高技术通讯 2023 年 第33 卷 第8 期:781-790

doi:10.3772/j.issn.1002-0470.2023.08.001

面向应用定义优先级调度的用户态协议栈研究①

沈逸凡②*** 张文力* 刘 珂* 陈明宇***

(*中国科学院计算技术研究所 北京 100190)

(**中国科学院大学 北京 100049)

摘 要 针对数据中心负载中混杂请求对延迟敏感型请求响应尾延迟产生干扰,而现有研究无法在不同负载场景下为延迟敏感型请求提供灵活优先调度的问题,本文提出了应用定义优先级调度的用户态协议栈。该设计利用用户态协议栈在数据中心请求处理所处的关键位置,支持上层应用根据负载特征灵活定制优先级识别与调度策略,并由协议栈为优先级识别提供数据包、传输控制协议(TCP)流等丰富的状态信息,实现了不需要改动网络协议栈,就可以对不同负载场景实现灵活的应用定义优先级识别与调度,从而避免延迟敏感型请求受到其他混杂负载请求带来的排队延迟与阻塞延迟干扰。实验结果表明,在不同的负载场景下,通过灵活准确的应用定义优先级调度,可以将延迟敏感型请求的响应尾延迟降低98.5%,有效保障了用户体验。

关键词 数据中心网络;优先级;任务调度;用户态协议栈

0 引 言

数据中心服务器需要处理来自不同应用的多种类型的混杂请求,这些请求往往带有不同的服务等级目标(service level objective,SLO)。例如,在阿里巴巴的数据中心服务器上[1]同时运行有对响应延迟不敏感的批处理、机器学习等任务,并同时需要对延迟敏感的网页服务、社交网络等交互式应用的请求进行服务。非延迟敏感型请求会对延迟敏感型请求造成抢占中央处理器(central processing unit,CPU)、引发排队阻塞等干扰,导致响应尾延迟上升。因此数据中心服务器需要对混杂的请求负载进行分类与调度,保证延迟敏感型请求的响应尾延迟,以提高用户体验。

在不同的负载场景中,数据中心服务器需要定义不同的策略以实现对延迟敏感型请求的准确甄别与调度。例如,在来自不同应用或是同一应用的不同类型的请求间,可以通过关键词匹配识别请求类型来进行优先级分类与调度(比如物联网服务器场景[2]中区分用户请求和长连接心跳包);而在网页服务中则需要根据对服务响应时间的估计,对基础响应延迟较低的请求优先进行服务,因为这些请求的延迟波动会对用户体验造成更大影响[3](相比5 s才能加载的网页,1 s的响应延迟波动对1 s就能加载的网页的用户体验影响更大)。

在数据中心服务器上,网络协议栈负责从网卡收发数据包并进行网络协议处理,处在每个请求处理的关键路径上,可以获得全局的状态信息,因此是实现请求负载分类调度的理想位置。然而,现有的网络协议栈并不能有效地对数据中心混杂的负载进行分类调度。传统操作系统使用的内核协议栈可以使用eBPF(extended Berkeley packet filter)[4]等方式实现丰富的分类调度策略,但面对不同应用和场景

① 国家重点研发计划(2016YFB1000203,2017YFB1001602)和国家自然科学基金(62072439)资助项目。

② 男,1992 年生,博士;研究方向:计算机系统结构,用户态协议栈;联系人,E-mail:shenyifan@ict.ac.cn。

(收稿日期:2022-03-28)

时需要重新编写 eBPF 代码,缺少灵活性,且内核协议栈的高额开销导致其已无法满足当下数据中心应用的性能需求[5-6]。用户态协议栈采用内核旁路技术,避免了内核协议栈的性能瓶颈,为数据中心应用提供高性能的网络服务[5-11]。文献[2]使用用户态协议栈,通过对请求添加静态优先级标签实现在服务端的优先队列调度,避免了排队延迟,但需要客户端与智能网卡的配合,当面对不同应用及负载场景时缺乏灵活性。文献[11]也使用用户态协议栈,通过识别请求类型,优先调度处理时间短的请求,以避免复杂请求阻塞带来的尾延迟,但只能在应用层进行调度,且只支持根据请求头部类型信息进行分类。这些系统在服务器上的应用及负载场景发生变化时,都要对系统进行复杂的修改才能实现对新环境负载的有效调度,且受设计架构约束,无法对某些场景进行有效的调度。

针对上述问题,本文提出了面向应用定义优先级调度的用户态协议栈。为实现不同负载场景下的延迟敏感型请求的响应延迟保证,本研究支持每个应用向协议栈制定符合自身请求协议及负载特征的优先级识别与调度策略,并由用户态协议栈提供优先级识别调度所需的状态信息及调度支持,实现在不修改网络协议栈的情况下同时对不同数据中心应用实现灵活准确的优先级识别调度。同时,数据中心供应商也可根据常见的调度策略及自身负载特征,为所有应用定制默认的优先级识别调度方案。

本文组织如下,第 1 节介绍研究背景与相关工作,第 2 节介绍如何在用户态协议栈支持应用定义的优先级识别及调度,第 3 节介绍系统实现及应用移植,第 4 节介绍实验评测方法及结果,第 5 节对全文进行了总结。

1　背　景

1.1　混杂请求对尾延迟的影响

在阿里巴巴[1]、Google[12]等数据中心服务器上,为提高服务器资源利用率,混杂有来自不同应用、带有不同负载特征与服务等级目标的请求,这些请求相互之间形成干扰,对响应延迟造成波动。混杂请求间的相互干扰主要有 2 种类型:大量延迟非敏感型请求造成的排队延迟和处理时间长的复杂请求占用 CPU 导致的阻塞延迟。

请求间相互干扰的一种体现是,当负载高峰时,大量的对延迟不敏感的请求对请求处理的各个阶段造成排队,导致少量对延迟敏感的请求因为排队而引发响应尾延迟上升,本文通过实验对该问题进行了验证。在实验中,仿照物联网(Internet of Things, IoT)服务器场景大量低优先级请求混合少量高优先级负载的模式[2],在客户端生成包含 5% 延迟敏感型请求和 95% 非延迟敏感型请求的突发式负载。在服务端,使用基于 mTCP[5] 的一个 echo 服务,对收取到的每个请求进行 10 ms 的处理后原样返回响应。本文记录了延迟敏感型请求的响应尾延迟以及它们在请求处理各个阶段的排队延迟(均取 99th 尾延迟),并测量统计了模拟理想情况(非延迟敏感型请求在网卡接收后就丢弃)下响应尾延迟。如图 1 所示,混杂负载的延迟敏感型请求相比在理想情况下要高 72 倍。其中,延迟敏感型请求在传输控制协议(transmission control protocol, TCP)接收缓冲区的等待时间约占总体延迟的 67%,这是在等待上层应用处理完大量的非延迟敏感型请求后才被处理。此外,延迟敏感型请求数据包在被从网卡接收队列收取到收包缓冲区后,在收包缓冲区的排队等待时间约占总体延迟的 33%。这是因为网络协议栈在处理每个 TCP 请求与响应数据包时都需要几百纳秒的开销,延迟敏感型请求数据包要等待大量非延迟敏感型请求数据包被协议栈接收处理后才能进入协议栈处理。从该实验中可以看到,混杂请求的排队延迟会对延迟敏感型请求的响应延迟产生严重干扰。

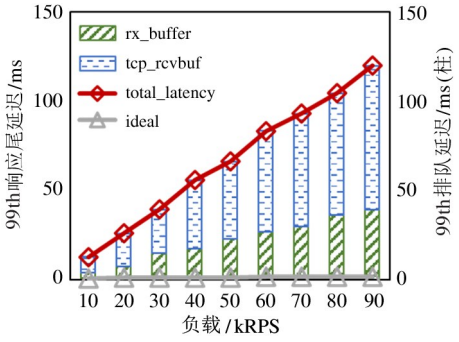


图 1　排队延迟对响应尾延迟的影响

请求间相互干扰的另一种方式,是处理时间长的复杂请求占用 CPU 导致对处理简单的延迟敏感型请求产生阻塞延迟。在证明该问题的实验中,仿照数据库服务场景大量微秒级简单读/写请求混合少量毫秒级遍历请求的负载模式[10],使用 99.5% 处理时间为 1 μs 的延迟敏感型请求混合有 0.5% 处理时间为 1 ms 的复杂请求。在 mTCP 上测试了响应尾延迟,并模拟计算了理论最优响应尾延迟(使用优先级抢占调度的情况下)。如图 2 所示,因为复杂请求占用 CPU 导致的阻塞,延迟敏感型请求的响应尾延迟以阶梯状呈现快速上升,达到了理论最优值的 62 倍。

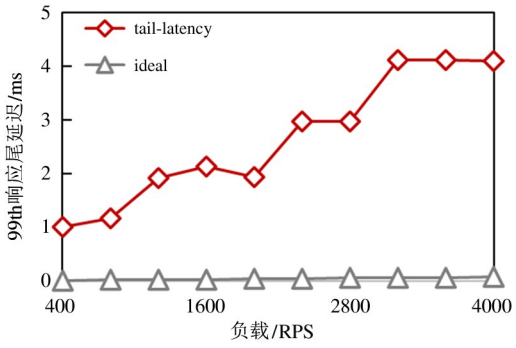


图 2　复杂请求阻塞延迟对响应尾延迟的影响

1.2　相关研究现状

请求间的相互干扰会导致延迟敏感型请求响应尾延迟上升,因此数据中心服务需要对请求类型进行识别并进行优先级调度。

传统操作系统提供的内核态协议栈使用内核态驱动通过网卡硬中断进行数据包收取,并在随后产生的软中断中对数据包进行网络协议处理,再由上层应用调用套接字(socket)接口通过系统调用进行数据收发。在使用内核协议栈时,应用可以通过 eBPF[4] 向内核的网络数据包处理流程的特定环节注册回调函数,根据数据包内容及其他内核协议栈相关信息实现丰富的调度策略。但是内核协议栈因其中断、系统调用、各级数据结构的锁等高额开销,导致整体性能无法满足当今数据中心应用的性能需求[5-6]。同时,开发人员需要为每个调度环节单独进行 eBPF 代码实现。如果有多种调度策略,还要将它们整合到一个函数中。受限于内核的保护机制,内核态的 eBPF 代码很难与用户态应用进行通信,不同环节的 eBPF 也只能分别对每个数据包进行调度,很难实现不同层次间的协同调度。此外,随着操作系统内核的更新,开发人员还可能需要对 eBPF 代码重新进行实现。

近年来,数据中心应用普遍采用用户态协议栈[5-11]的设计避免传统操作系统内核协议栈的性能瓶颈问题。用户态协议栈使用用户态驱动对网卡进行轮询,收取的数据包直接被送到用户态,在用户态完成网络协议处理后直接由上层应用进行收取,避免了操作系统内核的中断、系统调用等开销,并通过批处理、内存池管理等方式,大幅提升了网络处理能力。相当部分用户态协议栈[5-7]采用先到先服务的处理,不考虑任务调度问题。也有一些用户态协议栈以不识别请求内容的形式对任务进行调度,例如,ZygOS[8]、TAS (TCP acceleration as a service)[9]等用户态协议栈以 task-stealing[13] 的方式对任务进行调度,空闲的处理核可以从忙碌核的任务队列“偷”任务进行处理,一定程度上避免了复杂请求导致的阻塞延迟,但频繁的任务偷取会引发严重的核间竞争导致系统开销上升,且当所有处理核都被复杂请求阻塞时还是会导致阻塞延迟。文献[10]采用时间片的方式,由中心化协议栈线程将执行时间过长的请求线程中断并调度新任务执行,以避免复杂请求导致的阻塞延迟,但会产生大量的上下文切换开销。并且,不识别请求内容的任务调度都无法避免排队延迟对延迟敏感型请求响应尾延迟的干扰。

LNS (labeled network stack)[2]是一个利用标签实现请求类型识别及优先级调度的用户态协议栈框架。LNS 在客户端将请求优先级以标签的形式附着在请求数据包中。该优先级标签在服务端由智能网卡进行硬件识别处理,并根据优先级标签,在网卡队列调度、协议栈处理、事件框架、上层应用处理等各个阶段将请求放入不同优先级队列,以实现对高优先级请求的优先调度处理,避免了高优先级请求受到排队延迟对响应尾延迟的干扰。但 LNS 需要客户端的配合并需要专用网卡硬件,无法大规模部署并缺乏支持丰富应用的灵活性。此外,LNS 只能根据静态的请求类型标签进行优先级调度,无法支持

根据应用在服务端对请求处理时间的测量调度不同复杂度的请求以避免阻塞延迟。

Perséphone[11]在网络协议栈处理包后,在将请求提交到上层应用处理前,使用一个应用注册的分析回调函数对请求内容进行识别(读取应用层协议头类型,例如超文本协议读请求、Redis 写请求等),将请求分为不同类型。通过监控每个类型请求的平均处理时间,为短请求和复杂请求分别赋予高优先级和低优先级。在把请求分发给 worker 处理核进行请求处理时,优先分发高优先级的请求,并预留部分处理核只用于高优先级请求处理,从而有效避免了复杂请求对延迟敏感型请求产生的干扰。但是 Perséphone 在传输层后才进行优先级识别与调度,无法避免延迟敏感型请求在协议栈处理过程中的排队延迟。此外,Perséphone 只根据请求执行时间进行优先级调度,但实际生产环境中可能存在大量执行时间同样不长但对响应延迟并不敏感的低优先级请求,引发排队延迟导致的尾延迟上升。理论上 Perséphone 的架构也可以实现其他类型的调度策略,但是其调度函数只能使用应用层提供的信息,且无法在不修改系统实现的情况下动态调节优先级调度策略。

通过对以上问题的分析,本文提出了支持应用定义优先级调度的用户态协议栈,通过支持各应用灵活制定针对不同负载场景的优先级识别策略,并加以由应用定义的优先级调度,避免延迟敏感型请求在请求处理中受到排队延迟与阻塞延迟影响,以保障其响应尾延迟及用户体验。

2 应用定义优先级识别与调度设计

为了在用户态协议栈上实现对延迟敏感型请求的灵活调度,本文提出了应用定义优先级调度,即让用户态协议栈在不修改协议栈本身的情况下,支持数据中心维护人员及各类数据中心应用根据自身负载特征制定对延迟敏感型请求的识别与调度策略。

为此,设计了尽早对延迟敏感型请求进行识别以避免排队延迟的应用定义驱动层优先级识别(详见 2.1 节),设计了支持应用定制丰富的优先级识别策略的应用定义传输层优先级识别(详见 2.2 节),并设计了用户态协议栈对优先级信息的传递以及对应用定义优先级调度的支持(详见 2.3 节)。通过以上设计,用户态协议栈能够灵活支持不同负载场景下的延迟敏感型请求优先级调度,为其响应尾延迟与用户体验提供保障。

2.1 应用定义驱动层优先级识别

驱动层负责从网卡收发数据包,是数据包进入服务端软件栈的第一步,因此在驱动层实现优先级识别可以避免后续请求处理阶段的排队延迟。

如图 3(a) 所示,应用定义驱动层优先级识别让应用以注册回调函数的形式将优先级识别函数 *pri _filter*( )加入到驱动层处理。在用户态协议栈从网卡接收数据包并放入接收缓冲区时,对每一个数据包都调用该回调函数。回调函数返回的优先级会被写入数据包元数据,数据包也会被放入对应优先级的接收缓冲区中,在后续处理中进行优先级调度。

应用定义的驱动层优先级识别必须能够快速对数据包进行优先级区分,否则可能对后续所有处理产生阻塞。得益于直接数据输入输出(data direct I/O, DDIO)[14]技术的支持,应用定义的驱动层优先级识别函数所使用的数据(数据包元数据与数据包内容)都会被存放在处理器 cache 中,避免了访存延迟与开销,进行简单的协议头解析与标志位识别的额外开销在 50 ns 以下。

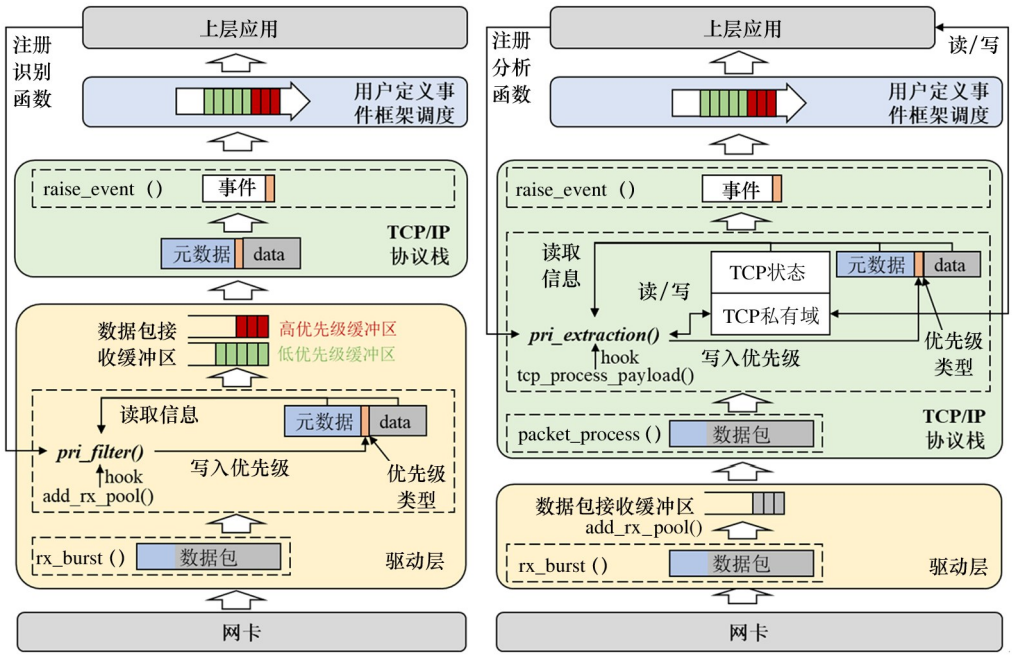
在应用定义的驱动层优先级分类函数中,可用于辅助优先级识别的输入信息包括数据包的元数据(例如数据包长等)以及数据包本身数据内容(例如协议头固定标志位提取等)。以下为使用驱动层优先级识别的 2 个典型场景举例。

场景 1 吞吐密集型应用在传输大块数据时通常每个数据包都会达到最大传输单元(maximum transmission unit,MTU)上限,因此可根据数据包元信息中的数据包长度进行优先级识别,将长度超过阈值的数据包设置为低优先级。

场景 2 对数据包请求头部进行优先级关键字提取,识别客户端写入的优先级标签(兼容 LNS)或是对请求类型进行识别(区分读请求及写请求等)。

2.2 应用定义传输层优先级识别

传输层[15]负责为应用进程提供端到端的通信



(a) 应用定义驱动层优先级识别　　(b) 应用定义传输层优先级识别

图 3　应用定义优先级识别架构示意图

服务。在传输层,除数据包信息外,还可以获取流状态等信息,并通过 socket 抽象与上层应用通信,在此可以利用丰富的语义信息实现灵活准确的优先级识别策略。本文在传输层主要讨论对 TCP 协议的支持与处理,但该设计思想适用于所有传输层协议。

如图 3(b) 所示,应用定义传输层优先级识别让协议栈支持应用以注册回调函数的形式为每一条 TCP 连接注册传输层优先级识别函数 *pri _ extraction*( )。每当 TCP/IP 协议栈处理到带有 TCP 负载的数据包时,会对该数据包调用当前 TCP 连接对应的优先级识别回调函数进行优先级信息分析。分析得到的优先级被写入数据包的元数据及向事件框架添加的事件中,在后续处理中进行优先级调度。

应用可以通过多种方式灵活地注册传输层优先级识别回调函数。数据中心供应商可根据常见的调度策略及自身负载特征,定制默认的优先级识别方案。应用可以向 TCP 的 listener 注册优先级识别函数,该识别函数会成为从该 listener 接收到的 TCP 连接的默认优先级识别函数,从而让一个数据中心应用的所有 TCP 连接都使用根据该应用请求特征定制的优先级识别函数。应用也可以通过 socket 接口为某个 TCP 流注册特定的优先级分析函数。甚至在优先级识别函数处理过程中,可以根据状态的改变,为当前 TCP 流更换使用新的优先级识别函数。通过为每个 TCP 连接注册独立的优先级识别函数,可以避免单个优先级识别函数处理过多应用类型导致处理过程冗长,从而避免对系统整体性能产生影响。

传输层优先级识别可以利用丰富的数据包与 TCP 流状态信息对数据包优先级进行识别。以下为使用 TCP 状态进行优先级识别的 2 个典型场景的举例。

场景 3　对于来自广域网高丢包高延迟链路的请求,用户对响应延迟的小幅度波动并不敏感[3],而相同的响应延迟波动会对低网络延迟用户的体验产生较大影响。因此根据 TCP 连接拥塞控制提取的往返延迟(round trip time, RTT)及丢包率等信息,将来自网络状况较差的连接的请求标记为低优先级。

场景 4　根据当前 TCP 连接已经传输的字节数,将已接收数据量超过阈值的 TCP 连接视为来自吞吐密集型应用,将其请求标记为低优先级。

当进行传输层优先级识别时,如果该数据包已经通过驱动层优先级识别获取了优先级,可以在传

输层优先级识别过程中读取数据包元数据中的优先级信息作为参考,并可以对其进行覆盖。

为支持应用定义优先级函数进行跨数据包的有状态分析,以及与上层应用的通信,如图 3(b) 所示,为每个 TCP 连接设置了长度为 64 字节(一个 cacheline)的专用于优先级识别的私有域(在系统中,该 64 字节占 TCP 状态信息结构体的 7%)。传输层优先级函数可对所属 TCP 连接的私有域进行读写操作,将优先级分析的中间状态暂存,从而实现有状态的分析。上层应用也可通过 socket 接口直接访问该私有域,从而实现应用与优先级识别函数的通信。此外,得益于用户态协议栈中协议栈处理与应用线程位于同一个命名空间(namespace),优先级识别函数可以直接使用上层应用传递的指针及对应的内存空间。以下为利用私有域进行优先级识别的 2 个典型场景的举例。

场景 5　当请求长度超出单个数据包长度上限时,优先级分析函数在私有域中记录当前请求未读取长度或匹配结束符,该 TCP 连接后续属于同一请求的数据包将被标记为与该请求第一个数据包相同的优先级。

场景 6　上层应用在处理请求过程中统计每个类型请求的执行时间并进行估计,通过私有域下发到优先级识别函数,将预计执行时间超过阈值的请求标记为低优先级。

2.3　优先级信息传递与事件框架优先级调度

应用定义优先级识别得到的优先级信息需要在协议栈处理的各个阶段间进行传递。如图 3 所示,由应用定义优先级识别函数返回的优先级会被写入到对应请求数据包的元数据的优先级标志位中,并随数据包被传递到后续阶段。当请求数据包被放入 TCP 接收缓冲区时,协议栈需要通过类 epoll[16] 的事件框架生成收包事件以提醒上层应用收取数据,并将数据包的优先级写入到事件结构体中,由事件框架根据事件优先级对其进行调度。

协议栈需要根据优先级信息在各个处理阶段对请求进行调度。为避免 1.1 节所分析的排队延迟对延迟敏感型请求响应尾延迟的影响,在协议栈从驱动接收缓冲区收取数据包以及上层应用从事件队列收取事件时,采用了类似 LNS 的基于数据包和事件优先级的优先队列调度[2]。

本文设计了应用定义的事件框架优先级调度。协议栈支持上层应用调用事件框架接口时收取指定优先级事件,从而实现更复杂的应用定义优先队列调度,例如,避免低优先级请求得不到处理而饿死。同时,上层应用可通过配置事件框架,将不同优先级的事件只分发到指定处理核。例如,场景 6 中将识别到的低优先级复杂请求分流到单独处理核进行处理,避免产生 1.1 节所分析的阻塞延迟。

3　系统实现

本文在 LNS[2] 的基础上对支持应用定义优先级调度的用户态协议栈进行了实现,在利用 LNS 已有的用户态协议栈优先级调度框架基础上,增加了应用定义的优先级识别与应用定义的事件框架优先级调度支持。修改与增加的代码量在 2000 行左右。如图 4 所示,在 LNS 的基础上,在驱动层,将智能网卡驱动更换为数据平面开发套件(data plane development kit, DPDK)的通用网卡驱动,并加入应用定义优先级识别;在传输层,加入应用定义传输层优先级识别支持,并对 TCP 连接对象增加了私有域等数据结构支持;在事件框架,在优先队列调度的基础上增加了应用定义事件框架优先级调度的支持。

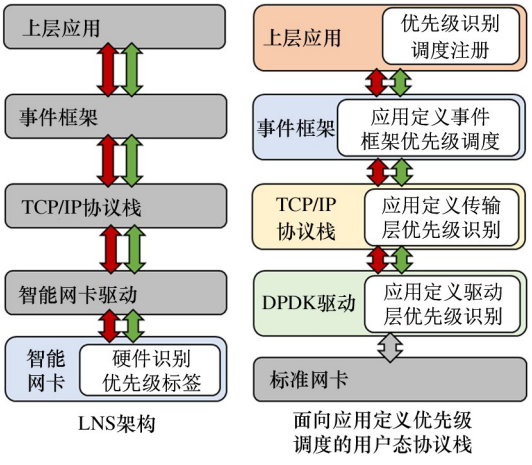


图 4　面向应用定义优先级调度的用户态协议栈实现

此外,本文还移植了 lighttpd[17],一个开源 Web 服务器,以验证应用定义优先级调度的用户态协议

栈在真实应用上的效果。选择支持用户态协议栈与多线程模式的 mTCP 版本 lighttpd 进行移植[5],并加入了基于场景 3 的优先级识别函数注册等配置修改,根据拥塞控制模块的 RTT 统计及丢包统计,将网络状况好的 TCP 流的请求标记为高优先级。修改与增加的代码量在 100 行左右。

4 实验与评测

本节验证了应用定义优先级调度在不同的负载场景下,可以通过定制优先级识别策略实现对延迟敏感型请求的灵活准确识别,并通过优先级调度保证其响应尾延迟,保障用户体验。

4.1 测试环境

本实验中使用 1 台客户端和 1 台服务器,对应用定义优先级调度的用户态协议栈的系统效果进行评测。服务端和客户端服务器均使用 E5-2630 v4 2.2 GHz 处理器,64 GB 内存,Intel82599 10 Gbps 网卡。

在服务端,部署了如第 3 节所述的支持应用定义优先级调度的用户态协议栈;实现了一个虚拟负载服务应用(synthetic server),对收取到的每个请求一段时间处理后返回响应,请求处理时间由请求内的相关域指定。

在该应用上,使用应用定义驱动优先级识别实现了对 LNS 的模拟,以避免使用专用网卡硬件,该调度效果与 LNS 基本完全一致;还在协议栈与应用层间增加了任务调度器,实现了对 Perséphone 的 DARC(dynamic application-aware reserved cores)调度算法[11]的模拟,并将优先级调度支持范围扩展到相同处理时间的不同类型请求上。将这 2 种调度方式与本文设计的应用定义优先级调度(application-defined priority scheduling, ADPS)进行了对比。此外,还对移植后的 lighttpd 进行了测试。

在客户端,使用一款开源负载发生器 MCC(massive concurrent connection)[18]来进行流量生成。MCC 可以生成突发式负载,将每秒生成的请求以网络线速的速度进行发送,以测试服务端在面对突发流量时的响应尾延迟。MCC 可以生成虚拟负载请求,在请求内部写入每个请求在服务端需要处理的时间,并在请求头部写入请求类型以进行优先级识别。MCC 同样可以生成超文本传输协议(hyper text transfer protocol, HTTP)请求负载,用于 lighttpd 的测试。每组实验重复 4 次,取其平均值作为结果。

4.2 应用定义驱动层优先级识别

测试应用定义驱动层优先级识别对延迟敏感型请求响应尾延迟的影响(场景 2)。使用 MCC 生成虚拟负载请求,每个请求处理时间为 10 ms,5% 的请求通过类型被标记为高优先级延迟敏感型请求,并统计该类型请求的 99th 响应尾延迟。在服务端,使用应用定义驱动层优先级识别对数据包优先级进行识别,并与 Perséphone 在协议栈后进行优先级识别调度的方式进行了对比。

如图 5 所示,使用应用定义驱动层优先级识别,由于有效地避免了高优先级请求的排队延迟,延迟敏感型请求的响应尾延迟相比没有优先级调度时降低了 98.5%。而 Perséphone 虽然在协议栈处理后对请求优先级进行识别并进行调度,避免了在应用层的排队延迟,但延迟敏感型请求依然会在数据包接收缓冲区等待协议栈处理时经历排队延迟,因此延迟敏感型请求的响应尾延迟比使用应用定义驱动层优先级识别时高 24.5 倍。

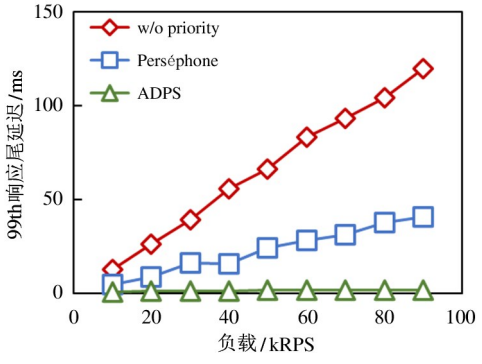


图 5 应用定义驱动层优先级识别响应尾延迟

此外,场景 1 与场景 4 的表现与本实验有所重合。场景 1 与场景 4 优先级调度目的均为区分延迟敏感型应用请求与吞吐密集型应用请求,方法区别为场景 1 在驱动层通过数据包长度区分,场景 4 在传输层通过 TCP 流传输数据量区分。实验发现,通过将吞吐密集型应用请求调度为低优先级,可以有

效降低延迟敏感型请求响应尾延迟。在传输层通过 TCP 连接数据量识别请求类型时(场景 4)表现与图 5的 Perséphone 基本一致,而使用应用定义驱动层优先级识别时(场景 1),因为避免了数据包在接收缓冲区排队,能够进一步降低延迟敏感型请求响应尾延迟,并与图 5 的 ADPS 基本一致。

4.3　应用定义事件框架优先级调度

本实验验证了应用定义事件框架优先级调度在混合有不同处理复杂度的请求时对延迟敏感型请求尾延迟的保证(场景 6)。使用 MCC 生成虚拟负载请求,99.5% 处理时间为 1 μs,0.5% 处理时间为 1 ms,并统计 1 ms 请求的响应尾延迟。在服务端,使用应用定义驱动层优先级识别将 1 μs 请求定义为高优先级,1 ms 请求定义为低优先级。在事件框架使用应用定义优先级调度,将低优先级事件分流到一个特定核进行处理,该核只负责处理低优先级事件,以避免产生阻塞干扰。同时,对比了 LNS 只进行优先级队列调度的模式,以及没有优先级调度下的响应尾延迟。

如图 6 所示,在没有优先级调度时,受到复杂请求的阻塞干扰,延迟敏感型请求响应尾延迟呈阶梯状迅速上升。LNS 通过优先级队列实现了响应尾延迟的大幅下降,但因为处理核在处理完高优先级请求后开始处理低优先级的复杂请求,导致后续到达的高优先级请求被阻塞,引发阻塞延迟。而使用应用定义事件框架优先级调度的情况下,低优先级的复杂请求完全不会对高优先级请求处理产生任何影响,有效地将响应尾延迟降低到没有优先级调度时的 1.6%。

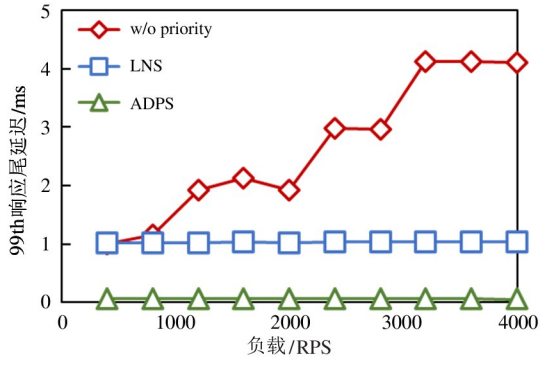


图 6　99.5% 1 μs +0.5% 1 ms 混合请求响应尾延迟

4.4　应用定义传输层跨数据包请求优先级识别

本实验验证了应用定义传输层优先级识别中使用 TCP 流私有域进行跨数据包有状态优先级分析的作用(场景 5)。用 MCC 生成虚拟负载请求,每个请求执行时间为 100 μs,5% 的请求通过类型被标记为高优先级延迟敏感型请求,并统计该类型请求的 99th 响应尾延迟。实验分 2 次进行,请求长度分别为 1024 字节(需要 1 个 TCP 包传输)和 4096 字节(需要 3 个 TCP 包传输),请求的长度被写在请求头部。在服务端,使用应用定义传输层优先级识别,根据请求头部中的请求类型判断优先级。当请求长度超过单个 TCP 包上限时,记录当前未读取数据字节数及对应优先级,将该 TCP 流后续属于该请求的数据包标记为相同优先级。使用应用定义驱动层优先级识别来复现没有客户端配合时的 LNS 调度。因为没有在每个数据包内加入优先级标签,当请求长度超过一个 TCP 包长度时,LNS 只能识别第一个数据包的优先级,并将该请求的后续数据包标记为低优先级。

如图 7 所示,在请求长度为 1024 字节时,应用定义优先级调度和 LNS 都能对延迟敏感型请求进行有效的优先级调度,尾延迟为无优先级调度时的 6%。但当请求长度上升到 4096 字节时,因为 LNS 无法对请求后几个数据包进行正确优先级识别,导致延迟敏感型请求受到排队延迟影响。LNS 在 4096 字节时延迟敏感型请求的响应尾延迟还是优于无优先级调度。这是因为在上层应用收取到高优先级收包事件时会尝试收取当前 TCP 连接所有数据,如果该请求的所有数据包都已在TCP接收

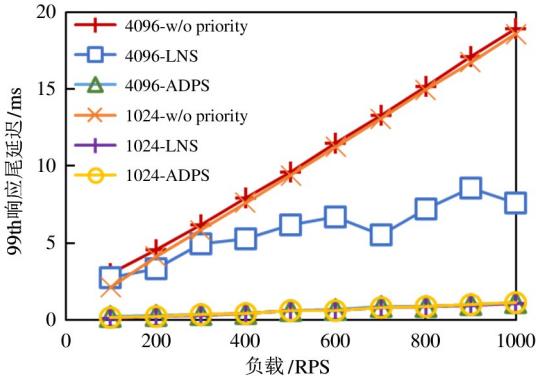


图 7　不同长度的混合优先级请求响应尾延迟

缓冲区中,则高优先级请求立刻可以得到处理。Perséphone 没有支持跨数据包请求优先级识别,表现与 LNS 基本一致。

4.5 应用定义传输层 RTT 优先级调度

本实验验证了应用定义传输层优先级识别中使用 TCP 相关状态判断网络状态,从而对请求做出优先级调度的效果(场景 3)。该场景的依据,来自于文献[3]。在不同的请求响应延迟阶段,响应延迟波动对用户体验质量(quality of experience,QoE)的影响是不同的,基本延迟较低时响应延迟波动会带来更大的 QoE 损失。

用 MCC 与客户端保持 1000 条长连接发送 HTTP 请求,并通过流量控制工具,为 50% 的 TCP 连接增加 50 ms 的网络延迟(网络状况较好),为剩余 50% 的 TCP 连接增加 200 ms 的网络延迟和 1% 的随机丢包(网络状况较差)。分别统计这 2 种 TCP 连接的响应延迟,并根据文献[3]对 Web 应用场景的 QoE 模型评估用户体验(使用归一化值)。在服务端,使用支持应用定义优先级调度的 lighttpd 对每个 HTTP 请求经过后端处理后返回长度为 1 MB 的响应,通过应用定义传输层优先级识别,根据拥塞控制模块的 RTT 统计及丢包统计,将网络状况好的 TCP 流的请求标记为高优先级。

如表 1 所示,网络状况较好的 TCP 连接的请求的 99th 响应尾延迟有优先级调度时相比没有优先级调度的情况下降低了2.14 s,由此带来0.4的 QoE 提升,其代价是网络状况较差的 TCP 连接的请求的 99th QoE 下降了 0.01。而在请求响应的中位数(50th)上,网络状况较好的 TCP 连接的响应延迟在有优先级调度时降低了 1.05 s,由此带来 0.16 的 QoE 提升,而网络延迟较差的请求的QoE会对应下降 0.06。由此可以得出结论,在 Web 服务场景中,当使用应用定义的传输层优先级识别根据网络状况为请求进行优先级调度时,可以显著地提升网络较好的用户的服务体验,并且不会对网络较差的用户的服务体验产生较大影响。

表 1 优先级对不同网络状况请求用户体验质量影响

请求连接类型	99th 响应延迟/s	99th QoE	50th 响应延迟/s	50th QoE
优-带高优先级	2.08	0.96	1.16	0.98
差-带低优先级	6.25	0.38	4.36	0.54
优-无优先级	4.22	0.56	2.21	0.82
差-无优先级	6.02	0.39	3.61	0.60

5 结 论

面对数据中心混杂的请求负载,本文分析了对延迟敏感型请求响应尾延迟产生干扰的 2 种主要来源,即大量非延迟敏感请求产生的排队延迟及复杂请求产生的阻塞延迟。针对这 2 种混杂请求带来的干扰,本文提出使用用户态协议栈对数据中心请求进行处理,并在用户态协议栈上根据服务等级目标对请求采取灵活优先级调度是保障延迟敏感型请求的有效方法。本文提出通过支持上层应用向用户态协议栈驱动层与传输层注册应用定义的优先级识别回调函数,利用数据包、TCP 流等丰富的状态信息,根据负载特征与应用场景需求,在不修改网络协议栈的情况下,对多种数据中心应用及负载场景实现灵活的应用定义优先级识别与优先级调度。实验表明,该设计可以有效避免混合负载请求对延迟敏感型请求响应尾延迟的干扰,保证用户体验。但是,应用定义优先级调度的用户态协议栈还存在进一步改进的空间。例如,各个层次的优先级识别与调度间可以实现更紧密的动态协同,应用定义的优先级调度需要有更灵活、更公平的策略与调度语义支持,需要考虑多种优先级调度策略组合以及需要有更多的优先级等级以适应多场景组合时更复杂的优先关系等。

参考文献

[1] CHEN Y, WANG J, LU Y, et al. Fangorn: adaptive execution framework for heterogeneous workloads on shared clusters [J]. Proceedings of the VLDB Endowment, 2021, 14(12): 2972-2985.

[2] ZHANG W L, LIU K, SHEN Y F, et al. Labeled network stack: a high-concurrency and low-tail latency cloud server framework for massive IoT devices [J]. Journal of Computer Science and Technology, 2020, 35(1): 179-193.

[3] ZHANG X, SEN S, KURNIAWAN D, et al. E2E: em-

bracing user heterogeneity to improve quality of experience on the web[C]// Proceedings of the ACM Special Interest Group on Data Communication. Beijing: Association for Computing Machinery, 2019:289-302.
[4] Linux Foundation. eBPF [EB/OL]. (2021-01-01) [2022-03-28]. https://ebpf.io/.
[5] JEONG E Y, WOOD S, JAMSHED M, et al. mTCP: a highly scalable user-level TCP stack for multicore systems [C]// The 11th USENIX Symposium on Networked Systems Design and Implementation. Seattle: USENIX Associations, 2014:489-502.
[6] BELAY A, PREKAS G, KLIMOVIC A, et al. IX: a protected dataplane operating system for high throughput and low latency[C]// The 11th USENIX Symposium on Operating Systems Design and Implementation. Broomfield: USENIX Associations, 2014:49-65.
[7] MARTY M, DE KRUIJF M, ADRIAENS J, et al. Snap: a microkernel approach to host networking[C]// Proceedings of the 27th ACM Symposium on Operating Systems Principles. Huntsville: Association for Computing Machinery, 2019:399-413.
[8] PREKAS G, KOGIAS M, BUGNION E. Zygos: achieving low tail latency for microsecond-scale networked tasks[C] // Proceedings of the 26th ACM Symposium on Operating Systems Principles. Shanghai: Association for Computing Machinery, 2017:325-341.
[9] KAUFMANN A, STAMLER T, PETER S, et al. TAS: TCP acceleration as an OS service[C]// Proceedings of the 14th European Conference on Computer System. Dresden: EuroSys, 2019:1-16.
[10] KAFFES K, CHONG T, HUMPHRIES J T, et al. Shinjuku: Preemptive scheduling for μ second-scale tail latency[C]// The 16th USENIX Symposium on Networked Systems Design and Implementation. Boston: USENIX Associations, 2019:345-360.
[11] DEMOULIN H M, FRIED J, PEDISICH I, et al. When idling is ideal: optimizing tail-latency for heavy-tailed datacenter workloads with Perséphone[C]// Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. Virtual: Association for Computing Machinery, 2021:621-637.
[12] DEAN J, BARROSOL A. The tail at scale[J]. Communications of the ACM, 2013,56(2):74-80.
[13] BLUMOFE R D, LEISERSONC E. Scheduling multithreaded computations by work stealing[J]. Journal of the ACM, 1999,46(5):720-748.
[14] Intel. Datadirect I/O technology [EB/OL]. (2016-04-06) [2022-03-28]. https:// www. intel. cn/content/www/cn/zh/io/data-direct-i-o-technology. html.
[15] Wiki. OSI model [EB/OL]. (2022-03-12) [2022-03-28]. https:// simple. wikipedia. org/wiki/OSI _ model.
[16] Linux Man Page Project. epoll - I/O event notification facility[EB/OL]. (2019-09-01) [2022-03-28] . http://man7. org/linux/man-pages/man7/epoll. 7. html.
[17] Gstrauss. lighttpd. [EB/OL]. (2022-01-19) (2022-03-28). http://www. lighttpd. net/.
[18] WU W, FENG X, ZHANG W, et al. MCC: a predictable and scalable massive client load generator[C]// International Symposium on Benchmarking, Measuring and Optimization. Qingdao: Springer, Cham, 2019:319-331.

User-space network stack with application-defined priority scheduling

SHEN Yifan[*,**], ZHANG Wenli[*], LIU Ke[*], CHEN Mingyu[*,**]
([*] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)
([**] University of Chinese Academy of Sciences, Beijing 100049)

Abstract

The mixed requests in the data center introduce the problem of the interference to the tail response latency of latency-sensitive requests response. However, existing researches cannot provide flexible priority scheduling for latency-sensitive requests under different load scenarios. A user-space network stack that applies application-defined priority scheduling is proposed. This design takes the advantage of the key position of the user-space network stack in the data center request processing, supports upper-layer applications to flexibly customize priority identification and scheduling strategies according to load characteristics, and provides rich status information such as packets and TCP flows for priority recognition, so as to realize flexible application-defined priority identification and scheduling for different load scenarios without changing the network stack. This work prevents latency-sensitive requests from being disturbed by queuing and blocking delays caused by the other promiscuous request load. Experimental results show that under different load scenarios, flexible and accurate application-defined priority scheduling can reduce the response tail latency of latency-sensitive requests by 98.5%, which effectively ensures the user experience.

Key words: datacenter network, priority, scheduling, user-space network stack