The Linux Scheduler: a Decade of Wasted Cores（进程调度）

一原文阅读

1作者与会议信息介绍

The Linux Scheduler: a Decade of Wasted Cores是由Jean-Pierre Lozi等人发表在Proceedings of the Linux Symposium这个专注于Linux内核以及相关技术的年度技术会议，与传统的顶级会议和期刊不同，该会议主要面向Linux开发者和爱好者，更多是服务于Linux社区，分享实际开发中的经验和技术实现，而不是严格的学术研究。本文通过发现和修复Linux调度器的关键性能问题，以及提供实用的工具，推动了Linux内核调度器在多核环境下的进一步优化，在学术界和工业界都具有较高的影响力。

2发现的问题与主要贡献

该论文主要研究的是进程调度相关的问题，具体说来，就是随着硬件性能的不断发展和架构的越发复杂，使得在单核中运行CFS算法的简单调度器，在多核，NUMA架构下，变得越来越复杂，甚至出现了一系列难以察觉的bug，使得有任务在队列中等待运行的同时，有的内核却处在空闲状态，没有即使处理这些任务。该论文最主要的贡献，就是在传统测试手段和调试工具无法检测问题的情况下，构建了新的工具Online Sanity Checker，确保在可用的内核上调度就绪的线程，还构建了调度器关键活动可视化的工具，展示系统中调度活动的关键数据，从而发现了4个关键的性能错误。

3四个性能错误的表现与解决

随着多核时代的来临，系统性能提升的同时也带来了一系列新的问题，比如NUMA架构下的非统一内存访问，不同的CPU有自己的本地内存，这就涉及到本地内存和远程内存速度差异的问题；再比如缓存一致性问题，当多个CPU贡献内存时，可能有着各自的缓存，这就涉及到了缓存间数据的同步问题；以及同步机制，协调多个线程或进程对共享资源的访问。以上种种问题，使得人们的极大注意力放在了解决上述问题当中，让调度器的实现变得十分的复杂，并且忽略了其最基本的功能——确保可运行线程使用空闲内核。论文中提到的4个问题，正是在这样的一个场景下发现的，包括The Group Imbalance bug，The Scheduling Group Construction bug，The Overload-on-Wakeup bug，The Missing Scheduling Domains bug。下面将重点介绍这4个性能问题的成因，解决方案，以及解决后性能的提升。

3.1The Group Imbalance bug

论文作者在多用户机器上，使用R机器学习包执行内核编译和数据分析时，遇到了这个组不平衡的问题。当时的服务器是64个，8个NUMA节点，运行的效率使得作者怀疑，没有充分利用所有的可用内核，而是所有任务挤在几个节点上，于是进行了一次实验：在机器上运行内核编译make（64个线程），2个R进程（各有一个线程），通过3个不同的ssh连接，即不同的终端，最后使用可视化工具进行分析，以下主要展示了各NUMA节点中运行队列的线程数量以及负载。

由下图可以观察到运行情况，在节点0和4中，几乎所有时间都只运行了一个线程，不难发现是两个R进程，其他的节点，几乎都是1个线程以上，即make的64线程被分摊到了其他6个节点当中，再来看b图，可以发现0和4上的负载远远大于其他节点上的。但存在着这样两个现实，首先一个NUMA节点上，是有8个核心的，R进程只有一个线程，也就是此时节点0和4中有7个核心是空闲状态，按道理调度此时应该把这些空闲核心利用起来，从其余忙碌的节点中分担一部分任务。第二个是就make和R进程而言，他们总体的负载是几乎一致的。

要解释下图的情况，就得了解一下调度器的负载平衡算法逻辑。原来调度器在执行负载平衡算法时，会计算各个调度组的平均负载，从而判断负载情况。在本例中，0和4上虽然有空闲的核心，但是由于R进程完全在0和4上运行，平均负载远大于其余节点，因此即使是有空闲核心，仍然认为是忙碌的，因此不会去分担其余节点上的任务。归根结底，就是平均负载未能如实反映各个调度组的实际运行情况，论文中提出了一个很简单的解决方法，那就是将比较平均负载改为比较最小负载，如果A调度组的最小负载小于B 的最小负载，则A要分担B任务，这样做既不改变复杂度，调度组间的迁移数也不会上升（统计经验表明）。 图c是引入最小负载后，各节点的表现，可以发现解决了各节点负载不均的问题 ，且make的完成时间减少了13%。

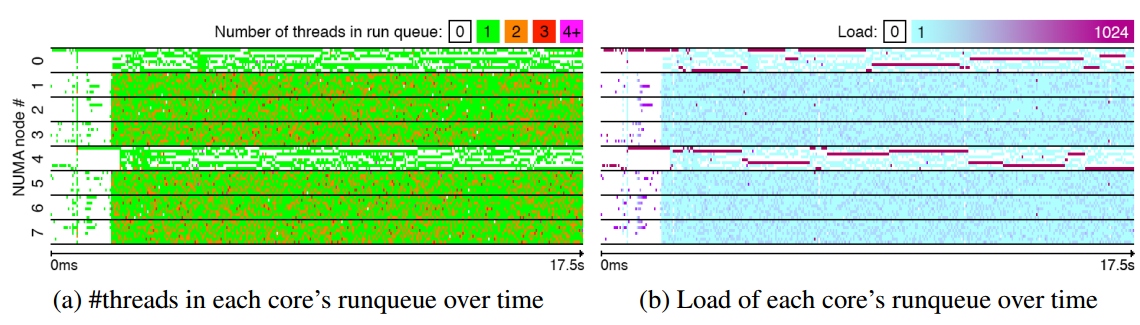


图1-1 组不平衡bug的具体表现

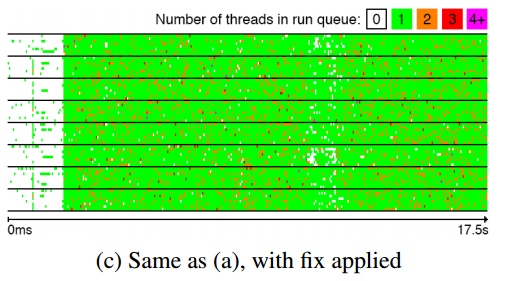


图1-1（续） 修复后的情况

3.2The Scheduling Group Construction bug

第二个错误是由于调度组构造产生的。Linux中定义了taskset的命令，它允许将应用程序固定在可用内核的一个子集上运行，也就是说，本意是想着当子集中某个核心负载严重时，任务可以切换到别的空闲的核心上，减轻负载。但由于调度组在构造过程中，是基于特定核心的，即固定视角的，以节点0为例，构建距离为1的调度组{0，1，2，4，6}，此时3不在其中，再以3构建距离为1的调度组{1，2，4，5，7}。若此时taskset指定节点1和2，这两个节点又恰好在两个调度组中，因此当1负载严重，2想要窃取任务时，由于1和2在两个调度组中，因此2在计算自己调度组的平均负载和1所在调度组的平均负载时，会发现负载几乎一致（尽管调度组间存在差异，但这种差异可能并不会让调度器去执行负载平衡算法），因此就不会从1中窃取任务。

这种现象在构造调度组是基于特定核心，以及taskset指定的两个节点距离恰好为2时，很大概率会出现。文中给出的办法是不在基于特定的核心，而是从每个核心的角度出发构造调度组，这样就能尽可能避免上述的情况。

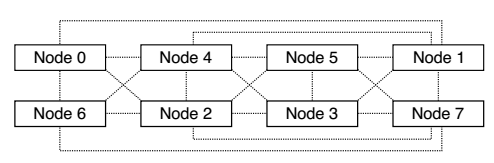


图1-2 8-node AMD Bulldozer machine的拓扑结构

这种调度组构建错误对于lu这种使用流水线算法来并行工作的任务影响尤其严重，优化前后的差异达到了27倍。

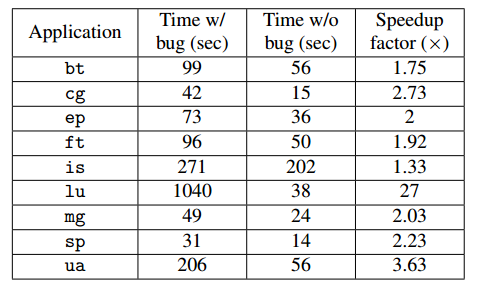


图1-3 有和没有调度组构建bug的性能差异

3.3The Overload-on-Wakeup bug

这个错误的主要表现是，任务a一开始在node1上运行，之后进入睡眠，当其再次被唤醒时，考虑到最大化缓存重用，a仍然会在node1上运行，但问题在于，node1可能处于忙碌状态，那么a就要继续等待node1，而忽略其余空闲的核心。下图是在过载的核心上唤醒线程的例子：蓝点是空闲核心，紫点是忙碌核心，方框代表在过载核心上唤醒线程，该现象频频发生，大大地浪费了空闲的核心。可以看到，一段时间后，负载又平衡了，这是因为调度器恰好运行周期性负载平衡算法或者紧急平衡算法，空闲的核心将任务窃取了。而出现了如此多的空隙，则是因为唤醒了一系列相关关联的任务，彼此间需要对方执行的结果，这就造成了相互等待，调度器需要频繁的阻塞唤醒，就造成了一系列的空隙。

文中提出的解决方法很简单，就是在唤醒时首先判断当前核心是否空闲，如果否，则寻找最长空闲时间的核心。值得一提的是，这里之所以找最长空闲时间的核心，是为了避免再次出现上述的情况。对于空闲核心，有短期空闲与长期空闲之分，前者可能每经过一小段时间就有线程唤醒休眠，在这样的核心上唤醒我们的线程，就再次陷入到之前的问题了，因此这里选择最长空闲时间的核心。如果没有空闲的核心，则在最初忙碌的核心中唤醒线程。

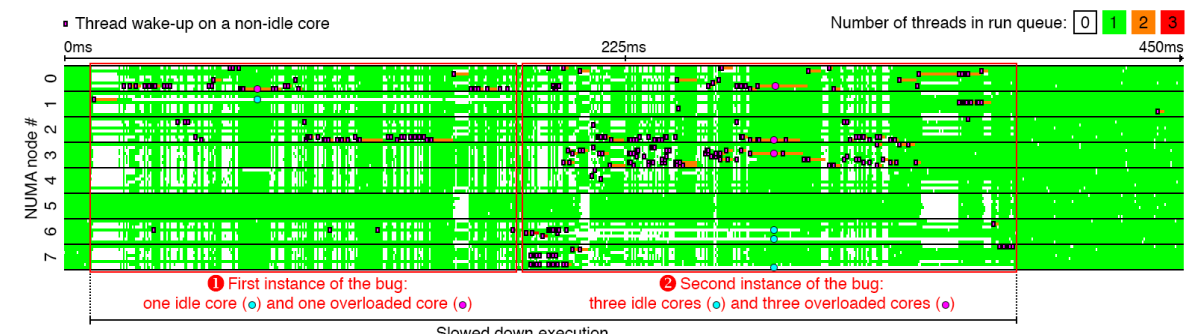


图1-4 在overload中唤醒的例子

如下是对商业数据库TPC-H进行测试后的效果对比。

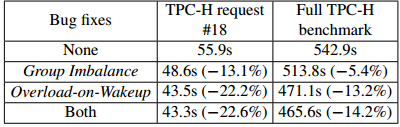


图1-5 TPC-H上修复bug的表现

3.4The Missing Scheduling Domains bug

这个错误的表现为当某个核心被禁用然后重新启动时，不正确地更新了表示机器中调度域数量的全局变量，，该变量被设置为NUMA节点内的域数量，也就是说，任何NUMA节点之间的负载平衡将不再执行，只会考虑节点内的平衡。再加上子线程的运行路径往往与父线程的一致，这就可能导致很多线程都运行在一个节点之中，如下图所示，节点0无法窃取1中运行的任务。

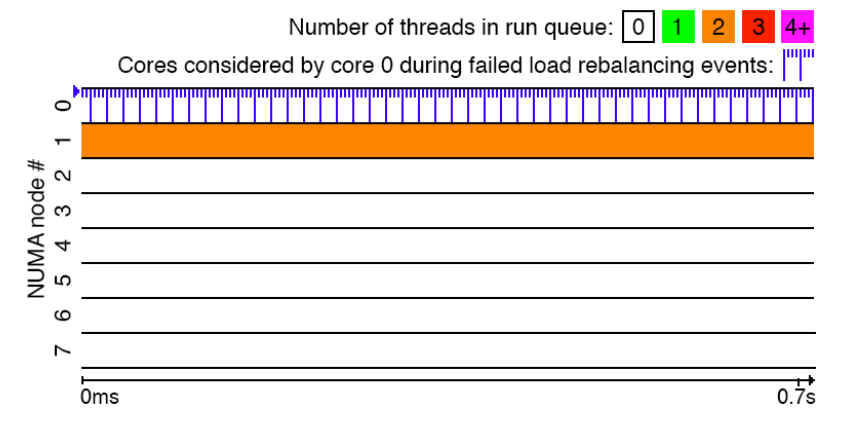


图1-6 禁用重启核心后，0无法窃取1的任务

这个bug的根本原因，发现它出现在重新生成调度域的代码中。每次禁用一个核心时，Linux都会重新生成调度域。这是一个两步的过程：首先生成单个NUMA节点内部的域，然后再生成跨NUMA节点的域。然而，在一次代码重构中，负责生成跨NUMA节点调度域的函数调用被Linux开发者不小心删除了。为了解决这个问题，我们重新添加了跨NUMA节点生成调度域的函数调用。通过将这一步骤加回到代码中，负载均衡机制得以恢复正常，解决了该bug。

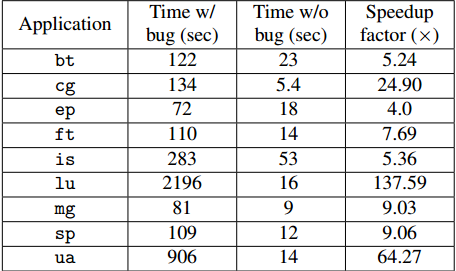


图1-7 NAS应用的运行时间（有和没有调度域消失错误）

4以往检测工具与本文提出的新工具对比

4.2节要介绍的工具Online Sanity Checker以及4.3节中的Scheduler Visualization Tool，移植性强，很容易在内核版本之间移植（从Linux 3.17到4.3），运行时的开销可以忽略不计，并且始终检测不变量违规。将它们放在标准工具带中可以帮助减少这类错误的未来发生。

4.1以往检测工具的不足

以往的工具想要检测这些漏洞是很困难的，它们不会导致系统崩溃或挂起，但会以标准性能监视工具难以察觉的方式消耗性能，这些事件造成的损害最严重的使查询速度降低23%。它们有的持续时间很多，只有几百毫秒，无法用htop、sar或perf等工具检测到，但在执行过程多次出现，久而久之对系统的正常运行产生严重的损害。即使在症状持续时间较长的情况下，也很难发现根本原因，因为它是调度器中许多异步事件的结果，难以跟踪检测。

4.2 Online Sanity Checker

Online Sanity Checker 是一个专门设计用于检测Linux调度器中潜在错误的工具，它能够有效地识别那些在短期内可接受但长期持续则不合理的状态。该工具不同于传统的断言（assert）或监控器（watchdog），它不仅能捕捉长期的系统违规行为，还能够避免对短期违规的误报。因为它不同于assert和看门狗等工具，它有一个检查间隔S，设置为1s，能覆盖大部分任务，当检查开始并发现异常时，进入短期监控状态，时长为100ms，这段时间内，工具会观察调度器的行为，判断违规状态是否能自动恢复。通过设置M值，工具能避免将短暂、瞬态的问题误认为是系统Bug。这就不同于assert发现错误后立马报错的表现。

在短期监控期间，Sanity Checker 会重点跟踪调度器中的关键事件，如线程迁移、线程创建和线程销毁等操作。这些操作是调度器修复系统状态的核心因素，因此通过监控这些事件，Sanity Checker可以准确判断违规状态是否在系统内部得到了修复。一旦检测到长期不变式违规，Sanity Checker会使用systemtap工具来记录负载均衡函数的调用过程、执行的语句以及相关变量的值。通过这些详细的剖析数据，研究人员可以了解系统在Bug触发时的行为，以及为什么调度器未能正常执行负载均衡。

4.3 Scheduler Visualization tool

Scheduler Visualization Tool 是一个用于可视化Linux调度器关键活动的工具，帮助研究人员理解Bug的症状并分析其根本原因。该工具通过图表展示系统中调度活动的关键数据，包括运行队列的大小（表示每个核心的就绪队列中等待执行的任务数量）、负载情况（表示每个核心的运行队列中的负载）以及在负载均衡和线程唤醒事件期间被考虑的核心（展示在这些调度事件中，调度器考虑了哪些核心）。这种工具对调度器的行为进行深入的剖析，并且它只有极低的开销。论文中的Figures 2, 4, 和 5 就是使用此工具生成的可视化图表，有助于定位调度失衡的根本原因。

二思考与感悟

在主流研究开发人员都在多核处理器和NUMA架构方面大做文章时，却忽略了调度器这一操作系统核心组件存在的问题。本文最大的价值不在于它一定程度解决了调度器未能充分利用所有核心，导致系统资源浪费的问题，而在于再次将人们的注意力，放到了操作系统的核心组件上来，上层哪怕堆的再高，若底层核心出现了问题，随时可能带来巨大的隐患。此外，它提出了一系列新的可视化工具，解决了传统检查工具所无法解决的问题，推动了可视化工具的发展。

阅读完The Linux Scheduler: a Decade of Wasted Cores这篇论文，让我对可视化工具以及调度器架构产生了一定的思考和感悟，我将围绕以往方法的优点和不足、新方法的局限性和不足之处、对UNIX/Linux操作系统今后发展趋势带来的影响，改善的可能方法或方向等方面展开。

首先是可视化工具方面。其中在Linux系统中，不乏各种各样的检测工具，他会爬取系统运行过程中，产生的各种各样的数据，比如systemtap、Pivot tracing等。但是，问题在于如何读懂这些数据，尤其是在用户不知道遇到了什么问题，又或者是不知道如何定位问题，该重点关注哪些变量。这时候，一个可视化的工具就很关键了，他可以以表格，热量图，饼图等各种各样的形式呈现数据，使得用户可以快速发现异常值，把握数据的变化趋势。文中作者发明的Online Sanity Checker和Scheduler Visualization Too，就很好的帮助用户定位问题所在。此类工具的发明也促使更多的人在可视化方面投入更大的精力。

近些年来，Linux的官方和社区其实在可视化方面已经下了不少功夫，如内核自带的性能分析工具Perf，第三方可视化工具和平台FlameGraph（火焰图），用于展示函数调用栈数据。它可以将系统采样数据（例如从Perf或者BPF收集的数据）转换为可交互的火焰图，帮助用户直观地看到性能瓶颈所在。但这些可视化工具都或多或少有一些问题，有的是UI不友好，需要命令行操作，有的是集成度差，需要下载多个插件结合使用，还有的安装部署复杂，需要对内核有一定的了解。上述种种因素，都限制着可视化的发展。

现如今，chatGPT非常火，我也经常在日常生活或者是科研学习的过程中使用。那么，在未来能不能在系统内嵌一个ai工具呢？以Linux为例，系统运行的各类数据，中间结果按类别整理，定期输入到ai工具中，当系统运行出现异常时，能不能由ai检测呢？又或者是让ai去分析追踪程序的复杂调用过程？其实，在其他领域，以医疗为例，就有那种异常检测的辅助工具。在未来，当硬件算力不断提升时，也许就有这种工具的诞生，我想，这或许是未来的一种优化的方向。

接着是调度器的架构。在单核系统中，调度器的优化常常是在最小化上下文切换开销，调度程度运行队列，调度量的长度等方面下功夫。然而，随着现代硬件架构（例如多核和NUMA架构）复杂性的增加，Linux调度器中加入了越来越多的优化以应对这些变化。这些优化虽然旨在提升性能，但也增加了调度器的复杂度，使得它变成了一个庞大而复杂的系统，并与电源管理、内存管理等其他内核模块紧密耦合。正是因为过多的优化尝试和调度器功能的不断扩展，导致了大量的Bug和复杂性。尽管在过去的几十年中，研究人员提出了多种改进调度算法的方法，但由于这些方法的集成难度较高，大多数并未被主流操作系统采用。

于是我在思考，能不能从调度器的架构入手。类似于Java中封装的思想，对于调度器的核心调度功能，设置核心模块，对于那些优化模块，按照一个统一的标准，定义成一个个优化子模块，当用户需要使用到某些优化时，只需导入对应的依赖。这种设置一方面将核心模块和优化模块分开，就避免了影响其最基本的调度功能，另一方面只需要根据需要，导入对应的依赖，这样就更加灵活，不会大幅度增加系统的复杂性。但是，尽管基于互不干扰的思想设计，但是基于同一问题，可能有不同的优化策略。怎么组合发挥最好的性能又是一个难题，并且优化可能也会带来意料之外的问题，因此，调度器的基本功能与硬件的不断发展仍然是一对难以调和的矛盾，希望在不久的将来，能有较好的解决方案。