天津科技大学本科生
毕业设计（论文）外文资料翻译

专　　业：物联网工程

姓　　名：王 炼

学　　号：**19036212**

指 导 教　师：梁 琨

2023 年 **3** 月 **20** 日

利用操作系统调度程序中的高频内核可提高应用程序的性能

摘　要

在现代服务器 CPU 中，各个内核可以以不同的频率运行，这允许对性能和能量比值权衡进行细粒度控制。然而，调整频率会导致高延迟。我们发现这会导致频率反转问题，即 Linux 调度程序将一个新的活动线程放置在一个空闲内核上，该内核需要数十到数百毫秒才能达到高频，就在另一个已经以高频运行的内核变为闲置的。 在本文中，我们首先通过在基于 80 核 Intel Xeon 的机器上编译 Linux 内核期间的调度程序行为的案例研究来说明重复频率反转的显着性能开销。在此之后，我们提出了两种策略来减少 Linux 调度程序中频率反转的可能性。当在 Intel Xeon 上对 60 多个不同的应用程序进行基准测试时，性能更好的策略 Smove 将 23 个应用程序的性能提高了 5% 以上（最多 56%，没有能源开销），并且性能降低了 5% 以上（最多 8%）仅适用于 3 个应用程序。在 4 核 AMD Ryzen 上，我们获得了高达 56% 的性能提升。

关键词： 完全公平调度（CFS）；处理器（CPU）；高性能核心；动态频率缩放（DFS）

1　引言

在性能和能耗之间取得平衡一直是计算系统开发过程中的一场较量。几十年来，CPU 一直支持动态频率调节(DFS)，允许硬件或软件在运行时更新 CPU 频率。降低 CPU 频率可以减少能源使用，但也可能会降低整体性能。尽管如此，对于经常闲置或不是很紧急的任务，降低性能的开销是可以接受的，因此在许多用例中通过降低频率来节省能源是可取的。虽然在第一台多核机器上，CPU 的所有内核都必须以相同的频率运行，但 IntelR©和 AMDR©最近的服务器 CPU 使更新单个内核的频率成为可能。此功能允许进行更细粒度的控制，但也带来了新的挑战。管理核心频率的挑战来源之一是频率转换延迟 (FTL)。事实上，将核心从低频过渡到高频，或者相反的过程，FTL 为几十到几百毫秒不等。 FTL 导致在进程创建时使用标准 POSIXfork()和 wait()系统调用的典型场景中的频率反转问题，或者是生产者-消费者应用程序中轻量级线程之间的同步问题。问题发生过程如下。首先，运行在核心 Cwaker 上的任务 Twaker 创建或解锁任务 Twoken。如果执行完全公平调度程序(CFS)，即 Linux 中的默认调度程序，找到一个空闲的核心 CCFS，它将把

Twoken 放在上面。此后不久，Twaker 终止或阻塞，因为它是一个父进程派生了一个子进程并在之后等待，或者因为它是一个线程已经完成生成数据并唤醒消费者线程，作为它在销毁之前的最后一个动作它将会睡眠一段时间。现在 Cwaker 处于空闲状态但执行频率很高，因为它直到最近才运行 Twaker，而运行 Twoken 的 CCFS 可能会以低频率执行，因为它之前处于空闲状态。因此，与内核负载相比，Cwaker 和 CCFS 的运行频率是相反的。在 Cwaker 达到低频而 CCFS 达到高频之前，即在 FTL 的持续时间内，这种频率反转不会得到解决。当前的硬件和软件 DFS 策略，包括最近添加到 CFS 的 schedutil 策略[9]无法防止频率反转，因为它们唯一的决定在于更新核心频率，因此每次都会为 FTL 付出很多代价。频率反转会降低性能并可能增加能源使用。在本文中，我们首先通过在具有 80 个内核（160 个硬件线程）的基于 IntelR© Xeon 的机器上构建 Linux 内核时 CFS 行为的案例研究，展示了真实场景中的频率反转问题。 .我们的案例研究发现，当通过 fork()和 wait()系统调用创建进程时，会出现重复的频率反转，并且我们的分析跟踪清楚地表明，频率反转导致任务在低频核心上运行以执行大部分任务。根据案例研究的结果，我们建议在调度程序级别解决频率反转问题。我们的主要观察是，调度程序可以通过在将任务放在核心上时考虑核心频率来避免频率反转。为此，我们提出并分析了两种策略。我们的第一个策略 Slocal 是让调度程序简单地将 Twoken 放在 Cwaker 上，因为频率反转涉及一个核心 Cwaker，该核心 Cwaker 可能处于高频状态，并且可能很快就会空闲。该策略提高了内核构建性能。然而，它存在风险，即 Twaker 不会立即终止或阻塞，导致在安排 Twoken 之前等待很长时间。因此，我们的第二个策略 Smove 在将 Twoken 放在 Cwaker 上时额外配备了一个高分辨率计时器，如果计时器在 Twoken 被调度之前到期，则将 Twoken 迁移到 CCFS，即最初为其选择的核心 CFS。此外，当 CCFS 高于最小频率时，即使通过将 Twoken 放在 Cwaker 上来稍微延迟它也是不值得的。因此，Smove 首先检查 CCFS 的频率是否高于最小值，如果是，则直接将 Twoken 放在 CCFS 上。本文的贡献如下。

（1）识别频率反转现象，这会导致一些空闲内核以高频率运行，而一些繁忙内核在很长一段时间内以低频率运行。

（2）案例研究，在 80 核服务器上构建 Linux 内核，每核频率独立。

（3）两种策略，Slocal 和 Smove，用于防止 CFS 中的频率反转。实施这些策略只需要对代码进行少量更改：在 Linux 内核中修改了 3 行（分别为 124 行）以实施 Slocal（分别为 Smove）。

（4）对我们在 60 种不同应用程序上的策略进行全面评估，包括流行的 Linux 基准测试以及来自 Phoronix [23] 和 NAS [5] 基准测试套件的应用程序。评估同时考虑了目前在 Linux 中默认使用的 powersave CPU 调控器和

实验性 schedutil 调控器。它还考虑了两台机器：一台大型 80 核 IntelR© Xeon E7-8870 v4 服务器和一台较小的 4 核 AMDR© Ryzen 5 3400G 台式机。通过服务器机器上的 powersave governor，我们发现 Slocal 和 Smove 的整体表现都很好：在评估中使用的 60 个应用程序中，Slocal 和 Smove 分别将 27 个和 23 个应用程序的性能提高了 5% 以上，并且将有 3 个应用程序的性能提高了 5% 以上。在最好的情况下，Slocal 和 Smove 将应用程序性能分别提高了 58% 和 56%，并且没有能源开销。然而，Slocal 在其中两个应用程序中的表现非常糟糕，在最坏的情况下甚至会降低 80% 的性能，这对于通用调度程序来说可能是无法接受的。在最坏的情况下，Smove 的表现要好得多：应用程序执行时间的增加仅为 8%，并通过能源使用方面 9% 的改善得到缓解。schedutil 的评估结果表明，该调控器没有解决频率反转问题，并展示了更多 Slocal 表现非常差的情况——而 Smove 却具有更好的在最坏情况下的表现。台式机上的评估也显示出类似的趋势，尽管规模较小但是 Smove 在边缘情况下的表现优于 Slocal。

2 案例研究：构建 Linux 内核

我们展示了一个导致我们发现频率反转现象的工作负载的案例研究：在具有 80 个内核的 4 插槽 IntelR© Xeon E7-8870 v4 机器上构建具有 320 个作业 (-j) 的 Linux 内核版本 5.4 (160 个硬件线程），标准频率为 2.1 GHz。得益于 IntelR© SpeedStep 和 Turbo Boost 技术，我们的 CPU 可以在 1.2 和 3.0 GHz 之间单独改变每个内核的频率。一个核心的两个硬件线程的频率是一样的。在本文的其余部分，为简单起见，我们使用术语“核心”来表示硬件线程。图 1 显示了内核构建工作负载运行时机器每个内核的频率。该图是使用我们开发的两个工具 SchedLog 和 SchedDisplay [10]生成的。 SchedLog 以非常低的开销收集应用程序的执行轨迹。 SchedDisplay 根据执行轨迹来生成图形视图。我们会在使用 SchedDisplay 生成中介绍的所有执行跟踪过程。SchedLog 记录显示的频率信息在图 1 中单位是一个滴答事件（CFS 中的 4 毫秒）。同时，在此类跟踪中没有彩色线意味着刻度已被 CFS 在该核心上禁用。 CFS 禁用非活动内核上的滴答以允许它们切换到低功耗状态。在图 1 中，我们注意到执行中的不同阶段。对于大约 2 秒的短时间，对于 4.5 到 18 秒之间的较长时间，以及大约 28 秒的短时间，内核构建具有高度并行的阶段，以高频率使用所有内核。这三个阶段中的第二个阶段对应于编译的大部分。在这三个阶段中，CPU 似乎得到了最大限度的利用。此外，在 22 到 31 秒之间，有一段很长的阶段，大部分是顺序代码，只有很少的活动核心，其中总是有一个以高频率运行。这个阶段的瓶颈是 CPU 的单核性能。然而，在 0 到 4.5 秒之间，以及在 18 到 22 秒之间，有些阶段会使用所有内核，但它们以 CPU 的最低频率 (1.2 GHz) 运行。仔细观察后，

这些阶段实际上主要是连续的：放大显示虽然在该阶段的持续时间内所有核心都被使用，但在任何给定时间只使用一个或两个核心。这就提出了两个问题：为什么要使用如此多的内核来执行近乎顺序的执行，以及为什么这些内核以如此低的频率运行。我们专注于核心利用率似乎次优的前几秒。放大 1 秒左右，我们首先查看运行队列大小和调度事件，如图 2(a) 所示。我们所处的模式是大多数顺序 shell 脚本的典型模式：进程是通过 fork() 和 exec() 系统调用创建的，并且通常一个接一个地执行。这些进程可以在图 2(a) 中轻松识别，因为它们以 WAKEUP_NEW 和 EXEC 调度程序事件开始。在 Core 56 上运行的进程在 0.96 s 左右的时间块后，三个这样的短生命周期进程在 Core 132、140 和 65 上一个接一个地执行。之后，两个运行时间较长的进程在 0.98 左右的 Core 69 上运行 s 标记，Core 152 在 0.98 s 和 1.00 s 标记之间。这种模式在图 2(a) 所示的整个执行过程中一直持续，任务在核心 148、125、49、52、129、156、60 和最后 145 上一个接一个地创建。执行过程的一部分，如图 2(b) 所示，给了我们一个关于为什么内核在这个阶段运行缓慢的提示：在任务开始在内核上运行的时间和结束运行的时间之间似乎有一个显着的延迟，尤其当核心频率开始增加时。例如，在 1.00 s 和 1.02 s 之间，Core 49 上的任务运行频率很低，只有在 1.04 s 左右结束时，核心频率才会上升到最大值，然后几乎立即开始再次下降，因为硬件注意到该核心上不再运行任何任务。同样的问题可以在 Core 152 的 1.00 秒之前观察到，而在 Core 69 的 0.98 秒左右。在最后一个例子中，当任务开始，即使在任务结束后频率仍持续下降，最终在 1.00 秒左右再次增加。似乎在执行的准备阶段，FTL 远高于任务的持续时间。由于彼此跟随的任务往往被安排在不同的核心上，因此它们很可能总是以低频率运行，因为大多数核心在执行的这个阶段的大部分时间都处于空闲状态。

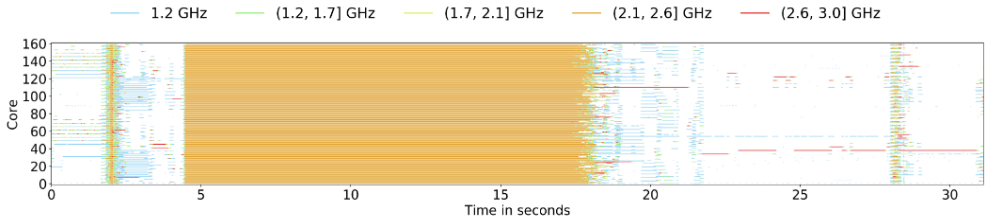
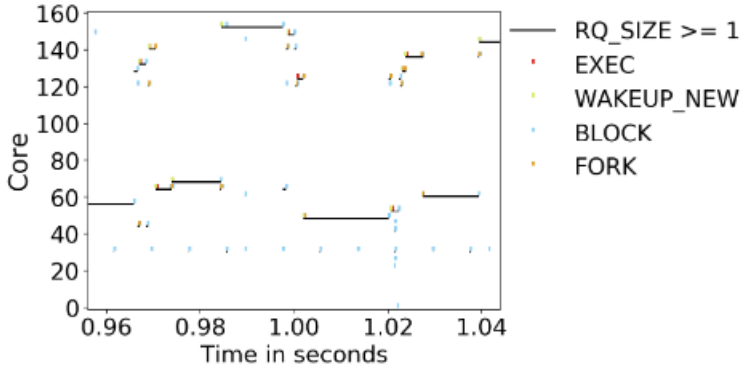


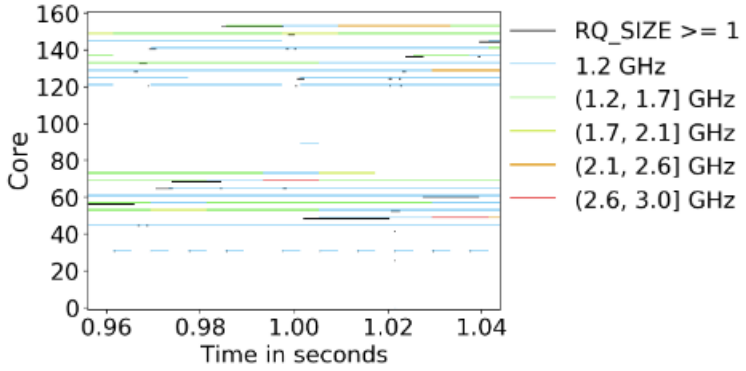
图 1　使用 320 个作业构建 Linux 内核版本 5.4 时的执行跟踪

为了证实我们对 FTL 的直觉，我们开发了一个细粒度工具[1]，使用 powersave governor 来监控单个内核在执行隔离忙循环时的频率。如图 3 所示，任务运行 0.20 秒，如图中开始和结束垂直线所示。内核从其最低频率 1.25 GHz 升至最高频率 3.00 GHz 需要 29 毫秒的 FTL 才能完成任务。当任务结束时，核心需要大约 10 毫秒才能回到其初始频率，但 FTL 的持续时间因以下因素而变得复杂：频率往往会反弹几次，持续约 98 毫秒，然后才能稳定在核心运行的

最低频率。这些测量结果与我们对图 2(b) 的解释一致：几十毫秒的 FTL 明显长于图中可见任务的执行时间，因为最长的任务在 1.00 秒和 1.02 秒之间运行了大约 20 毫秒。请注意，FTL 的持续时间主要取决于硬件检测负载变化，然后决定更改频率的时间。之前的工作[22] 表明核心改变其频率的实际延迟仅为 IntelR© CPU 上的数十微秒。



(a) Scheduler events.



(b) Core frequencies.

图 2　放大图 1 的稀疏区域

回到图 2(a)，我们观察到的现象如下。 Linux 构建的（最近）连续阶段中的计算，通过 fork() 和 wait() 系统调用作为进程顺序启动，并且这些计算的执行时间比 FTL 短。因此，核心在执行计算后会加速，即使此时计算已转移到新 fork 的进程，如果机器处于空闲中，这些进程可能会在最近未使用的核心上运行。事实上，CFS 经常为任务选择不同的内核来唤醒，如果大多数内核处于空闲状态，则很可能最近没有使用所选内核，因此运行频率较低。发起 fork() 的任务在不久之后执行 wait() 操作，那么在执行的过程中频率增加的部分就被浪费了。我们遇到了反复出现的频率反转问题，这是由一个非常常见的场景引起的：启动一系列顺序进程，就像在 shell 脚本中通常做的那样。通过 fork() 和 wait() 系统调用顺序创建进程并不是反复出现频率反转的唯一原因。这种现象也可能发生

在相互解除阻塞的轻量级线程中，在生产者-消费者应用程序中也很常见。实际上，为新任务选择核心以在其上唤醒的 CFS 代码也用于为已存在的唤醒任务选择核心。请注意，CFS 不会根据任务类型（即进程或线程）使用不同的代码路径.

3 防止频率反转的策略

由于频率反转是调度决策的结果，我们认为它必须在调度程序级别解决。根据我们的经验，对调度程序的每次更改都可能对某些工作负载产生不可预测的后果。变化越复杂，后果就越难以预测。因此，对调度程序提出广泛或复杂的更改，或者完全重写，会使性能提升的来源变得不清楚。力求最小化、简单化的更改使得能够与 CFS 进行同类比较。我们提出了两种解决频率反转问题的策略。第一个是提供良好性能表现但在某些调度场景中可能会出现最坏情况简单策略。第二种解决方案旨在获得与第一种解决方案相同的好处，同时以牺牲一些简单性为代价最大限度地减少最坏情况发生的可能。

3.1 在本地放置线程

我们提出的第一个防止频率反转的策略是 Slocal：当一个线程被创建或解除阻塞时，它被放置在与创建或解除阻塞它的进程相同的核心上。在通过 fork() 和 wait() 系统调用创建单个进程的上下文中，这种策略意味着创建的进程更有可能运行在高频核心上，因为核心的频率可能已经很高了甚至影响到了父进程的活动。此外，如果父进程在不久之后调用 wait() ，则两个进程在同一内核上运行的持续时间将受到限制。在生产者-消费者应用到上下文中，当生产者线程唤醒消费者线程时，这种策略再次暗示消费者线程更可能运行在高频核心上，并且有两个进程运行的持续时间中如果生产者的最后一个动作是在阻塞或终止之前唤醒消费者，则在同一核心上将再次受到限制。然而，在某些情况下，Slocal 可能会损害性能：如果创建或唤醒另一个任务之后没有快速阻塞或终止，则创建或唤醒的任务将等待 CPU 资源一段时间。 Linux 调度程序的周期性负载平衡器可以缓解此问题，它将其中一个任务迁移到另一个负载较少的核心。但是，等待下一个负载平衡事件可能会很长。在 CFS 中，周期性的负载均衡是分层进行的，因此会出现周期不同即同一个缓存域中的核心比不同 NUMA 节点上的核心更频繁地进行均衡。在大型机器上，这些周期可能从 4 毫秒到数百毫秒不等。 Slocal 通过完全替换其线程放置策略显着改变了 CFS 的行为。此外，上述缺点使其成为某些工作负载的高风险解决方案。鉴于我们之前设定的先决条件，这两个问题都使该解决方案不能令人满意。

3.2 推迟线程迁移

为了在不等待定期负载平衡的情况下修复问题，我们提出了第二种策略，即 Smove。使用 vanilla CFS，当创建或唤醒线程时 cfs 决定它应该在哪个核心上运

行。 Smove 推迟使用这个选定的内核，以允许唤醒线程利用更有可能以高频运行的内核。令 Twoken 为新创建或唤醒的任务，Cwaker 为创建或唤醒 Twoken 的任务 Twaker 正在运行的核心，CCFS 为 CFS 选择的目标核心。调度程序的正常行为是直接将任务 Twoken 排入 CCFS 的运行队列。我们建议延迟此迁移，以允许 Twoken 在 CCFS 以低频率运行时更有可能使用高频核心。首先，如果 CCFS 的运行频率高于 CPU 的最低频率，我们将 Twoken 放入 CCFS 的运行队列中。否则，我们准备一个高分辨率定时器中断，它将在 D μs 内执行迁移，并将 Twoken 排入 Cwaker 的运行队列。如果在 Cwaker 上安排了 Twoken，则取消定时器。 Smove 背后的基本原理是，如果任务可以在本地放置在可能以高频运行的内核上时快速执行，我们希望避免唤醒低频内核。确实，Twaker 在投放的时候是在运行的，也就是说 Cwaker 很可能运行频率很高。延迟 D μ s 可以在运行时通过写入 sysfs 伪文件系统中的参数文件来更改。我们选择了 50 μs 的默认值，这接近于我们的 Linux 内核构建实验期间 fork 和 wait 系统调用之间的延迟。我们发现在 25 μs 和 1 ms 之间改变此参数的值对第 4 节中使用的基准测试没有显着影响。

4 评估

本节旨在证明我们的策略可以提高大多数工作负载的性能，同时不会降低能耗。我们运行 Phoronix 基准套件，NAS 基准套件[23]，以及其他应用程序的广泛应用程序，例如 hackbench（Linux 内核调度程序社区中的一个流行基准）和 sysbench OLTP（一个数据库基准）。这些实验在配备 80 核 IntelR© CPU 的服务器级 4 路 NUMA 机器和配备 4 核 AMDR© CPU 的台式机上运行（表 1）。两个 CPU 都可以选择每个核心的独立频。 我们在 2019 年 11 月发布的最新 LTS 内核 Linux 5.4 中[3]实现了 Slocal 和 Smove，并将我们的策略与 Linux 5.4 进行了比较。实施 Slocal（或 Smove）只需要修改 CFS 中的 3（即第 124）行。我们运行所有实验 10 次。使用 IntelR© RAPL[19]在两台机器上评估能耗，该功能测量 CPU 插槽和 DRAM 的能耗。性能结果是每个基准测试报告的结果，因为它们涉及不同的指标，例如执行时间、吞吐量或延迟，并且单位不一致。为了更好的可读性，以下所有图表都显示了与 CFS 运行的平均值相比在性能和能源使用方面的改进。因此，无论测量单位如何，总是越高越好。 CFS 结果的平均值显示在所有基准的图形顶部，并带有基准的单位。在 Linux 中，频率由称为调控器的子系统控制。在现代 IntelR© 硬件上，powersave governor 将频率的选择委托给硬件，因为它可以执行更细粒度的调整。硬件频率选择算法试图在对性能影响最小的情况下节省能源。硬件根据各种试探方法（例如退休指令的数量）来估计内核的负载。这是大多数 Linux 发行版上 IntelR© 硬件的默认调控器。

schedutil 调控器，自 Linux 4.7（2016 年 7 月）起由 Linux 社区开发，试图将控制权交还给操作系统。它使用内核调度程序 CFS 的内部数据来估计每个内核的负载，并相应地更改频率。另外两个调控器，performance 和 ondemand，在 Linux 中可用，但我们不感兴趣：前者以最高频率运行所有内核，从而禁用动态缩放，而现代 IntelR© 处理器不支持后者。为了证明我们的工作与使用的调控器正交，我们使用 powersave 和 schedutil 来评估我们的策略。我们首先展示 IntelR© 服务器上的完整结果，然后总结 AMDR© 台式机上的结果。然后我们重新审视我们的内核构建案例研究并研究一些最坏情况的结果（mkl、hackbench）。最后，我们讨论 Smove 策略的开销。

表 1　我们的实验机器配置

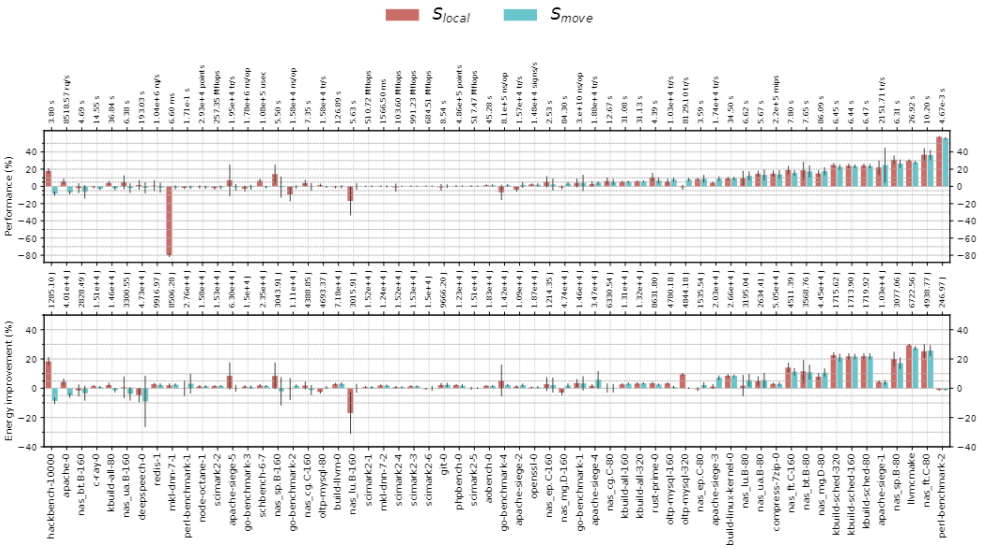
CPU vendor	Intel®	AMD®
CPU model	Xeon E7-8870 v4	Ryzen 5 3400G
Cores (SMT)	80 (160)	4 (8)
Min freq	1.2 GHz	1.4 GHz
Base freq	2.1 GHz	3.7 GHz
Turbo freq	3.0 GHz	4.2 GHz
Memory	512 GB	8 GB
OS	Debian 10 (buster)	Arch Linux

4.1 使用 powersave 执行

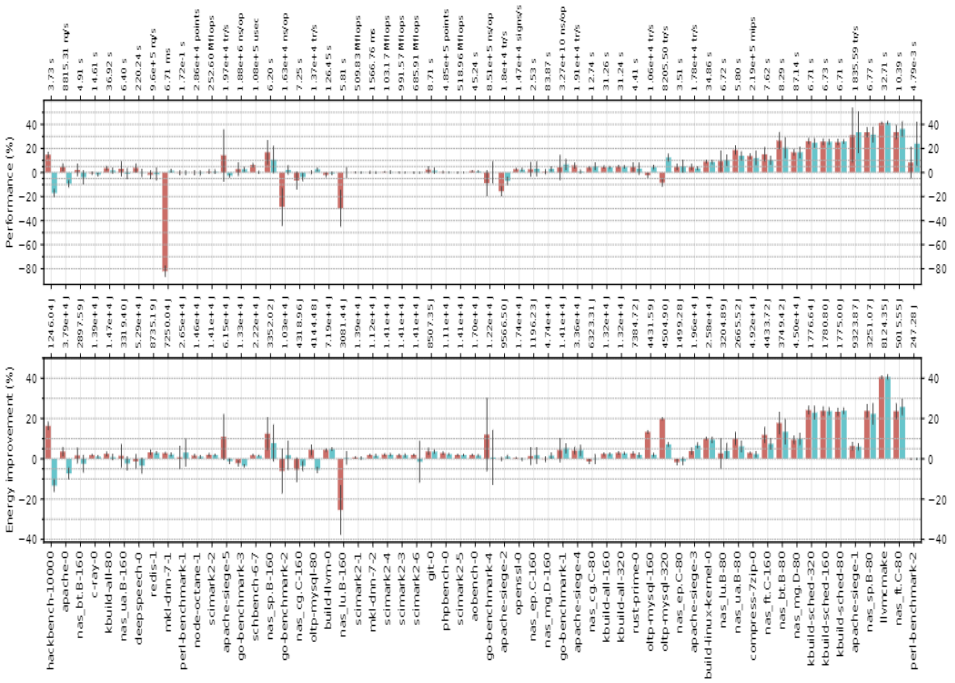
我们首先考虑 powersave 下的执行。图 4(a) 显示了与 CFS 相比，Slocal 和 Smove 在性能和能耗方面的改进。我们认为与 CFS 相比不超过 5%的误差。

性能：Slocal 和 Smove 总体表现良好，在 60 个应用程序中分别有 27 个和 23 个优于 CFS。正如预期的那样，这些策略的最佳结果是在广泛使用 fork 或者 wait 模式的基准测试中得到的，因此表现出大量的频率反转。在最好的情况下，Slocal 和 Smove 在 perl-benchmark-2 上分别提高了 58% 和 56%，perl-benchmark-2 衡量了 perl 解释器的启动时间。这个基准从避免频率反转中受益匪浅，因为它主要由 fork 或者 wait 模式组成。在性能损失方面，这两种策略都只降低了 3 个应用程序的性能，但降低的百分比不同。 Slocal 使 mkl-dnn-7-1 恶化了 80%，nas_lu.B-160 恶化了 17%，而 Smove 在 hackbench 上的最坏情况恶化了 8.4%。

能源消耗：总的来说，Slocal 和 Smove 都改善了能源使用。在我们的 60 个应用程序中，与 CFS 相比，我们分别将 16 个和 14 个应用程序的能耗提高了 5% 以上。大多数改进都体现在性能也得到提高的基准测试中。在这些情况下，节能可能主要是由于应用程序的执行时间较短。然而，我们也看到了一些性能与 CFS 相当的应用程序的改进。



(a) 与使用 powersave 调节器的 CFS 比较



(b) 与使用 schedutil 调控器的 CFS 的比较

图 4 性能和能耗改进 w.r.t.服务器机器上的 Linux 5.4（越高越好）

这是因为我们避免唤醒处于低功耗状态的内核，因此节省了启动和运行这些内核所需的能量。在损耗方面，Slocal 仅在一个应用程序 nas_lu.B-160 上比 CFS 消耗更多的能量。这种损失是由于 Slocal 在此应用程序上的不良性能造成的。该基准测试的指标是其执行时间，增加执行时间而不相应降低频率会增加能耗。Smove 在两个应用程序上比 CFS 消耗更多的能量：hackbench，因为性能损失，

以及 deepspeech，其标准偏差太高，导致结果没有意义。

总体得分：为了比较我们的策略的整体影响，我们计算所有运行的几何平均值，其中每次运行都统一为 CFS 的平均结果。 Smove 的性能提高了 6%，能源使用量减少了 3%，这两个指标相结合提高了 4%。 Slocal 具有相似的总体得分（始终为 5%），但其最差情况表明 Smove 是通用调度程序的更好选择。这些微小的差异是意料之中的，因为我们评估的大多数应用程序与 CFS 和我们的策略的表现相似。我们还使用 t 检验评估结果的统计显着性。p 值最多为 $3 * 10^{-20}$，我们认为我们的结果具有统计意义上的可靠性。

4.2 使用 schedutil 执行

接下来，我们考虑在 schedutil 调控器下执行。作为基准，图 5 首先显示了与使用 CFS 的 powersave 调控器相比，schedutil 调控器在性能和能源方面的改进。总的来说，我们观察到 schedutil 调控器在提高能源使用率的同时降低了大多数应用程序的性能。这表明这个新调控器在节能方面比硬件实施的调控器更积极。我们省略了原始值，因为它们已经在图 4(a) 和 4(b) 中展示出来了。图 4b 显示了在使用 schedutil 调控器时，与 CFS 相比，我们的策略在性能和能耗方面的改进。

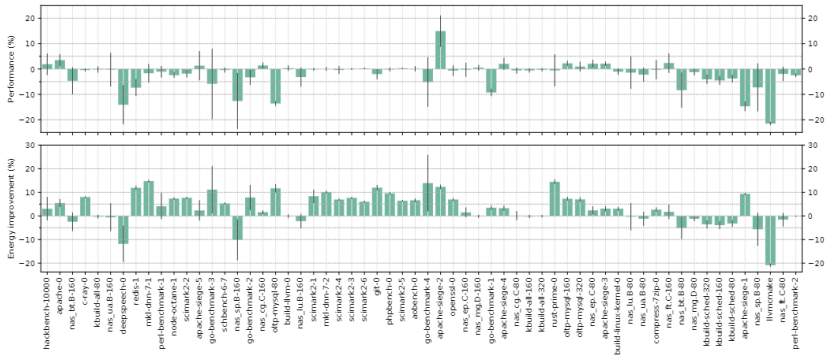


图 5　schedutil 的性能与服务器计算机上使用 CFS 的 powersave 的比较

性能:在 60 个应用程序中，Slocal 和 Smove 分别在 22 个和 20 个应用程序上表现优于 CFS。相关的应用程序与使用节电调速器改进后的应用程序相同。然而，在性能损失方面，Slocal 比 Smove 更受 schedutil 调控器的影响，有 7 个应用程序的性能比 CFS 更差，而只有 2 个应用程序相对较好。

能源消耗:带有 CFS 的 schedutil 在能源使用方面的整体改进表明我们可能会看到与 Slocal 和 Smove 相同的趋势。事实上，结果与我们观察到的 powersave 调速器非常相似。

总体得分:对于 schedutil 和 Smove，此调控器的几何平均值如下：性能为 6%，能源为 4%，两个指标组合为 5%。 Slocal 有类似的结果（分别为 2%、6% 和 4%），但最坏的情况对于通用调度程序来说仍然太不利了。这些结果也具有统计显着性，p 值最多为 $3*10^{-20}$。

4.3 在台式机上的评估

我们在表 1 中显示的较小的 4 核 AMDR© 台式机 CPU 上评估 d 了我们的策略。与 IntelR© CPU 相比，AMDR© CPU 上的 powersave governor 始终使用最低的可用频率，使我们策略在上下文中无法使用。因此，我们在此机器上使用 schedutil 调控器。如图 6 所示，我们观察到与服务器计算机上相同的总体趋势。 Slocal 和 Smove 在改进性能时表现相似，而 Smove 在性能下降的少数基准测试中表现更好。我们测得 Smove 在最坏情况下减速 11%，在最佳情况下加速 52%，总性能提高 2%。此外，Smove 将 7 个应用程序的性能提高了 5% 以上，而在相同规模下仅降低了 4 个应用程序的性能。 Slocal 策略在改进和降级应用程序的数量方面给出了相同的结果，但遇到更糟糕的边缘情况。其最佳性能提升为 42%，而其最差性能提升为 25%，总性能提升为 1%。我们得出结论，即使没有重大的全局改进， Smove 仍然是消除核心数较少的机器上的频率反转的好策略。我们的性能结果具有统计显着性，Smove 的 p 值为 0.0005，Slocal 为 0.03。在能源消耗方面，与 CFS 相比，Slocal 和 Smove 似乎几乎没有影响。然而，我们能够通过所有三种策略收集到的度量有很大的差异，我们在 IntelR© CPU 上没有观察到这一点。我们怀疑这是由于 AMDR© 处理器上可用的与能量相关的硬件计数器或这些计数器缺乏良好的软件支持。

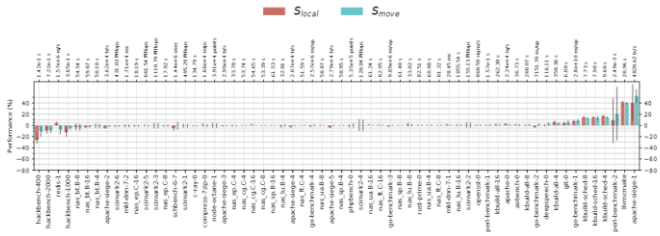


图 6　性能改进 w.r.t.台式机上的 Linux 5.4（越高越好）

4.4 深入分析

我们现在对特定基准进行详细分析，这些基准在我们的解决方案中表现特别好或特别差。在本节中，所有跟踪都是通过 powersave 调速器获得的。

Kbuild　图 7 使用 CFS（顶部）和 Smove（底部）构建 Linux 内核的执行情况。在 CFS（0-2 秒、2.5-4.5 秒、17-22 秒）上多核以低频运行的大部分连续阶段中，Smove 在较高频率下使用较少的核。这主要是由于 fork()/wait()模式：由于 waker 线程在 fork()后不久调用 wait()，Smove 定时器不会超时，被唤醒的线程仍然在高频运行的本地核心上，从而避免频率反转。长并行阶段之前的阶段在 CFS 上的执行时间为 4.4 秒，而在 Smove 上仅需 2.9 秒。为了更好地了解 Smove 的影响，图 8 显示了 kbuild-sched-320 基准测试，它仅构建 Linux 内核的调度程序子系统。在这里，并行阶段比完整构建要短得多，因为要编译的文件更少，从而使执行的顺序阶段更加可见。同样，我们看到使用的内核更少，频率更高。

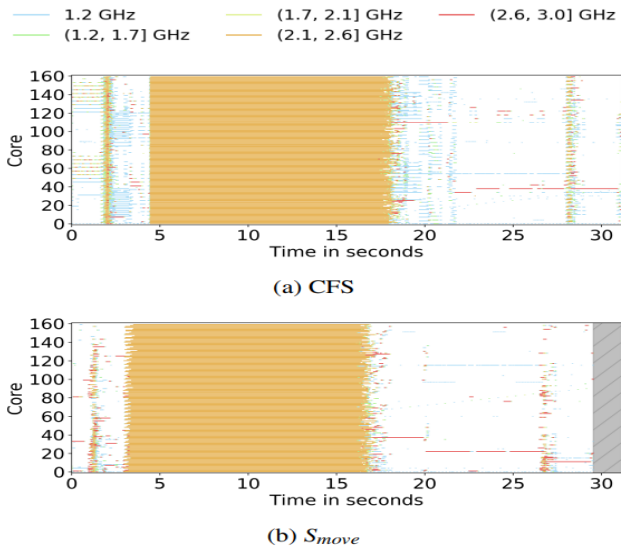
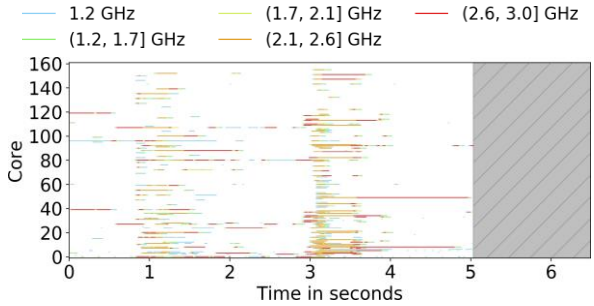


图 7　使用 320 个作业构建 Linux 内核版本 5.4 时的执行跟踪

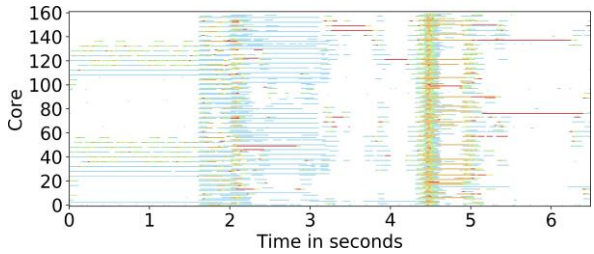
mkl mkl-dnn-7-1 基准测试是 Slocal 的最坏情况：所有线程保持阻塞和解除阻塞，因此避免周期性负载平衡并继续返回同一组内核。因此，与另一个线程共享一个核心的线程将倾向于使用 Slocal 策略保留在那里。图 9 显示了每个内核的运行队列上的线程数，所有三个调度程序都带有 powersave 调控器。黑线表示运行队列中有一个线程，红线表示有多个。 CFS 将线程快速分布在所有核心上，并在不到 0.2 秒的时间内实现了每个核心一个线程的平衡。另一方面，Slocal 试图最大化核心重用并超额订阅 36 个核心。这导致永远不会使用所有内核，在多个内核过载的情况下最多可实现 85% 的 CPU 利用率。正如 Lozi 等人所定义的，这是对工作守恒属性的持续违反[21]。即如果一个核心有多个线程，则没有核心空闲在它的运行队列中。有趣的是，在我们的实验中，分散线程的平衡操作是由于系统或守护线程（例如 systemd）唤醒并立即阻塞，从而触发调度程序的空闲平衡。在后台没有任何运行的机器上，我们可能会长时间处于过载状态，因为空闲内核上的滴答被停用，从而消除了定期平衡的机会。我们可以在 nas-lu.B-160 上看到相同的模式，这是另一个不适用于 Slocal 的基准测试。Smove 通过在可配置的延迟后将过载内核的线程迁移到可用的空闲内核来解决该问题。

hackbench hackbench-10000 基准测试是 Smove 策略在性能方面最差的应用程序。这个微基准对于调度程序来说压力特别大，有 10000 个正在运行的线程。然而，展示的模式很有趣，可以更好地理解 Smove 的缺点，并提供有关如何改进我们策略的见解。该基准测试分为三个阶段：线程创建、通信和线程终止。图 10 显示了在使用 CFS、Slocal 和 Smove 执行 hackbench 期间所有内核的频率。第一阶段对应于所有三个调度程序的前两秒。主线程使用 fork()系统

调用创建 10,000 个线程，所有子线程立即等待屏障。使用 CFS，子线程被放置在空闲核心上，当线程到达屏障时，这些核心再次空闲。这意味着所有核心都保留。大部分是闲置的。这也导致主线程在这个阶段保持在同一个核心上。但是，Slocal 和 Smove 将子线程放置在本地，导致主线程核心的超额订阅和负载均衡器的迁移。因此，主线程本身有时会从一个核心迁移到另一个核心。当所有线程都创建后，主线程释放等待屏障的线程并等待它们终止，从而开始第二阶段。在此阶段，子线程通过在管道中读写进行通信。 CFS 试图均衡所有内核之间的负载，但它的启发式算法对跨 NUMA 节点的迁移造成巨大的性能损失，因此单个节点以高频率运行（内核 0、4、8 等在我们的机器上共享同一个节点）而其他核心几乎没有工作要做，并且以较低的频率运行此阶段在 2.8 秒时结束。



(a)CFS



(b)Smove

图 8　使用 320 作业构建 Linux 内核 5.4 版 sched 目录时的执行轨迹

执行的其余部分是主线程收集其子线程并终止。 Slocal 积极地打包线程，导致第二阶段的运行队列很长，因此由于大量诱发的过载而促进了节点之间的负载平衡。然而，Slocal 仍然没有使用所有内核，主要是避免在超线程内核对上运行（内核 n 和 n + 80 在我们的机器上是超线程的）。 Slocal 运行第二阶段的速度比 CFS 快，它在 2.5 秒时终止，因为它始终使用一半的高频内核，而许多其他内核以中等频率运行。另一方面，Smove 在第二阶段表现不佳，仅用了 3.4 秒就完成了。该行为似乎非常接近 CFS，四分之一的核心仍以高频率运行。但是，Smove 会导致其他内核出现更多闲置或低频。这是由于 Smove 在本地放置线程：许多线程争用本地核心；一些能够使用该资源，而另一些则在触发定时器中断时迁移。与 CFS 相比，多余的延迟会导致空闲，而迁移也会使内核空闲，与 Slocal 相比降低了它们的频率。此外，当线程因为定时器到期而迁移时，它们都被放置在同一个核心上，并超额订阅它。对于 hackbench，选择中间立场是

最糟糕的策略。我们还可以注意到，由于此工作负载的高波动性，负载平衡无法缓解这种情况。 Lozi 等人也在数据库应用程序上证明了这个问题[21]。这种 hackbench 设置是一种极端情况，在现实生活中不太可能发生，机器超载（10000 个线程）和高度不稳定的应用程序。这个微基准只对研究我们策略的行为有意义。尽管如此，Smove 的性能仍优于 Slocal。

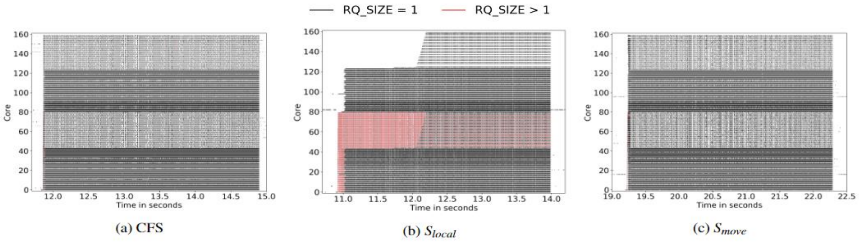


图 9 mkl-dnn-7-1 执行期间每个内核的线程数

4.5 Smove 的调度开销

Smove 比 Slocal 更复杂，因此我们分析了它与 CFS 相比的开销，作为我们策略的上限。我们确定了两种可能的开销来源：查询频率和使用计时器。首先，我们评估查询核心频率的成本。查询内核的频率主要包括读取两个硬件寄存器并执行一些算术运算，因为当前频率是这两个寄存器频率除以 CPU 的基本频率。尽管与调度程序的其余部分相比，这是一个非常小的计算量，但我们通过在每个滴答事件而不是每次需要时查询此信息来进一步减少它的误差。在我们的基准测试中，我们注意到无论是否在每个滴答处查询频率，性能都没有差异。其次，我们评估在调度程序中触发大量计时器的成本。为此，我们在两个版本的 Linux 上运行 schbench：vanilla 5.4 内核和一个带有定时器的修改版本，在与 Smove 相同的条件下。然而，这里的计时器处理程序并不像 Smove 中那样迁移线程。我们选择 schbench 是因为它执行与 hackbench 相同的工作负载，但作为性能评估，它提供了通过管道发送的消息的延迟而不是完成时间。

表 2 显示了该基准测试的结果。总的来说，两个内核版本的延迟的 99.5% 是相同的，除了 256 个线程，其中定时器有负面影响。我们还可以观察到触发的计时器数量随着线程数量的增加而增加，但在 256 个线程之后下降。这种行为是预期的：更多的线程意味着更多的唤醒，但是当机器开始过载时，所有内核都以高频率运行，并且定时器的启动频率降低。这个临界点在 256 个线程左右到达，因为 schbench 线程不断阻塞，这意味着一次可运行的线程通常少于 160 个。

5 讨论

如前所述，我们提出的解决方案 Slocal 和 Smove 相对简单。我们现在讨论频率反转问题的其他更复杂的解决方案。

高频池：一个可能的解决方案是保持一个核心池以高频率运行，即使没有

线程在其上运行。这将允许将线程放置在一个以高频率瞬时运行的空闲核心上。然而，这个池可能会浪费能量并降低繁忙内核可达到的最大频率，当活动内核的数量增加时频率会降低。

调整放置启发式：我们可以将新的频率启发式添加到现有的放置策略中。然而，使用以更高频率运行的核心与例如缓存局部性之间的权衡并不明确，并且可能根据工作负载和架构而有很大差异。

频率模型：一个内核的频率对其他内核性能的影响是硬件特定的。如果调度程序要做出与频率相关的决策，它还需要考虑其决策对所有内核频率的影响。此类模型目前不可用，创建起来会很复杂。

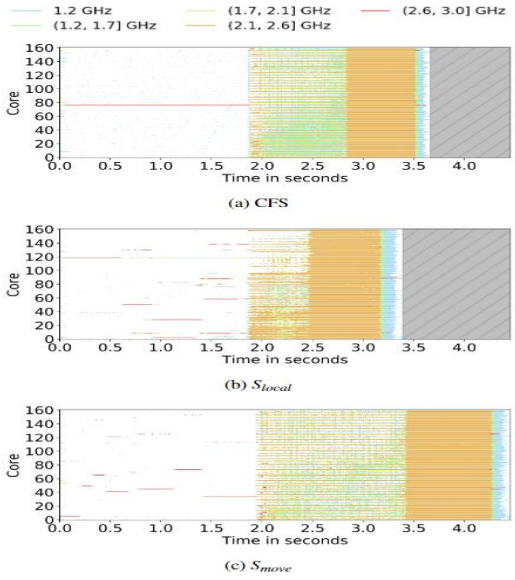


图 10　执行 hackbench 时的核心频率

6 相关工作

动态频率缩放：二十多年来，人们一直在研究使用 DFS 来减少能源使用。韦斯等[33]第一个提出根据其负载调整 CPU 频率，目的是最大化每焦耳指标的数百万条指令。在此之后，在 2000 年代初期，Chase 等[11] 以及 Elnozahy 等[17] 建议减少表现出工作负载集中的未充分利用服务器的频率。 Bianchini 和 Rajamony 在 2004 年的一项调查中总结了这些早期工作[6]。如今，在硬件方面，大多数 CPU 都支持 DFS，最近的产品系列拥有精心设计的硬件算法，能够为同一芯片上的内核动态选择非常不同的频率，采用增强型英特尔 SpeedStepR© [2] 等技术和 AMDR© SenseMI [4]。尽管如此，近年来随着 DFS 逻辑从软件方面向硬件方面的转变，在 Linux 中开发实验性 schedutil[9]调控器的决定是基于软件仍然在 DFS 中发挥作用的想法，因为它更了解加载正在执行。同样，我们的策略

表明，由于 FTL，软件将任务放置在高频内核上比等待硬件在任务放置后增加内核频率更有效。

跟踪低效的调度程序行为：Linux 内核自带的 Perf [15,16,32] 支持通过 perf sched 命令监控调度器行为。虽然 perf sched 可以非常准确地分析调度程序在简单工作负载上的行为，但它对 Linux 内核构建和其他实际工作负载有很大的开销。洛齐、等[21] 识别 Linux 调度程序中的性能错误。为了分析它们，他们编写了一个基本的分析器来监控每个内核的排队线程数和负载。他们的基本分析器不监视调度事件。我们在本文中使用的 SchedLog 和 SchedDisplay[10] 可以以低开销，记录有关所有调度程序事件的相关信息，并通过功能强大且可编写脚本的图形用户界面有效地浏览大量记录的数据。Mollison 等人[25]将回归测试应用于调度程序。他们的重点仅限于实时调度程序，没有考虑 DFS。更一般地说，他们一直在努力测试和了解 Linux 调度程序对性能的影响。自 2005 年以来，LKP 项目[12]一直专注于寻找性能回归，并且社区已经提出了无数可以识别内核中的性能错误的工具[7, 18, 26, 28]。然而，这些工具的重点是检测内核代码内部的减速，而不是检测由内核决定引起的应用程序代码的减速。因此，他们无法检测到糟糕的调度行为。

改进调度程序行为：大多数以前的工作都集中在通过改进特定性能指标的新策略来改进通用操作系统调度，例如减少对共享资源的争用[31、35]、优化 CPU 缓存的使用[29、30]，改善 NUMA 局部性[8, 14]或最小化闲置[20]。这些论文在他们的实验中系统地禁用了 DFS。默克尔等[24]提出了一种调度算法，通过共同调度使用互补资源的应用程序来避免资源争用。它们通过降低执行不利工作负载的核心频率来减少争用。Xiao zhang 等[34]提出了一种促进 DFS 的多核架构调度策略，尽管它们的主要重点是减少缓存干扰。他们只考虑每个芯片的 DFS，因为当时每个内核的 DFS 并不常见。 Linux 内核开发人员最近关注 DFS 和涡轮频率[13]，因为人们发现，在先前空闲的内核上运行的短暂抖动过程可以使该内核切换到涡轮频率，从而减少使用的频率由其他内核——即使在抖动过程完成之后。为了解决这个问题，提出了一个补丁[27]来明确标记抖动任务。调度程序然后尝试将这些标记的任务放置在活动的核心上，并希望保持活动状态。相比之下，我们发现的频率反转问题并不是由涡轮频率引起的：它可能发生在任何 DFS 策略中，其中不同的内核可能以不同的频率运行。

子进程优先：CFS 有一个功能似乎与我们的解决方案相关：sched_child_runs_first。在创建线程时，此功能会为子线程分配一个较低的 vruntime，使其具有比其父线程更高的优先级。如果 CFS 将此线程放在与其父线程相同的核心上，则该线程将抢占父线程；否则，线程只会在别处运行。此功能不影响线程放置，因此无法解决频率反转问题。将此功能与 Smove 结合使用

会破坏 Smove 的目的，因为它总是取消计时器。该策略类似于 Slocal，只是子线程将始终抢占其父线程。

7 总结

在本文中，我们确定了 Linux 中的频率反转问题，该问题发生在具有 DFS 的多核 CPU 上。频率反转导致在低频内核上运行任务，并可能严重降低性能。我们已经实施了两种策略来防止 Linux 5.4CFS 调度程序中出现此问题。实施这些策略需要很少的代码更改，它们可以很容易地移植到其他版本的 Linux 内核。在 60 个不同的应用程序集上，我们证明了我们更好的解决方案即 Smove，通常可以显着提高性能。此外，对于没有出现频率反转问题的应用程序，Smove 对 3 个评估的应用程序造成 8% 或更少的损失。随着独立核心频率缩放成为最新一代处理器的标准功能，我们的工作将针对更多机器。在未来的工作中，我们希望通过将核心频率直接包含在放置算法中来改进调度程序中的线程放置。这种改进将需要考虑各种参数，例如特定于体系结构的 DFS、同时多线程和维护缓存局部性。这项工作得到了 Oracle 捐赠 CR 1930 的部分支持。我们还要感谢匿名审阅者和我们的 shepherd Heiner Litz 的反馈。Linux 5.4 的 Slocal 和 Smove 补丁可在以下位置获得 https://gitlab.inria.fr/whisper-public/atc20。