关于降低移动网页浏览器能耗的技术报告

徐子恒
161160037
161160037@smail.nju.edu.cn

赖伟
161250052
161250052@smail.nju.edu.cn

ABSTRACT

智能手机已经成为了人们生活中必不可少的电子移动设备，但是应用的高能耗导致智能手机的续航始终是一个难以克服的挑战。网页浏览器是智能手机中最核心的应用之一。但因为移动浏览器在性能上进行了大量优化，给移动设备的能源带来了巨大的负担。因此，我们所要介绍的技术正是为了减少智能手机加载网页所消耗的能源，同时尽可能不增加页面加载时间和损害用户体验。

CCS Concepts

• **Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics; • **Networks** → Network reliability;

Keywords

智能手机; 移动网络浏览器; 网页加载; 能源效率

1. INTRODUCTION

网页浏览器是智能手机中最核心的应用之一。但因为移动浏览器在性能上进行了大量优化，给移动设备的能源带来了巨大的负担。因此我们希望提高网页浏览的能效，特别是减少网页加载的能耗。本文中介绍的技术试图在不影响用户体验且不增加页面加载时间的情况下降低智能手机上网页加载的能耗。

首先，我们会介绍浏览器内部的体系结构和系统行为，以了解能量是如何被用于加载网页，从而发现提高能效的机会。尽管许多浏览器制造商都在努力提高移动设备的能效，但调查结果表明，目前的移动浏览器尚未完全针对网页加载进行能源优化。首先，不管网络条件如何，网页浏览器总是在积极进行网络资源处理，这就带来了能源效率低下的风险。其次，绘制率过高，导致大量能量被消耗而没有带来用户可感知的好处。最后，拥有新兴的 ARM big.LITTLE 架构 [2] 的现代 CPU 的节电能力未得到充分利用。从根本上说，在网页加载过程中浏览器过度优化了性能表现，而忽视了能源成本。

为了降低网页加载的能耗，必须重新考虑能源和性能之间的权衡，以制定网页加载的新设计原则。本文会介绍基于这些原则的三种新技术，每种技术对应解决了上述能效问题之一。第一种是使用 *network − aware resource processing*（NRP）[1] 技术来适应不断变化的网络条件，从而实现能耗的降低。这种技术使用了自适应资源缓冲技术来动态控制资源下载速度，从而在不增加页面加载时间的情况下提高能源效率。第二种技术是 *adaptive content painting*（ACP）[1] 技术，这种技术可以避免不必要的内容渲染，从而达到减少能源开销的目的。并在节能和页面加载时间之间做出权衡来确保用户体验不会受到影响。最后，为了更好地利用 big.LITTLE 架构，可以使用 *application − assisted scheduling*（AAS）[1] 技术来利用浏览器的内部知识来制定更好的调度决策。具体来说，这种技术采用了基于 QoS 反馈的自适应线程调度，只要满足相关 QoS 要求，浏览器就可以让线程在小内核上运行以节约能源。

Data Communication May 2018, Nanjing, China

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

2. RELATED WORK

2.1 Energy-efficient mobile web browsing

Zhu 等人 [14] 根据统计推断模型，通过网页基元，HTML 和 CSS 的特征估计页面加载时间和能量消耗，介绍了用于在异构核心上进行移动浏览的网页的高能效，延迟敏感性调度。Butkiewicz 等人 [15] 提出了一个线性回归模型，该模型根据网页的特征（例如图像的数量和大小）和网络服务器（例如服务器/原点的数量）预测页面加载时间。Chameleon [16][17] 在用户提供的约束条件下，在 OLED 移动系统上呈现具有高能效色彩方案的网页。Qian 等人 [18] 深入分析了移动网页浏览如何使用无线网络资源，并提出了节能浏览指南，例如将高网页分解为几个较小的子页面，并减少蜂窝网络中由 JavaScript 触发的延迟或定期数据传输。Zhao 等人 [19] 重新组织 Web 浏览器中的计算阶段和预测的用户阅读时间，

以便快速将无线接口置于基于 3G 的电话的省电 IDLE 状态

2.2 Mobile browsing performance optimization

目前正在开发新的 Web 协议，如 SPDY [20] 和 HTTP 2.0 [21]，可以改善当前 HTTP 协议的性能。它们的主要功能包括（i）将 HTTP 事务多路复用到单个 TCP 连接中，以及（ii）优先处理某些对象负载（例如 JavaScript over image）。最近的一项研究 [22] 发现，SPDY 可以通过使用单个 TCP 连接获得巨大收益，从而显着提高 HTTP 1.1 的页面加载时间。然而，他们还表明，这样的好处可以被网页和浏览器计算中的依赖关系所淹没，并且建议重新构建页面加载过程以减少页面加载时间。除了这种与基础设施相关的方法外，仅开发客户端方法的优点是易于部署，无需基础设施支持。Wang 等人 [23] 表明，两种流行的客户端专用方案（缓存和预取）可能对移动浏览无效，而推测性加载可能有助于克服它们的局限性。Ma 等人 [24] 首先提出并采用主动方法对移动 Web 缓存性能进行全面研究，以确定缓存性能不理想的问题并揭示其根本原因。Meyerovich 等人 [25] 引入了算法来并行化网页布局和渲染引擎以加速浏览器计算。

2.3 Energy saving for mobile apps

Pathak 等人 [26] 以精细的方式审视了智能手机应用程序的能源消耗，报告了一些有趣的发现，例如第三方广告模块消耗了大约三分之二的免费应用程序。Xu 等人 [27] 通过各种技术研究了电子邮件客户端的节能问题，包括减少 3G 尾部时间并将数据传输与数据处理分离。一些研究人员专注于 DVFS 在移动 CPU 上的有效性。最近的一项研究 [28] 考虑了 big.LITTLE 架构的节能问题，其重点在于将核心离线和频率调整集成在一起，而不关注确定哪种类型的核心（大或小）应用运行以提高能效。

3. BACKGROUND

在本节中，我们将介绍 Chromium 浏览器的体系结构以及加载网页时存在的能源浪费问题。

3.1 Chromium Browser Architecture

如图 1 所示，Chromium 使用单个浏览器进程和多个 Renderer 进程的多进程体系结构。每个 Renderer 进程运行一个渲染引擎的实例（以前是 WebKit[4]，现在是 Blink[5]）以及解析和执行 Web 内容的 JavaScript 引擎。每个 Renderer 进程通常对应于 Web 浏览器 UI 中的一个选项卡。浏览器进程运行网络堆栈并从网络中为所有呈现器进程获取网络资源，从而在所有呈现器进程之间共享高效的网络资源。渲染器进程在沙盒环境中运行，对客户端设备和网络的访问受限，防止渲染引擎中的漏洞侵害整个 Web 浏览器。

Chromium 中的每个进程都有多个线程,。进程和线程被设计为异步工作。浏览器进程和 Renderer 进程通过 IPC 机制（例如命名管道）相互通信，并使用共享内存交换数

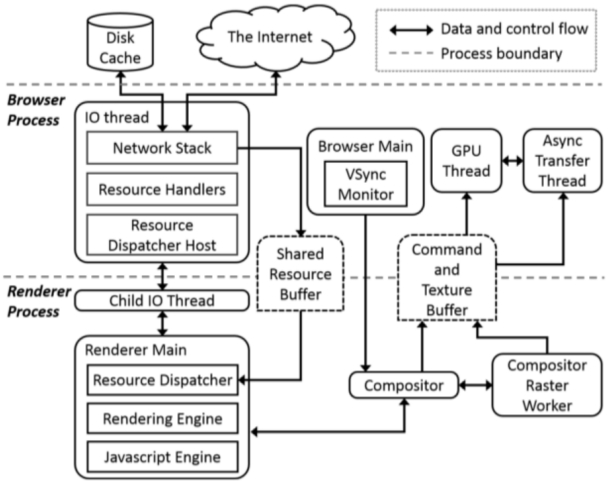

Figure 1: Chromium 浏览器的体系结构

据。浏览器进程将获取的 Web 资源放入共享资源缓冲区，Renderer 进程会从该缓冲区读取数据以创建相应网页的图形层。合成器线程然后将生成的图形数据保存到命令和纹理缓冲区中，以便在浏览器进程中进行 GPU 处理，因为沙盒渲染器进程没有直接访问 GPU 的权限。GPU 线程将最终屏幕图像生成到显示帧缓冲区中以显示在设备屏幕上。

图形组成和 GPU 处理由来自 Android 框架的称为垂直同步（VSync）的回调驱动。每个 VSync 信号指示显示帧的开始，以便在下一个 VSync 之前生成图形数据并将其移动到显示帧缓冲区以供在屏幕上绘制。垂直同步信号每秒产生 60 次。在浏览器进程中，浏览器主线程的 VSync 监视器监视 VSync 信号，并将它们转发到呈现器进程中的合成器线程。

图 1 所示的体系结构也适用于基于 Chromium 源代码构建的 Google Chrome，Opera Mobile 和 Android stock 浏览器 [6][8]。浏览器共享相同的渲染引擎，JavaScript 引擎和底层核心模块。

3.2 Energy Cost Of Page Loading

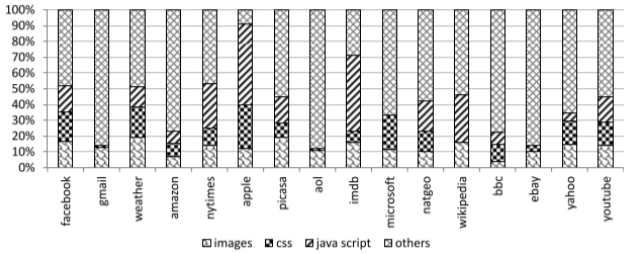

Figure 2: 加载网页的总能耗

Web 资源处理是浏览的主要组件之一，同时还有资源下载和内容渲染。由图 6[13] 可知，Web 资源处理通常包括 HTML 和 CSS 解析，图像解码，Javascript 执行以及更新

DOM 树和图形层。在 Chromium 中，只要呈现器进程通知了浏览器进程中可用的数据，就会执行上述处理。如图 1 所示，这两个进程使用共享资源缓冲区交换数据。浏览器进程在套接字处理程序上使用 read 系统调用来从网络接收网页资源。每个读取系统调用都会将接收到的数据直接写入资源缓冲区，默认情况下最大数据大小为 32 KB。在每次读取系统调用时，无论接收到多少数据，浏览器进程都会立即通知呈现器进程开始处理接收到的数据。

尽管立即处理接收到的数据是很自然的，并且可以最大限度地减少页面加载时间，但这不是节能的。原因在于许多读取系统调用会返回少量数据，从而导致 Browser 进程和 Renderer 进程之间的大量 IPC。平均而言，每个网页的 IPC 总数为 871（每秒 176）。每个 IPC 都有一个固定的开销。此外，网络资源处理中还有其他开销。每次处理数据块时，即使数据块很小，Renderer 进程也必须经历整个数据渲染管道。特别是对于图像数据，Compositor，Raster Worker 和 Async Transfer 线程中涉及许多图形活动。因此，累积的管理费用会浪费大量能源。

加载网页时，Chromium 会逐步处理 Web 资源，并不断将部分渲染的显示结果更新到屏幕上。如图 1 所示，屏幕更新由 VSync 信号驱动。在接收到 VSync 信号后，合成器线程通过共享命令和纹理缓冲区将当前渲染图形数据传送到浏览器进程的 GPU 线程。GPU 线程然后在 GPU 上执行特定于平台的图形命令以将所有纹理合成为最终图形结果以显示在屏幕上。这种数据渲染和屏幕更新的图形管道称为内容绘制。

加载网页时可能会出现许多内容，但是在大多数情况下，内容绘制只会在屏幕上产生非常小的甚至不可见的变化。这些无法察觉或非常小的屏幕变化无助于改善用户体验。但是，它们会导致整个图形处理流水线和两个流程之间的 IPC 发生开销，从而导致不必要的能源成本。

除此以外，big.LITTLE 的节能潜力并没有被充分利用。大多数情况下，线程都是在大内核上执行的，而小内核却处于空闲的状态。

导致这一现象的原因在于操作系统是基于负载驱动方法调度线程的，该方法偏好性能而不是节能。调度程序试图尽快完成一个线程，而不是节省更多的能量。在对称多核架构上，由于所有的 CPU 核心都是相同的，这种负载驱动的方法是节能的。如果一个线程可以更快完成任务，CPU 可以更快地进入睡眠状态以节省能量。[7] 但是，在像 big.LITTLE 这样的异构多核架构中，尽早完成线程可能不会降低能源成本。与一个大内核相比，一个小内核执行每个指令的能耗更低。因此，在较小的内核上运行线程虽然需要更长的时间，但消耗的能量更少。因此，如果一个线程可以容忍延迟，就可以安排在一个小内核上运行。但是，操作系统无法知道线程可以承受多少延迟，因此无法决定是否在较小的内核上运行线程以提高能效。

4. OVERVIEW

在本节中，我们将介绍旨在更好地平衡移动网页加载中的能源成本和性能的几种技术。

4.1 Reducing Javascript Power Consumption

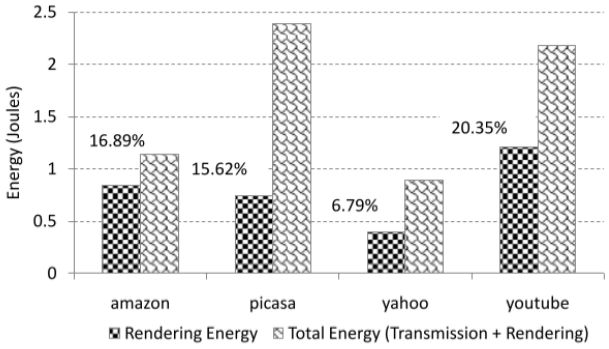

Figure 3: Javascript 的能耗

如图 2 所示 [13]，Javascript 是网页中最耗能的组件之一。很多网页加载了大量的 JavaScript 文件来渲染网页，然而并非所有的脚本都被页面使用，这造成了巨大的能源浪费。

Wikipedia 网页有两个链接到页面的 Javascript 文件 - application.js 和 jquery.js。application.js 是特定于维基百科网站的 Javascript，jquery.js 是通用的 jQuery Javascript 库。在维基百科页面的每个部分，如简介，目录等，可以通过点击每个部分上方的按钮来折叠和展开。jquery.js 中的 Javascript 主要用于单个用途，以根据单击的按钮的 ID 动态识别正确的部分。但是单独加载这个 Javascript 到内存需要 4 焦耳。

为了证明这种能量是可以避免的，我们用不同的 Javascript 重新设计了页面。这一次，每个文本部分和按钮被赋予相同的 id，并且 Javascript 函数使用 $document.getElementById()$ 来获得正确的部分，并且使用 $element.value = show/hide$。application.js 现在被这个简单的 Javascript 取代。我们发现在缓存模式下，修改过的维基百科页面呈现为 9.5 焦耳。只需将 application.js 和 jquery.js 文件作为链接添加到页面，就可以将能耗降至 15 焦耳。

该实验显示，在移动页面上缩小 Javascript 以仅包含页面使用的功能，这大大减少了能源使用。使用通用的 Javascript 库可以简化网页发展，但增加了由此产生的页面使用的能量

4.2 Reducing CSS Power Consumption

与之前的实验类似，我们发现具有未使用的 CSS 规则的大型 CSS 文件消耗的能量多于最低要求的能量。例如，Apple 消耗大量精力下载和呈现 CSS（图 3[13]）。下载和渲染此页面的 CSS 的总能量约为 12 焦耳。这是因为 Apple 主页需要 5 个不同的 CSS 文件，其中包含页面中使用的不同规则。

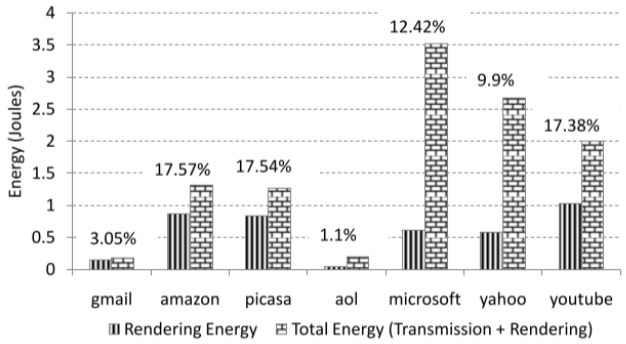

Figure 4: CCS 的能耗

我们通过用仅包含页面所需规则的一个 CSS 文件替换多个 CSS 文件来修改 Apple 站点。这导致了 5 焦耳的能量下降。这约占苹果公司 CSS 总能耗的 40%。这种能量可以通过仅使用 CSS 规则的 CSS 文件来保存。

这表明像 Javascript 一样，CSS 文件应该是页面特定的，并且只包含页面中元素所需的规则。

4.3 Image Formats: Comparison and Optimization

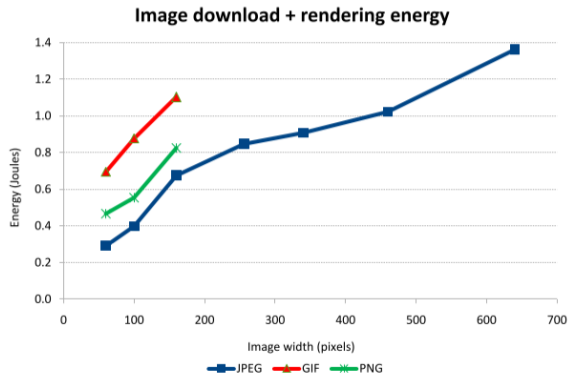

Figure 5: 不同图片格式的能耗

如图 4[13] 所示，不同的图片格式有很大的能耗差距，所以如果在加载网页时把图片进行格式转化可以有效地减少加载网页时的渲染消耗。

4.4 Offloading Browser Computation

由于手机能量有限，我们可以考虑减少其工作强度，因此，依托于云计算技术，我们可以假设将所有的计算转移到云服务器中，只让手机显示结果。在网页浏览的环境中，我们可以将图像渲染（包括压缩和转换为 bitmap（位图））转移到云端，最终手机只显示处理好的位图。已知的浏览器中 Opera 和 SkyFire 采用了这种方法。他们的手机浏览器通过一个代理来与网络交流，这个代理完成了大部分渲染页面的工作。

一般实现方法有两种：

1. 前端代理：web 代理检查所有到达手机的流量，并部分呈现该页面。此时，代理决定在将内容发送到手机之前应该如何修改内容。这种方法以前由旧的 WAP 网关使用，因为他们把 HTML 翻译成 WAP。Opera 和 SkyFire 也是使用这种办法。图 5[13] 展现了这种方法的好处

2. 后端服务器：手机按原样下载网络内容，但随后将某些操作卸载到服务器场。此时手机会决定需要卸载的内容。但就目前的技术水平而言，这是个很难实现的技术。

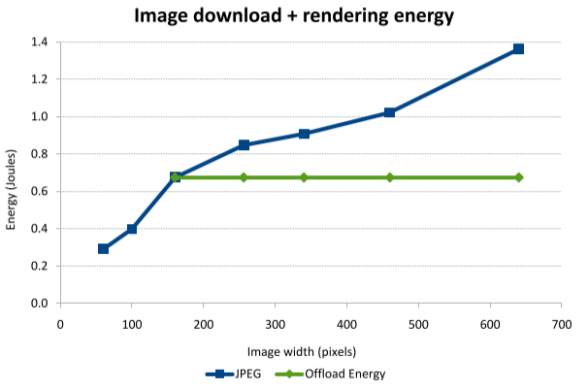

Figure 6: 前端代理的好处

4.5 Network-AwareWeb Resource Processing

进行网络资源处理的频率会影响浏览器的性能和能效。频繁的资源处理显然会导致能源效率低下和大量的能源消耗。但是对网络资源进行批处理可能会延迟 Web 资源的处理并增加页面加载时间。

为此，我们建议设计适应网络资源下载速度的网络资源处理，也就是 $Network-aware\ Resource\ Processing$（NRP）[1]。一个基本的基本原则是下载得越快，批量就应该越大。也就是说，当下载速度较慢时应该使用批量的规模变小，以免在 Web 资源处理中带来过多的延迟。我们强制浏览器进程缓冲从网络接收到的数据，并且只有在缓冲数据量大于阈值时才通知渲染器进程。这样，我们可以减少网络资源处理的次数以及涉及的 IPC 数量，从而可以降低能源成本。

为了平衡资源处理延迟和能源效率之间的平衡，我们设计缓冲区阈值 $buf\_threshold$ 和总缓冲区大小 buf_size 以适应资源下载速度 $net\_goodput$，如下所示:

$$buf\_threshold = \alpha * net\_goodput$$

$$buf\_size = \beta * buf\_threshold$$

其中 $net\_goodput$ 是使用指数加权移动平均线（EWMA）估算的，这是一个基于线性历史的估计器，其平滑因子为

0.3。[9][10] 参数 α 表示一段数据在缓冲区中可以经历的最大等待时间，这个时间的设置需要我们在节能和资源处理延迟之间找到平衡。参数 β 控制总缓冲区大小。缓冲器过度使用时会增加，使用不足时会减少。

附加参数 buf_size 旨在最大限度地减少总内存使用量。处理每个 Web 资源时，它需要分配一个大小为 buf_size 的专用缓冲区。由于加载网页时可能会有很多网络资源，因此如果缓冲区大小很大，则总内存使用率可能较高。通过我们的方法，我们将 buf_size 设计得很小，同时又足够大以允许下载跟上资源处理。

理想情况下，NRP 应该直接适应屏幕上用户感知的内容更改。也就是说，缓冲应该适应 Web 内容在屏幕上显示的速度。然而，从接收到的数据中检测显示的内容的变化是困难的，并且由于像 JavaScript 这样的动态内容而造成很高的开销。因此，我们选择网络吞吐量（接收数据的速率）作为显示内容变化的间接但低开销的指标。通过根据吞吐量来控制缓冲器阈值，我们平衡能量成本和网络资源处理的延迟。

为了进一步减少 NRP 对页面加载时间的影响，我们不将缓冲应用于可能延迟其他资源下载的关键资源。这些关键资源包括主框架的顶层 HTML 资源（称为 Chromium 中的主要资源）以及网页的子框架或 iframe。这些资源在从网络接收到后立即处理。此外，即使块大小小于 $buf\_threshold$，我们也不会缓冲 Web 资源的最后一块，以防止小的最后一个数据块长时间停留在资源缓冲区中。这是通过检测响应是否完成来实现的，这由浏览器中的网络流解析器执行。因此，它也适用于动态内容。如果响应完成，我们通知 Renderer 进程立即开始处理所有收到的数据，而不是等待达到 $buf\_threshold$。

4.6 Adaptive Content Painting

内容绘制需要在用户体验和能量开销之间进行权衡。尽管高帧速率可以提供非常流畅的用户体验，但它可能会导致 GPU 和 CPU 渲染网页时产生高昂的开销，并且与高分辨率结合时会消耗大量能量 [12]。。由于大多数内容渲染只会产生零或非常小的可见屏幕变化，所以多种内容渲染可以聚合在一起以节省能量而不损害用户体验。受此启发，我们将内容绘画设计为适应屏幕上的可见变化，我们将其称为 *adaptive content painting*（ACP）[1]。

我们引入了一个名为 $paint\_rate$ 的新参数，该参数限制了内容绘制的速度并动态适应内容更改速度，因此我们可以聚合内容绘画以节省能量，同时保留用户体验。对于给定的 $paint\_rate$ 值，ACP 强制实际的内容绘制速率不会超过 $paint\_rate$ 的值（实际速率可能低于 $paint\_rate$）。$paint\_rate$ 参数的范围从最小绘画率到最大绘画率。$paint\_rate$ 的初始值被设置为最小值。当内容快速变化时，$paint\_rate$ 参数会增加，反之亦然。在合成器线程将更改后的内容绘制到屏幕上后，如果网页继续更改并需要更新屏幕以反映更改，我们将参数线性地增加 1 以达到最大值。另一方面，当合成器将所有更改传送到屏幕后，当网页显示内容停止改变时，我们将参数降低到最小值。

我们将最大绘制速率限制为 10 帧/秒以平衡能量成本和内容显示的平滑度。我们认为，10 帧/秒应该足够平滑，因为 1）在页面加载时，网页内容仅部分显示，因此对用户体验的影响很小（特别是在小屏幕智能手机上）; 和 2）屏幕上的内容变化率通常很低。此外，现有的研究 [**?**] 表明，延迟高达 100 毫秒通常是不可感知的。因此，我们认为 10 帧/秒（即每帧 100 毫秒）在用户体验和节能之间提供了良好的折衷。当内容变化缓慢时，我们将最小绘制速率设置为 2 帧/秒以节省更多能量。

理想情况下，ACP 技术应量化画框之间的视觉变化以适应内容的变化程度。但是，这样做需要逐个像素比较连续的帧，这会造成大量计算并因此导致高能量成本。因此，我们使用线性增加 $paint\_rate$ 参数的轻量级方法，而无需任何额外的计算成本。

此外，内容绘画应该意识到用户交互。用户触摸屏幕时应使用较高的渲染率，以确保用户体验顺畅。因此，在我们的实施中，我们还会在用户触摸屏幕时检测用户输入和停止速率限制。

NRP 和 ACP 的两种技术并不完全独立。NRP 批量处理资源并因此减慢内容绘画。与 NRP 类似，ACP 还减少了浏览器进程和渲染器进程之间的 IPC 数量，以节省能源。

4.7 Application-Assisted Scheduling

为了更好地利用 big.LITTLE 架构来节省能源，我们建议利用浏览器的内部知识进行节能高效的调度。我们允许浏览器决定一个线程是应该运行在一个大的还是小的核心上，而不是让 OS 任务调度程序通过被动地观察线程的负载来做出所有的调度决策。浏览器比 OS 调度器知道更多有关它们线程的信息，例如线程执行的是什么类型的任务，线程有多重要，线程可以容忍的结束时间，线程的语义以及它们之间的关系。因此，浏览器可能会更好地决定将线程分配给大内核还是小内核。我们称这种技术为 *application − assisted scheduling*（AAS）[1]。

我们设计适用于服务质量（QoS）的 AAS 技术，因为 QoS 对用户体验非常重要。QoS 通过浏览器的帧速率估算。QoS 要求根据 $paint\_rate$ 的变化值进行动态调整。如果 ACP 不工作，则该要求被固定为涂漆速率的最大限制（即 10 帧/秒）。AAS 首先在小内核上分配与 QoS 有关的一组线程，并监视当前帧速率。当前帧速率落后于 QoS 要求时，AAS 认为它是 QoS 的违规。例如，如果 $paint\_rate$ 的值为 5，则将 QoS 定义为在 200 ms 内绘制每个帧。如果浏览器无法在 200 ms 内绘制每帧，AAS 会判定 QoS 已被违反，从而迁移大内核相关的线程。当当前帧速率以稳定的方式超过 QoS 要求时，AAS 将这些线程带回到小内核。AAS 在发现当前帧速率与窗口上的 QoS 要求之间的累积差距不小于阈值时做出这样的决定。因此，在满足 QoS 的同时，其相应的线程保持在小核心上以节约能量而不损害用户体验。

为了更好地利用小核心来节省更多的能量，我们设计了

AAS，以保守的方式将线程从小核心转移到大核心，使用三秒钟的时间窗口，并将线程从大核心移回到小核心核心使用更短的一秒钟时间窗口。

ACP 调整 *paint_rate* 限制，而 AAS 仅监视实际帧速率（帧绘制时间）并安排线程以满足限制。由于 *paint_rate* 只设置最大涂装速率而不是固定涂装速率，因此 AAS 技术和帧速率之间没有严格的循环依赖关系。

AAS 中的小内核和大内核之间的线程迁移对于所有应用程序都是通用的。AAS 技术动态地改变了 QoS 相关线程的亲和性，因此不需要将特定线程硬分配给核心。但是，应由 AAS 管理哪些线程以及如何定义 QoS 是特定于应用程序的。在第 5 节中，我们将向我们展示 Chromium 的哪些线程应用 AAS 技术。

5. CONCLUSION

本文提出了多种有效的技术来优化智能手机网页加载的能耗。包括减少复杂的 JavaScript 和 CSS 以及使用 JPEG 图像，讲计算转移到云端，网络感知资源处理，自适应内容绘制和应用辅助调度。经过实验测试，这些技术能够显着降低网页加载的能源成本。虽然它们会略微增加页面的加载时间，但大部分用户不会察觉到这种增加。

6. REFERENCES

[1] Bui D H, Liu Y, Kim H, et al Rethinking energy-performance trade-off in mobile web page loading. *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking. ACM, 2015: 14-26.*

[2] ARM big.LITTLE technology. http://www.thinkbiglittle.com.

[3] Alexa, Top Sites in United States. http://www.alexa.com/topsites/countries/US.

[4] WebKit. http://www.webkit.org.

[5] Blink. http://www.chromium.org/blink.

[6] Differences between Google Chrome and Linux distro Chromium. http://code.google.com/p/chromium /wiki/ChromiumBrowserVsGoogleChrome.

[7] A. Carroll and G. Heiser. Mobile multicores: Use them or waste them. *Proc. USENIX HotPower, 2013.*

[8] A. Cunningham. New Opera for Android looks like Opera, tastes like Chrome. http://arstechnica.com/ information-technology/2013/05/new-opera-for-android-looks-like-opera-tastes-like-chrome.

[9] Q. He, C. Dovrolis, and M. Ammar. On the predictability of large transfer TCP throughput. *Proc. ACMSIGCOMM, 2005.*

[10] M. Mirza, J. Sommers, P. Barford, and Xiaojin Zhu. A machine learning approach to TCP throughput prediction. *Networking, IEEE/ACMTransactions on, 18(4):1026–1039, 2010.*

[12] K. W. Nixon, X. Chen, H. Zhou, Y. Liu, and Y. Chen. Mobile gpu power consumption reduction via dynamic resolution and frame rate scaling. *HotPower, 2014.*

[13] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption. *Proc. WWW, 2012.*

[14] Y. Zhu and V. J. Reddi. High-performance and Energy-efficient Mobile Web Browsing on Big/Little Systems. *Proc. IEEE HPCA, 2013.*

[15] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding Website Complexity: Measurements, Metrics, and Implications. *Proc. ACM IMC, 2011.*

[16] M. Dong and L. Zhong. Chameleon: A Color-adaptive Web Browser for Mobile OLED Displays. *Proc. ACM MobiSys, 2011.*

[17] M. Dong and L. Zhong. Chameleon: A Color-Adaptive Web Browser for Mobile OLED Displays. *IEEE Transactions on Mobile Computing (TMC), 2012.*

[18] F. Qian, S. Sen, and O. Spatscheck. Characterizing Resource Usage for Mobile Web Browsing. *Proc. ACM MobiSys, 2014.*

[19] B. Zhao, Q. Zheng, G. Cao, and S. Addepalli. Energy-Aware Web Browsing in 3G Based Smartphones. *Proc. IEEE ICDCS, 2013.*

[20] SPDY. http://www.chromium.org/spdy.

[21] Hypertext transfer protocol version 2.0, draft-ietf-httpbis-http2-07. http://tools.ietf.org/ html/draft-ietf-httpbis-http2-07.

[22] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? *Proc. USENIX NSDI, 2014.*

[23] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? *Proc. USENIX NSDI, 2014.*

[24] Y. Ma, X. Liu, S. Zhang, R. Xiang, Y. Liu, and T. Xie. Measurement and Analysis of Mobile Web Cache Performance. *Proc. WWW, 2015.*

[25] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. *Proc. WWW, 2010.*

[26] A. Pathak, Y. C. Hu, and M. Zhang. Where is the

Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. *Proc. ACM EuroSys, 2012.*

[27] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing Background Email Sync on Smartphones. *Proc. ACM MobiSys, 2013.*

[28] A. Carroll and G. Heiser. Unifying DVFS and offlining in mobile multicores. *Proc. IEEE RTAS, 2014.*