于轻尾和重尾请求服务时间分布。我们证明，在广泛的工作负载场景下，Shinjuku比IX和ZygOS提供了显著的尾部延迟和吞吐量改进。对于RocksDB服务器同时处理点和范围查询的情况，Shinjuku实现了6.6倍的高吞吐量和88%的低尾延迟。

1. **Shinjuku**

新宿使用集中式排队和调度架构，并依赖于低开销和频繁抢占，以确保在各种服务时间分布下的低尾延迟。

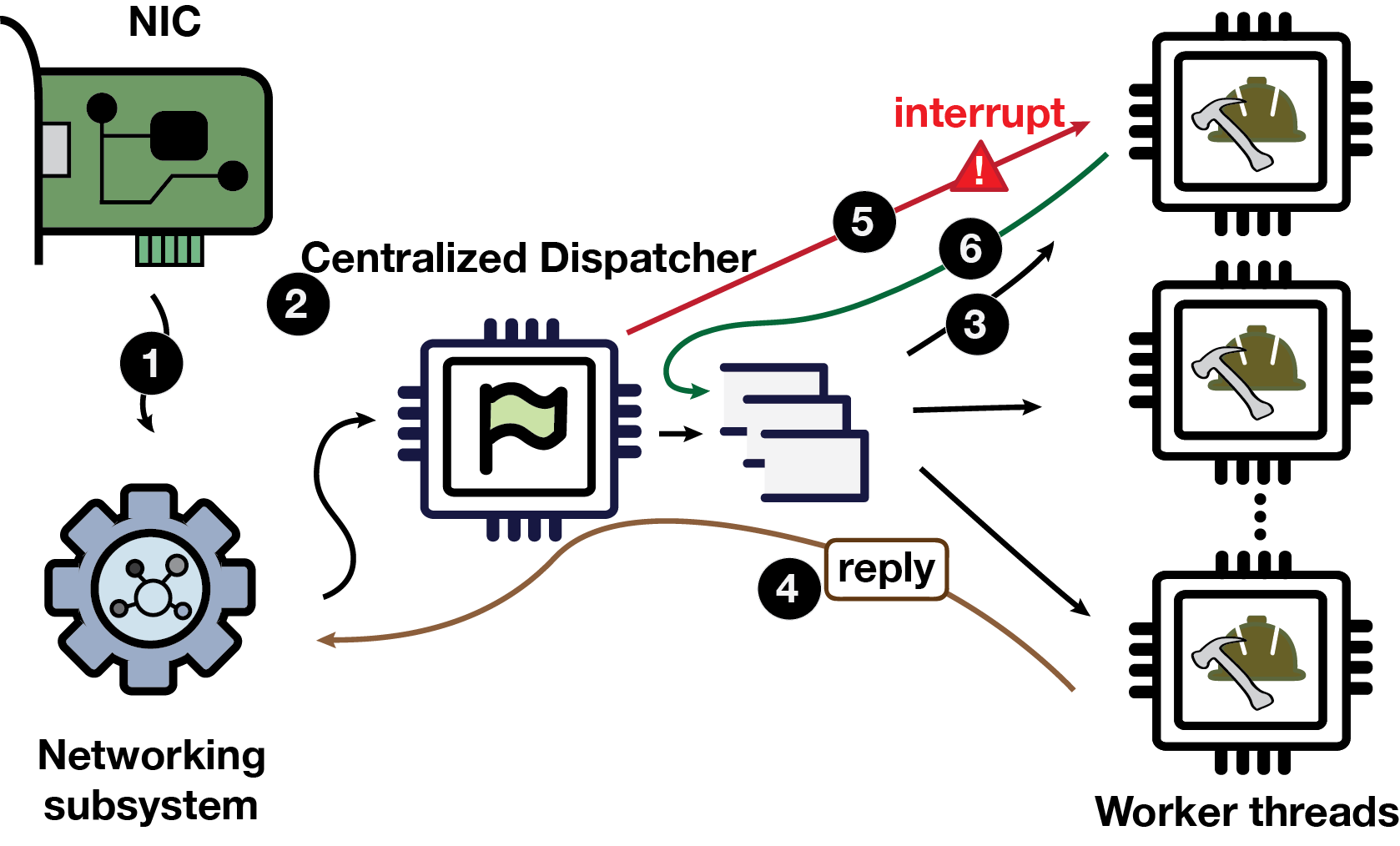


图1 Shinjuku系统设计

**3.1设计概述**

图1总结了Shinjuku中的关键组件和典型的请求流。传入的请求首先由网络子系统处理，该子系统处理所有网络协议处理并识别请求边界**①**。网络子系统可以使用一个或多个专用内核或超线程、一个smartNIC或两者的组合来实现。通过将网络处理与请求调度分离，Shinjuku可以与针对不同条件进行优化的一系列网络协议(UDP、TCP等)和各种优化的网络堆栈相结合。网络子系统将请求传递给集中式调度线程(dispatcher thread)**②**，该线程将对请求进行排队并将其调度给工作线程。调度程序为每个传入请求生成一个上下文，以支持抢占和重调度。在最简单的形式中，调度程序为所有挂起的请求维护一个队列。分派器将请求发送给工作线程**③**，每个线程使用专用的硬件核心或超线程。大多数请求将不间断地完成执行。任何应答的网络处理都可以在网络子系统或工作线程本身进行，以优化延迟。工作线程至少会通知网络子系统释放为传入请求分配的缓冲区空间**④**。

调度程序使用时间戳来识别应该根据调度策略抢占的长时间运行的请求。假设有排队请求，我们在我们研究的工作负载中抢占了5μ秒到15µ秒后运行的请求，这与Linux内核中的时间片相比是非常频繁的。例如，CFS调度器的目标抢占延迟为6ms，最小抢占延迟为0.75ms。工作线程执行上下文切换，并从调度程序接收不同的请求来运行**⑤**。长请求在调度程序中重新进入队列**⑥**，然后根据**②-⑥**步骤处理后续。

由于Shinjuku是一个单地址空间操作系统，其组件之间的通信发生在共享内存上。我们为每一对通信线程使用专用的缓存线对。与IX和ZygOS类似，Shinjuku利用Dune系统进行流程虚拟化。对于Dune, Linux和Dune内核模块运行在VMX根模式环0中，其中hypervisor将运行在虚拟系统中。Shinjuku运行在VMX非根模式环0中，其中将运行一个来宾操作系统。这允许它在将控件与数据平面分离时使用非常低的开销中断。使用Shinjuku的应用程序上下文可以在VMX非根模式环0或环3中运行。

**3.2快速抢占**

为了使用微秒延迟的抢占调度，Shinjuku需要快速抢占。一种简单的方法是让调度程序使用Linux信号通知工作者。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机器上大约为2.5µ秒)。它们需要用户到内核空间的转换以及一些内核处理。

**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ipi。每个核心都有一个本地APIC，一个I/O APIC连接到系统总线上。为了发送一个IPI，发送核心在其本地APIC中写入寄存器，该寄存器通过I/O APIC将中断传播到目标核心的APIC，而目标核心的APIC又将执行向量传递给中断处理程序。

我们通过虚拟化本地APIC寄存器扩展了Dune以支持ipi。当核心A上的非根线程写入其虚拟APIC以将中断号V发送到核心B时，这会导致VM退出到以根模式运行的Dune。Dune将V写入到核心B发布的中断描述符，然后使用真正的APIC将中断242发送到核心B。这导致核心B执行VM退出到Dune中的中断处理程序，该处理程序在恢复应用程序时将中断编号V注入到非根模式。

如**表1**所示，使用ipi的这种普通抢占实现略快于Linux信号，但由于发送方和接收方的VM出口成本，仍然存在很大的开销。

**优化中断传递**。我们首先关注使用已发布的中断在接收核心B (Shinjuku worker)上删除VM出口，这是一个x86特性，用于在没有VM出口的情况下接收中断。为了启用发布中断，B上的Dune配置其硬件定义的VM控制结构(VMCS)，以将中断242识别为特殊的发布中断通知向量。B还向VMCS注册了它发布的中断描述符。内核A在写入虚拟APIC时仍然执行VM退出。A上的Dune代码将V写入B发出的中断描述符中，并将中断242发送给B。然而，B随后直接注入中断V，没有VM出口。表1显示，消除接收端VM出口可以减少54%的接收端开销(从2662个周期减少到1212个周期)。这允许频繁地抢占工作线程，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，如果不改变硬件(比如支持轻量级用户级中断)，它就不能得到显著的改进。

**优化中断发送**。最后，我们通过信任Shinjuku dispatcher直接访问真实(非虚拟)APIC来删除发送核心(dispatcher线程)上的VM出口。使用扩展页表(EPT)，我们将其他核心的已发布中断描述符和本地APIC寄存器映射到Shinjuku调度程序的来宾物理地址空间。因此，调度程序可以直接发送一个IPI，而不会导致VM退出。表1显示，消除发送端VM出口可以将发送端开销降低到298个周期(在2GHz系统中为149ns)。这提高了调度程序的可伸缩性，并允许它提供更多的重新服务每秒任务和/或更多工作线程(核心)。

表1给出了用于支持新宿抢占的中断传递的发送端和接收端优化组合的结果。较低的发送端开销(298个周期)使得构建一个集中的、抢占式的分派器变得可行，该分派器每秒处理数百万个调度操作。低接收端开销(1212个周期)使得每5µ秒抢占一次请求以调度更长的请求而不浪费超过10%的worker吞吐量是可行的。

**3.3低开销背景开关**

当请求被调度到空闲核心或抢占时，我们在每个worker中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，在普通的Linux进程中，开销非常大，在Dune进程中，开销会增加一倍。在切换过程中，swapcontext需要一个系统调用来设置信号掩码，这需要在《沙丘》中有一个VM出口。交换上下文中的其余工作—即。保存/恢复寄存器状态和堆栈指针——不需要系统调用。

表2评估上下文切换优化。首先，我们跳过设置信号掩码，从而消除了系统调用，使Dune与普通Linux不相上下。这引入了属于同一个应用程序的所有任务需要共享相同的信号掩码的限制。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者上下文时，我们必须保存FP寄存器，因为它们可能已经用于请求处理，但是我们不需要为工作者上下文恢复它们。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中的最后两个选项在工作核心中进行上下文切换。总体成本从36到109个周期(2GHz系统为18到55ns)。

**3.4抢占式调度**

集中调度和快速抢占和上下文切换机制允许新宿实施抢占调度策略。我们开发了两种策略，这两种策略的区别在于我们是否可以在请求类型之间进行先验区分。策略依赖于频繁抢占为任何工作负载提供近乎最佳的尾部延迟，对于低分散工作负载近似c- ffs，对于所有其他情况近似PS。

单队列(SQ)策略:该策略假设我们不预先区分请求类型，并且对尾部延迟有一个单一的服务水平协议(SLO)。例如，在搜索服务中，我们无法先验地知道哪些请求的服务时间会更长。所有传入的请求都放在一个FCFS队列中。当一个worker空闲时，调度程序将位于队列头部的请求分配给它。如果快速处理请求，则此策略作为集中式ffs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时间的请求(在我们的实验中是5到15µ秒)，并且假设队列不是空的，就会抢占它。请求被放回到队列中，工作线程被分配到队列当前头部的请求。图1中通过模拟评估的c-PRE-SQ策略就是这个单一队列策略。

多队列策略:该策略假设网络子系统可以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[33]中分离出来，或者为不同的请求类型使用不同的端口。Linux已经支持使用eBPF[2]窥探数据包。每种请求类型可以有不同的尾部延迟SLO。调度程序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，此策略的操作与上面描述的单个队列策略相同。如果有多个队列非空，调度程序必须选择一个队列在某个工作者空闲或请求被抢占时提供服务。一旦选择了队列，调度程序总是在头部接收请求。

队列选择算法的灵感来自于BVT[24]，它是一种针对延迟敏感任务的进程调度算法。在BVT中，每个进程都有一个扭曲因子，用于量化其相对于其他进程的优先级。对于Shinjuku，我们需要一个类似的扭曲因子，在短期内支持具有较小目标延迟的请求，但也考虑具有较长延迟目标的请求的老化。由于Shinjuku调度请求和不长时间运行的进程具有优先级(如BVT)，下面所示的选择算法使用每个队列的目标SLO延迟作为输入(例如，目标第99百分位延迟)。对于位于每个队列头部的请求，算法使用时间戳来计算该请求类型在系统中已经花费的时间(排队时间)与SLO目标延迟的比率。该比率最高的队列将被选中。算法最初倾向于只能容忍较短排队时间的短请求，但最终选择可能已经等待了一段时间的长请求。每个队列的SLO是用户设置的参数。在我们的实验中，我们通过使用单队列策略单独运行每个请求类型来设置它，并使用观察到的99%延迟。这捕获了这样一种需求，即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。

一个被抢占的请求可以放置在队列的尾部以近似于PS，也可以放置在队列的头部以近似于c- ffs。这个选择可以由应用程序设置，也可以基于服务时间统计数据的在线测量。我们使用的经验法则是，对于多模式或重尾工作负载，请求应该放在队列的尾部，而对于轻尾工作负载，请求应该放在队列的头部。即使对于轻尾分布，也需要频繁的抢占，以便允许Shinjuku为其他请求类型的队列提供服务。图2中通过模拟评估的c-PRE-MQ策略就是这种多队列策略，当被抢占时，两种请求类型都被放置在相应队列的头部。

**3.5实现**

当前的Shinjuku实现基于Dune，需要x86-64系统[20]中的VT-x虚拟化特性。Dune可以移植到其他具有类似虚拟化支持的架构上。我们对《沙丘》的修改涉及1365 SLOC。新宿调度员和工作人员代码为2535 SLOC。我们在§4中使用的网络子系统是基于IX[16]的。前面提到的所有代码库都是C语言的。

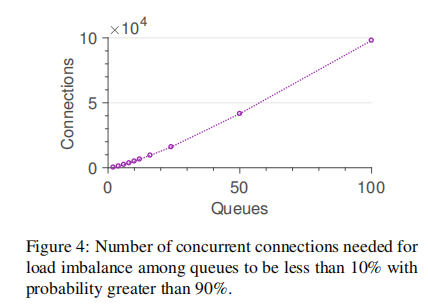
**API**:要使用新宿，应用程序需要注册三个回调函数:初始化全局应用程序状态的init()函数;init per core(int core num)函数，初始化每个工作线程的应用程序状态(例如局部变量或配置选项);reply \* handle request(request \*)函数，它处理单个应用程序级请求并返回指向应答数据的指针。

**上下文管理**:我们使用Linux ucontext库的修改版本进行上下文管理。上下文结构由特定于机器的已保存状态表示、信号掩码、指向上下文堆栈的指针和指向上下文的指针组成，当上下文完成执行时将恢复上下文。调度程序为来自内存池的每个请求分配上下文对象和堆栈空间。当请求上下文完成执行并由工作线程返回时，调度程序释放它们。

**线程间通信**:除了抢占之外，我们还使用一种低开销的共享内存通信方案，类似于[50]中使用的方案。运行在专用内核或超线程上的每对线程都通过共享的缓存线对进行通信，每个通信方向对应一条缓存线。发送线程用它想要发送的数据填充缓存行。请求或上下文指针，如图3所示。然后，它设置接收方轮询的字节的值，以通知它缓存线已经准备好读取了。这种方法需要两个缓存线状态转换，一个是从共享状态转换到独占状态，发生在发送方写入数据时，另一个是从独占状态转换到共享状态，发生在接收方读取数据时。通过缓存线发送和接收消息的平均往返延迟是211个周期。调度程序发送消息的最小工作量大约是70个周期，即在2GHz的机器中为35ns。这为调度程序可以处理的请求数量设置了一个理论上的上限28 MRPS，假设它所要做的只是在共享内存位置放置指向请求的指针并通知空闲的工作线程。

**3.6讨论**

硬件限制:§4表明，一个dispatcher线程每秒可以处理至少5M个请求，并且可以轻松地饱和一个有12个核和24个超线程的完整套接字。为了将单个应用程序扩展到更高的核心和/或套接字计数，我们必须提高调度程序吞吐量。我们使用的方法是让每个调度程序线程处理工作线程的一个子集，并使用NIC RSS特性将请求引导到不同的调度程序。一个相对简单的硬件特性将极大地提高调度程序的可伸缩性，即不同核心之间的低开销消息传递机制。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，并用于工作分配。



**连接计数**:IX和ZygOS使用RSS将请求分发给工作者。使用蒙特卡罗模拟，当我们增加核心数量时，我们计算出RSS将不平衡保持在10%以下所需的连接计数。如图4所示，它们需要16,000个连接(客户端或流)来避免拥有24个超线程的服务器上的不平衡。高连接数对于面向公共的服务(例如，公共负载均衡器或HTTP服务器)是常见的，但对于内部服务则不是。DCTCP项目[9]在每个1毫秒窗口内最多发现几百个到后端服务器的连接。相反，新宿使用RSS将请求分发给调度员。由于每个dispatcher可以管理数十个内核，因此Shinjuku不受§2中讨论的高连接(客户端或流)计数的要求。例如，300个连接足以在2个调度程序之间实现负载平衡。当一个调度员就足够了，新宿即使只有一个连接也能高效运行。

**替代调度策略**:新宿可以支持更多的调度策略，除了我们提出的两个。在未来的工作中，我们将探索将新宿与数据中心范围的分析工具[49]和在线实验工具[55]集成在一起，以便动态推断服务时间分布并相应地调整策略。我们还将探索微秒级的本地化和异构感知调度策略[30,27]。例如，考虑一个应用程序，它在响应客户机请求之前创建了很大的内存占用。在这种情况下，我们将避免抢占和上下文切换，因为多条缓存线将不得不移动到不同的核心，这可能非常昂贵。

**控制平面**:在线服务经历负载变化，例如每天的负载模式和峰值。因此，随着时间的推移调整新宿流程使用的工人数量是有意义的。Shenango[7]通过在微秒时间尺度上调整应用程序之间的核心分配来解决这个问题。我们计划探索两个系统集成的可能性。

**安全模型**:Dune内核模块[15]使用虚拟化硬件支持将Shinjuku进程与Linux内核和任何其他进程(普通Linux或基于Shinjuku的进程)隔离开来。Linux还可以随时从新宿进程中删除内核和网络队列。在新宿进程中，应用程序代码必须信任新宿运行时，如果应用程序上下文在VMX非根环0中执行，新宿运行时必须信任应用程序代码。例如，APIC寄存器映射到进程地址空间的事实意味着一个进程可以通过向特定核心发出大量中断来对另一个进程发起拒绝服务攻击。

我们测量了一个环3→环0→环3转换的成本只有84个周期。新宿的未来版本将在环3中运行应用程序代码，而新宿运行时将在VMX非根环0中运行，以非常小的开销消除这种攻击向量。而且，使用这种方法，应用程序代码中的错误只会导致上下文崩溃，而不会影响运行时系统。

**用户代码中的同步**:在线服务被设计成在多核上很好地运行。它们跨请求同步，但是同步时间短且不频繁，以实现可伸缩性。无论我们是否禁用或允许读写锁抢占，可伸缩的应用程序都将使用Shinjuku执行。我们目前在任何非线程安全代码期间禁用中断，使用调用安全(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内核回收内核的能力。内存分配代码是一种特殊情况，通常使用线程本地存储优化锁。我们预加载了我们自己的C和c++库版本，这些库在执行分配函数时禁止中断(因此也禁止抢占)。如果这些函数需要很长时间，就会影响到用新宿观测到的尾潜伏期。

任何在请求中频繁使用粗粒度锁或争用锁的应用程序，无论在任何系统(包括Shin-juku)上采用何种调度策略，伸缩性都很差。