分　数：	
评卷人：	

華中科技大學

研究生（数据中心技术）课程论文（报告）

题 目：操作系统中的调度

学　　号 M202173715

姓　　名 崔雁翔

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

2022 年 1 月 10 日

目录

操作系统中的调度综述

崔雁翔

摘　要　操作系统调度的目的是在有限的资源下，通过对多个程序执行过程的管理，尽可能满足系统和应用的指标，例如应用的等待响应时间、完成时间，系统的资源利用率、吐率、功耗等；这些指标根据应用场景的不同也会相应变化。然而，在操作系统中做好调度绝非易事，系统中没有全知全能的 “先知”来指导调度，复杂多变的应用场景和不同程序的需求特征都会增加调度的难度。因此，针对调度的研究伴随着整个计算机系统的发展历程，调度的设计与实现随着时代变迁也在不断演化。

关键词　操作系统；调度；公平性

A Survey on Scheduling in Operating System

Cui YanXiang

Abstract　The purpose of operating system scheduling is to manage the execution process of multiple programs to meet the system and application indicators as much as possible under limited resources, such as application waiting time, completion time, system resource utilization, spit rate, Power consumption, etc.; these indicators will also change according to different application scenarios. However, it is not easy to do a good job of scheduling in the operating system. There is no omniscient "prophet" in the system to guide the scheduling. The complex and changeable application scenarios and the requirements of different programs will increase the difficulty of scheduling. Therefore, the research on scheduling is accompanied by the development of the entire computer system, and the design and implementation of scheduling are also evolving with the changes of the times.

Key words　**operating system; scheduling; fairness**

1. 研究背景

1.1 调度介绍

一般来说，一个系统会同时处理多个请求，但是其资源是有限的. 调度就是用来协调每个请求对资源的使用的方法。调度不仅仅是计算机中的概念。在柜台办理业务的时候，人们默认会按照先到先得的原则排队，如果将柜台理解为资源，每个人都对应着一个请求，那么排队就是让请求通过先到先得的方式来使用资源，这是一种直观的调度策略。在业务办理窗口经常能够看到 “VIP 用户优先”的标识，这就体现了调度中优先级的概念，会根据重要程度给每个请求分配优先级，优先级高的请求可以优先使用资源。

CPU、内存、磁盘这些系统中常见的资源毫无疑问都需要系统中的调度进行管理。操作系统中的内存管理同样是一种调度，内存管理将虚我内存映射到物理内存，这里的物理内存就是有限的资源。当需要使用的虚拟内存容量超过物理内存的实际容量时，换页机制就是在对哪部分物理页的内容可以留在内存中进行调度，并将剩余物理页上的内容写回磁盘，进而复用那此剩余物理页。另外，调度不仅仅体现在操作系统内核中。内核中的调度器无法感知用户态程序的具体语义，因而无法做出最优的调度决策。同时，应用程序对性能的需求不断提高，这促使越来越多用户态的程序也开始实现自己的调度器。调度不但用在计算机系统的方方面面，也体现在我们工作、生活、学习的点点滴滴。

1.2 调度指标

计算机的应用场景复杂多变，用户对于不同的场景会有不同的预期，因此会有不同的调度指标来指导调度策略的选择。常用的调度指标包括：与性能相关的吞吐量、周转时间、响应时间；一些非性能指标，例如公平性、资源利用率；某些任务、场景特有的需求，例如实时任务的实时性、终端设备的能耗。

吞吐量：吞吐量是单位时间内处理的任务数量，在计算机处理的任务中，有一类任务被称为批处理任务，比如机器学习的训练、复杂的原理科学计算。这类任务在计算机上执行时无需与用户交互，其目标就是尽快完成，主要的调度指标是任务处理的吞吐量，需要让吞吐量尽可能高。

响应时间：响应时间是指任务从被发起直至第一次向用户返回输出以响应用户所需的时间。随着计算机的应用场景越来越多，人们开始需要使用计算机执行一些交互式任务，例如程序调试。对于这类交互式任务来说，它们需要在执行过程中对用户操作进行响应。用户关心的是自己的请求（例如自己敲击键盘的输入）能否及时被处理。调度应该确保交互式任务的响应时间足够短，进而使用户获得良好的体验。

实时性：操作系统还会被用于处理有截止时间要求的实时任务。例如车载系统中的距离探测器会定时检测汽车与外部物体的距离，如果当前车速过快且距离过近，则车载系统会强制刹车。又比如视频的画面渲染，每一帧画面需要在截止时间前完成，否则会造成视频卡顿。在系统保证实时任务执行结果正确的同时，调度还必须让实时任务在截止时间前完成，即满足实时性。

能耗：当下，移动互联设备已经走进每家每户，手机已经成为人们日常生活中不可缺少的一部分。手机等移动设备有一项指标是待机时间，待机时间长的设备可以满足人们在外长时用使用的需求，概括地说待机时间对应于系统的能耗指标。因此，移动设备上的操作系统调度还需要尽可能降低能耗。

同时，还有一些调度指标是所有场景共有的。调度器应该尽可能地保证系统资源被充分利用，提高资源利用率；系统中执行的任务都有其必要性，在通常情况下，应保证每个任务都有执行的可能，即满足公平性；最后，系统使用调度器的原因是希望优化任务的执行，而非引入新的性能瓶颈，所以调度器做出决策的时延应尽可能短，降低调度开销。

1.3 调度中的权衡

调度开销与调度效果：调度器要做出一个合理的调度决策，就需要统筹足够多的信息，进行足够多的计算，但这会导致其决策的时间变长。因此，调度器的开销和调度后的任务执行效果之间存在权衡。

优先级与公平性：系统中的任务通常会被分配优先级，一方面要保证高优先级的任务优先执行，另一方面又不能让低优先级的任务执行的时间过短，甚至无法执行。这其中也存在着任务优先级和任务公平性的权衡。

性能与能耗：在手机等移动设备中，能耗是比较重要的调度指标。如果调度器过分追求性能，总是使 CPU 维持高速运转，就可能导致单位时间能耗过高，手机电量迅速耗尽，从而影响用户体验。因此，调度器还需要考虑性能和能耗之间的权衡。

1.4 经典调度介绍

这里介绍一些经典而直观的调度策略，虽然这些策略看上去

先来先服务调度算法：先来先服务(FCFS)调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用 FCFS 算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。

短作业(进程)优先调度算法：短作业(进程)优先调度算法 SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。

高优先权优先调度算法：为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程。

时间片轮转法：在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把 CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几 ms 到几百 ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

2. 相关研究

2.1 Linux Scheduler

Linux 作为世界上最受欢迎的操作系统之一，截至目前

其调度器已经经历了很多版本，这部分内容简单介绍一下Linux 中使用过的一些调度器

2.1.1 O(n)调度器

在 Linux 2.4 版本以前,Linux 调度器的设计实现较为简单，是一个基于 RR 策略运行队列,在 2.4 版本使用的就是一个 O（n）调度器，顾名思义，O（n）调度器是指调度决策时间复杂度为 O（n），n 是指调度器中运行队列的数量，这部分内容主要介绍一下 O（n）调度器的思想以及其缺陷。

O（n）调度器采用了一个负载分担的思想，所有的任务被存储在一个全局运行队列中。被选择调度的任务会从运行队列中移除，当该任务执行完成并且需要再次被调度时，会被重新放入运行队列的队尾。当调度器选择下一个被调度的任务是，它需要遍历运行队列中所有任务，并且重新计算它们的动态优先级，然后选取动态优先级最高的任务。

在早期 Linux 中一个系统中的任务量还很少，O（n）的复杂度仍能保证系统正常工作，但是随着计算机系统与硬件的不断发展，一个系统中运行的任务越来越多，O（n）调度器的问题也逐渐出现，主要有以下两点：

（1） 调度开销过大。O（n）时间复杂度会导致任务过多的情况下调度器的决策时间长，浪费 CPU 资源。同时在所有任务执行完一个时间片后，O（n）调度器需要更新它们的时间片，这也会造成额外开销。

（2） 多核扩展性差。O（n）调度器使用了锁来保护全局运行队列的并发访问，在多核场景下可能会出现扩展性问题，某 CPU 核心对全局运行队列的遍历和修改操作会导致其他 CPU 核心缓存中的对应信息被清空。

2.1.2 O(1)调度器

O（1）调度器是为了解决 O（n）调度器存在的问题而出现的，O（1）调度器使用了两级调度思想，每个 CPU 核心单独维护一个本地运行队列，让任务仅在同一个核心上调度，每个本地运行队列实际上是由两个多级队列：激活队列和过期队列组成，分别用于管理仍有时间片剩余的任务和时间片耗尽的任务。当一个任务的时间片耗尽时，它就会被加入过期队列。如果当前队列中没有可调度的任务，O（1）调度器会将两个队列的角色互换，开始新一轮的调度。在每一个多级队列中都维护了一个位图，位图中的比特位用于判断对应的优先级队列是否有任务等待调度。在制定决策时 O（1）调度器会根据位图找到激活队列中第一个不为空的队列，并调度该队列的第一个任务，时间复杂度为 O（1），运行队列中的任务数量无关。

O（1）调度器的时间开销很小，在大部分场景下都能获得不错的性能，但是 O（1）调度器仍存在一些弊端。

交互式任务的判定算法过于复杂。O（1）调度器为了保证交互式任务被优先调度，采用了大量启发式方法来判定一个任务是否为交互式任务，大部分场景下这些启发式算法可以做到正确判断交互式任务，但是在一些特定场景下可能会失效。另外这些启发式算法的代码中硬编码了大量参数，代码的维护性很差。

2.1.3 完全公平调度器（Completely Fair Schedule）

为了解决 O（1）调度器的问题，Linux 在 2.6.23 版本开始使用完全公平调度器代替了 O（1）调度器。O（1）调度器需要频繁的使用启发式算法来判定交互式任务，再给予交互式任务更多的执行的机会。而 CFS 调度器只关心非实时任务对 CPU 时间的公平共享，避免了复杂的调度算法实现与调参，同时 CFS 调度器可以动态地设定任务时间片，确保任务的调度时延不会太高。

CFS 调度器具有动态时间片：为了避免静态时间片带来的问题 CFS 调度器使用了调度周期的概念，保证没经过一个调度周期，运行队列中所有时间片所有任务都会被调度一次。因而最坏情况下，任务的调度时延即为一个调度周期。同时间片一样，调度周期会带来权衡问题。如果调度周期过长，则一系列任务必须在很长时间的运行后才能体现公平性，且任务的调度时延可能过长；如果调度周期过短，则调度开销会变大。

CFS 调度器使用红黑树作为运行队列。CFS 没有使用以 vruntime 从小到大排列的队列作为运行队列，而是使用了红黑树。红黑树是一种平衡二叉查找树。根据索引键能以 O(logN) 的时间复杂度在红黑树中插人一个节点，其中 N 为红黑树的键值数量。同时红黑树中存储的键值是有序的，根据索引键排列。因此红黑树可通过维护最小的索引键，来 O(1)地查找最小索引键以及对应的键值。

2.1.4 总结

在 Linux 上调度器的发展也不是一成不变的，每当其存在一些缺陷时，Linux 的开发者们便会想办法去解决这些缺陷，随着全世界的开发者们的努力才有了现在的这个 Linux 的 CFS 调度器。

2.2 Fair Scheduling for AVX2 and AVX-512 Workloads

2.2.1 介绍

本文来自 ACT 2，作者针对程序执行 AVX 指令集中相关的指令会引起 CPU 降频的行为提出了一种针对 AVX 工

作负载的公平调度方法。当执行耗电指令时，功率受限的 CPU 可能不得不暂时降低其频率以防止过度耗电。通常，这种频率降低也会影响不执行此类耗电指令的任务。最近支持 AVX2 和 AVX-512 的英特尔 CPU 表现出这种行为，在这项工作中，作者描述了一种技术来识别性能受其他 AVX2/AVX-512 任务影响的受害者任务。作者描述了对 CPU 时间计算的修改，其中根据任务所经历的频率降低来调整分配给这些牺牲任务的 CPU 时间。该更改导致基于 CPU 时间的公平调度程序将任务的优先级设置为它们再次获得公平的 CPU 性能份额的程度。作者的原型能够将涉及 AVX2 应用程序的工作负载的不公平性从平均 7.9% 降低到 2.5%，将 AVX-512 的不公平性从 24.9% 降低到 5.4%。

2.2.2 背景与目标

本文来自 ACT 2，作者针对程序执行 AVX 指令集中相关的指令会引起 CPU 降频的行为提出了一种针对 AVX 工作负载的公平调度方法。当执行耗电指令时，功率受限的 CPU 可能不得不暂时降低其频率以防止过度耗电。通常，这种频率降低也会影响不执行此类耗电指令的任务。最近支持 AVX2 和 AVX-512 的英特尔 CPU 表现出这种行为，在这项工作中，作者描述了一种技术来识别性能受其他 AVX2/AVX-512 任务影响的受害者任务。作者描述了对 CPU 时间计算的修改，其中根据任务所经历的频率降低来调整分配给这些牺牲任务的 CPU 时间。该更改导致基于 CPU 时间的公平调度程序将任务的优先级设置为它们再次获得公平的 CPU 性能份额的程度。作者的原型能够将涉及 AVX2 应用程序的工作负载的不公平性从平均 7.9% 降低到 2.5%，将 AVX-512 的不公平性从 24.9% 降低到 5.4%。

2.2.3 设计与实现

受害者任务识别：

作者根据寄存器访问来检测高耗电代码——许多复杂的指令集扩展引入了额外的架构寄存器，因此对这些寄存器的访问标志执行了这种高耗电代码。该方法使用基于陷入异常的机制来识别 AVX-512 代码：当操作系统清除 XCR0 寄存器中的 ZMM_Hi256 和 Hi16_ZMM 位时，512-的上下文切换位向量寄存器被禁用和 AVX-512 指令触发未定义指令异常。

具体做法是在一个时间片内捕获第一个 256 位或 512 位寄存器访问以检测任务是否使用 AVX2/AVX-512。在每次上下文切换时，测试下一个任务是否具有有效的 256 位或 512 位寄存器内容，如果有，则将该任务标记为 AVX2/AVX-512 任务。如果没有有效的 512 位寄存器状态，将阻止进一步的 512 位寄存器访问，如果也没有有效的 256 位寄存器状态，则会清除 XCR0 寄存器中的 AVX 位，以使未来的 AVX2 指令触发异常。如果在调用未定义指令异常处理程序时未启用 256 位寄存器，只需重新启用这些寄存器，将当前任务标记为 AVX2 任务并继续执行。类似地，如果 256 位寄存器已启用但 512 位寄存器尚未启用，将启用 512 位寄存器并将当前任务标记为 AVX-512 任务。在下一次调度器调用期间，可以测试之前的任务是否被标记为 AVX2 或 AVX-512 任务，以确定是否应用降频补偿。

性能影响计算：

对于那些收到降频影响的任务需要给予补偿，以让它们可以获得相等的 CPU 性能。一种简单的解决方案是让这些受害者任务获得更多 CPU 时间，具体的做法是根据实际运行时 CPU 的频率与单独执行时的频率之比缩放 CPU 时间。

2.2.4 实验与评估

作者的设计实现基于 MuQSS 调度器，是在 CFS 调度器的基础上修改得到的，其调度策略仍然使用 CFS 的调度策略。作为评估的基准，作者使用 nginx Web 服务器、Parsec 3.0 基准套件中的大多数基准以及 Phoronix 测试套件 9.0.1 中的 Linux 内核构建基准（以下称为“内核构建”）。这些基准测试作为降频补偿的潜在受害者任务。在作者的实验中，负责 AVX 频率降低的后台应用程序是 x265 视频编码器，因为它支持 AVX-512。MuQSS 可以配置为在多个逻辑 CPU 之间共享运行队列，作者选择了超线程兄弟之间的运行队列共享。

公平性：

图 4 显示了公平性实验的结果。可以立即看出，作者的原型大大降低了两个应用程序之间的不公平性。 虽然基线系统显示 AVX2 的平均不公平率为 7.9%，AVX-512 的平均不公平率为 24.9%，但作者的原型分别将这些数字降低到 2.5% 和 5.4%。

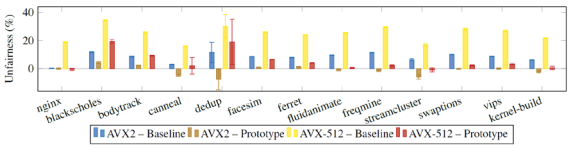


Figure 4: Unfairness for applications executed alongside x265 – for most workloads, our scheduler greatly reduces the unfairness between AVX2/AVX-512 applications and other applications executed in parallel. For AVX2, the unfairness was reduced from 7.9% (blue bars) to 2.5% (brown bars) on average, and for AVX-512 from 24.9% (yellow bars) to 5.4% (red bars).

图 1

开销：

虽然降频补偿可以降低 AVX2/AVX-512 任务对其他

任务的性能影响，但对操作系统的必要修改也会造成开销。 特别是，修改后的调度程序算法每秒在每个内核上执行多次。 额外的代码不仅增加了调度器本身所花费的时间，而且增加了调度器的缓存占用空间。因此检测修改后的调度器产生的额外开销是必要的，从下图可以看到，对于大多数应用场景，作者的调度器相较于 CFS 而言，几乎没有太多的额外开销。

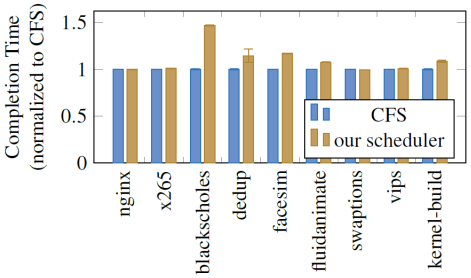


Figure 6: Our scheduler mostly provides competitive performance compared to CFS.

图 2

2.3 ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling

2.3.1 介绍

这是一篇来自谷歌的论文。本文提出 ghOSt，让 kernel 把做调度决定的能力托管给 userspace，满足谷歌数据中心上 workloads 和 platforms 快速演化的需求。改善调度可以大大提高关键负载的吞吐、时延、可扩展性以及安全性。

研究表明，在数据面操作系统（Data Plane OS）中使用可定制化的调度策略能在数据中心实现可观的性能收益。然而这些收益难以达到，因为不可能指望同一台机器上都跑都同一类调度需求的应用，尤其是在多租户环境中。

ghOSt 为给 Linux userspace 提供了通用的调度策略托管。ghOSt 提供状态封装、通信和 action 机制，允许在用户态代理中复杂表达调度策略，以及同步协助。开发人员允许使用任何语言来开发和优化策略。ghOSt 支持多种调度模型（per-cpu、run-to-completion 和抢占式），且调度执行的开销较低。作者们用 ghOSt 优化 Google Snap 和 Google Search，可以达到与内核调度器相媲美的 throughput 和 latency，并允许策略优化、不间断升级和错误隔离。

2.3.2 研究背景

现在云场景下 workloads 种类越来越多，场景需求越来越多，所以原本一个总体的调度策略又要兼顾大部分 workloads 的性能，又要满足部分 workloads 的特殊需求，这是非常困难的一件事情。并且就算开发成功，每次升级调度策略也都要重启整个 host，带来了很大的开销和应用 down-time。

之前那些想要通过设计用户态解决方案来提高性能和降低 kernel 复杂度的工作都有着很大短板：要么对应用的实现进行实质的修改，要么需要专门的资源才能快速响应，或是需要对特定的 kernel 和版本进行修改。

于是本文提出 ghOSt 系统的主要目标：

1. 调度策略应当容易实现和测试；
2. 调度需要有效、广泛；
3. 突破 per-CPU 调度模型的限制；
4. 支持同时多个调度策略；
5. 升级不需要重启 host 和错误隔离。

2.3.3 设计与实现

如图 3 所示。userspace 的 agent 做出调度决定并且指导 kernel 如何在 CPU 上调度 native 线程。ghOSt 的 kernel 部分被实现为一个调度 class，就类似常用的 CFS class。这个 scheduling class 向 userspace 提供丰富的 API 来定义任意调度策略。为了帮助 agents 做出调度决定，kernel 向 agent 导出线程的状态，通过 messages 和 status words。agents 然后通过 transactions 和 system calls 向 kernel 传递数据。

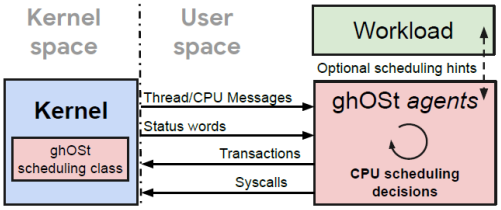


Figure 1. Overview of ghOSt.

图 3

整个系统的调度架构如下图 2 所示，ghOSt 可以设置起多个 enclave，每个 enclave 中可以运行不同的调度策略。ghOSt 内核模块会将调度的决定逻辑托管给运行在每个物理 CPU 上的 ghOSt Agent。

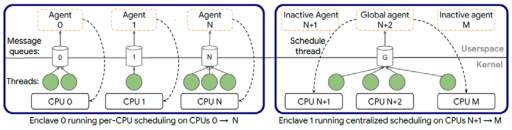


Figure 2. ghOSt policies manage CPUs assigned to enclaves. Each enclave can be managed by a different ghOSt policy.

图 4

如果该 enclave 采用 per-CPU 的调度策略，则 ghOSt 内核模块会向各个 cpu 上的 ghOSt Agent 分别提供必要的线程信息。

如果该 enclave 采用 Centralized 的调度策略，则该 enclave 中仅有一个 ghOSt Agent 会生效，所有的调度信息也仅会提供给它。

2.3.4 实验与评估

ghOst 开销

ghOSt 的产生的额外开销如下图

1. Message Delivery to Local Agent	725 ns
2. Message Delivery to Global Agent	265 ns
3. Local Schedule (1 txn)	888 ns
Remote Schedule (1 txn for 1 CPU)	
4. Agent Overhead	668 ns
5. Target CPU Overhead	1064 ns
6. End-to-End Latency	1772 ns
Group Remote Schedule (10 txns for 10 CPUs)	
7. Agent Overhead	3964 ns
8. Target CPU Overhead	1821 ns
9. End-to-End Latency	5688 ns
10. Syscall Overhead	72 ns
11. pthread Minimal Context Switch Overhead	410 ns
12. CFS Context Switch Overhead	599 ns

Table 3. ghOSt microbenchmarks. End-to-end latency is not equal to the sum of agent and target overheads as the two sides do some work in parallel and the IPI propagates through the system bus.

图 5

Centralized 场景下 global agent 的可扩展性：

根据 agent overhead 可以计算出在这个 intel 机器上如果用 global agent 占用一个核心来管理的话，可以大约每秒在 100 个核心上各管理 252000 个任务，每个任务每次被调度后运行 40us。

当然上面的只是理论上，本文实验了下 global agent，实验机器配置为 2*socket，28 core/socket， 2 logical core/core。

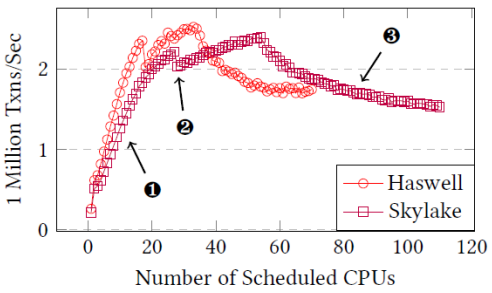


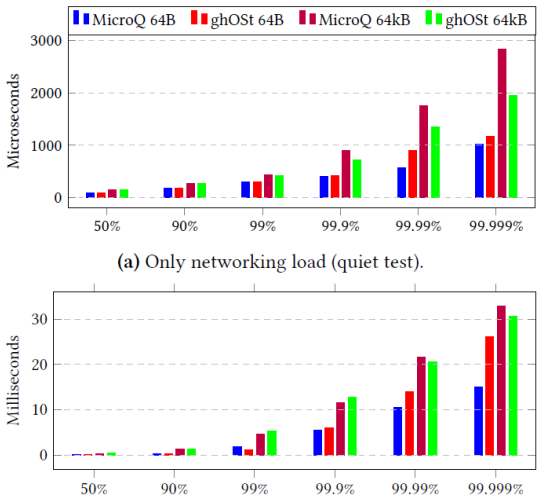
Figure 5. The scalability of a global agent.

图 6

第一段上升说明了越来越多 cpu 可以被用来调度。第二段下降了一下是因为此时已经是测试机单个 socket 的 28 核了打开超线程，允许 global agent 和别的 ghOSt 线程一起运行在同一个物理核上，所以 global agent 虽然受影响了，但是还能继续上升。第三阶段，等到了 56 cpu 左右，超线程也用完后开始跨 socket 进行调度，这时候受 NUMA 影响，agent 对 numa 节点上的 CPU 进行调度时需要内存操作核发送跨 numa sockets 的 interrupt，从而导致每次操作开销都更大，引起 agent 的吞吐量下降。

与谷歌内部的 MicroQuanto 对比：

google 把 ghOSt 和内部的 kernel scheduler MicroQuanta 进行对比，这个 MicroQuanta 是一个软实时的 kernel scheduler 用来管理 snap 框架，这个 snap 框架类似 dpdk，需要维护一些 polling 线程来和网卡交互或是实现一些定制化的网络和安全协议。snap 会根据网络负载来新增或减少 polling 的线程。



(a) Only networking load (quiet test).

(b) With additional load (loaded test).

Figure 7. Google Snap latencies for 64B and 64kB messages. busy due to a server process thread. MicroQuanta, on the other hand, has to wait for a blackout period.

图 6

测试结果如上图 6所示,在处理 64B 的网络包时 ghOSt 的时延会比 MicroQuanta 差 10%是因为小数据包的处理时间很短所以 ghOSt 进出 userspace 的开销就会变得明显。处理 64KB 大小的网络包时，ghOSt 的时延会更好则是因为 ghOSt 能跨 cpu 进行负载均衡。

在谷歌搜索业务场景下的测试：

在谷歌搜索的 workload 上（数据库查询 workload）进行测试，当前 google search 使用的是 CFS，实验环境下换成 ghOSt 后经过优化，可将 ghOSt 的吞吐量达到与原本 CFS 相媲美的程度，但是能够在 latency 上有 40-45%的提升，作者也将它归因于是 ghOSt 的 global agent 能够带来比传统 per-CPU 调度更好的负载均衡。

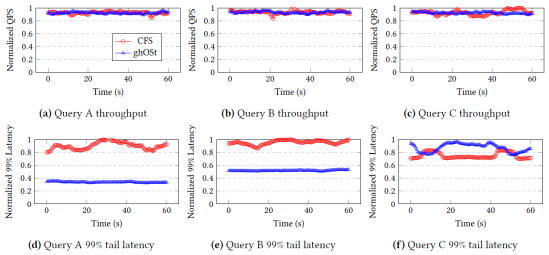


Figure 8. Google Search benchmark results with CFS and ghOSt scheduling.

图 7

3. 参考文献

[1] Mathias Gottschlag, Philipp Machauer, Yussuf Khalil, and Frank Bellosa. Fair Scheduling for AVX2 and AVX-512 Workloads[C]. 2021 USENIX Annual Technical Conference.2021

[2] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don,Luigi Rizzo, Oleg Rombakh, Paul Turner, Christos Kozyrakis, Google, Inc. Stanford UniversityLuo. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling[C]. SOSP '21, October 26–28, 2021, Virtual Event, Germany

[3] 陈海波，夏虞斌.现代操作系统原理与实现 [M].机械工业出版社,2021.