申请上海交通大学工程硕士学位论文

嵌入式Linux系统电源管理的改进研究与实现

|  |  |
| --- | --- |
| 学校代码： | 10248 |
| 作者姓名： | 凌云 |
| 学 号： | 1130372134 |
| 第一导师： | 王东 |
| 第二导师： |  |
| 学科专业： | 软件工程 |
| 答辩日期： | 年 月 日 |

上海交通大学软件学院

2016年12月

A Dissertation Submitted to Shanghai Jiao Tong University

for Master Degree of Engineering

IMPROVEMENT AND IMPLEMENTATION OF POWER MANAGEMENT IN EMBEDDED LINUX SYSTEM

|  |  |
| --- | --- |
| University Code： | 10248 |
| Author： | Yun Ling |
| Student ID: | 1130372134 |
| Mentor 1： | Prof. Wang Dong |
| Mentor 2: |  |
| Field： | Software Engineering |
| Date of Oral Defense： |  |

School of Software

Shanghai Jiaotong University

Dec. 2016

**上海交通大学**

**学位论文原创性声明**

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期： 年 月 日

**上海交通大学**

**学位论文版权使用授权书**

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

**保密**□，在 年解密后适用本授权书。

本学位论文属于

**不保密**□。

（请在以上方框内打“**√**”）

学位论文作者签名： 指导教师签名：

日期： 年 月 日 日期： 年 月 日

**嵌入式Linux系统电源管理的改进研究与实现**

**摘 要**

随着嵌入式Linux系统的广泛应用和不断发展，用户对其降低能源消耗和延长电池续航时间的要求也越来越高。因此，嵌入式Linux系统的电源管理技术一直是开发嵌入式Linux设备的研究重点。目前针对这一领域的研究一般集中在系统各个组件分立的电源管理技术，然而在目前不断进化的硬件趋势下，对多核异构不对称处理器架构下的系统任务调度、处理器动态电源管理以及相关组件的统筹管理技术的研究具有更重要的意义。

本文对现有的嵌入式Linux系统电源管理框架和技术进行了深入分析和研究，阐述了电源管理中存在的问题，分别从CPU调度和设备两方面实现电源管理技术的改进。所做的工作主要包括：

1. 深入分析了目前嵌入式Linux系统电源管理各个领域的技术，以及在CPU和设备两方面所存在的问题。
2. 在CPU调度方面，从目前调度器和处理器动态电源管理互相独立没有协调的问题入手，提出并实现了一种通过统筹任务调度和处理器工作电压及频率调整的有节能意识的调度器设计，达到降低处理器功耗的目的。这个设计以改进目前负载追踪算法为基础，从基于功耗成本对任务放置、负载均衡、与处理器工作电压及频率调整协同等几个方面对调度器进行了改进。通过将任务分配到功耗成本最低的处理器核或处理器核簇上，并根据处理器的负载情况调整其工作电压及频率来达到降低功耗的目的。
3. 在设备动态电源管理方面，首先介绍了Runtime PM的设计思想和目标设备Atmel MXT1664触摸屏硬件特性对于动态电源管理的支持，然后从如何确定切换低功耗状态的时机问题入手，提出并实现了一种设备动态电源管理机制，用预测空闲时间的方法确定状态切换时机。通过减少频繁的进入和退出低功耗状态带来的开销，提高动态电源管理效率。
4. 通过在高通MSM8996和APQ8074目标硬件平台上的测试，验证了具有节能意识的调度器设计在CPU空闲时间、高频运行状态时间的改进，设备动态电源管理设计在MXT1664触摸屏控制功耗上的提升。

本文首先介绍了研究背景和嵌入式Linux系统电源管理领域相关文献的综述，然后分析了现有技术存在的问题，接着从CPU调度和设备动态电源管理两方面提出了改进方案并进行了代码实现，最后通过在实际硬件平台上的测试数据验证了改进方案的有效性。

关键词 嵌入式Linux，电源管理，调度器，Runtime PM

**IMPROVEMENT AND IMPLEMENTATION OF**

**POWER MANAGEMENT**

**IN EMBEDDED LINUX SYSTEM**

**ABSTRACT**

With wide adoption and rapid development of Embedded Linux systems, the requirements from users on lowering power consumption and increasing battery life on such systems are much higher than ever. Therefore, power management in Embedded Linux system is always a important part of device development on Embedded Linux. Currently, researches in this domain are mostly focused technologies on individual components of the system. Howerver, with continuing evolving of hardware, researches for cooperating among task scheduling, processor dynamic power management as well as related components under multi core and Heterogeneous Multi-Processing architecture are more important.

We perform a thorough research and analysis on current Embedded Linux system power management architecture and technologies in this paper, unveil problems in current situation, implement improvements from CPU scheduling as well as device power management. Our work includes:

1. Analyzed exisiting technologies in Embedded Linux system power management as well as problems in both CPU scheduling and device power management thoroughly.

2. On CPU scheduling, starting from the isolation of scheduler and dynamic power management of processors and absent of cooperation problem, we proposed energy aware scheduler design. It is designed to lower the overall processor power consumption by corporating task scheduling and adjustment of voltages as well as frequencies of CPU cores.. The design is based on improvement of current load tracking algorithm. It includes improvement on task placement, task balancing as well as co-working with CPUFreq subsystem. It lowers the overall processor power consumption by placing tasks on the CPU core or CPU cluster with the least power cost.

3. On device power management, first we introduced the design principal of Linux Runtime PM framework as well as hardware power management feature in Atmel MXT1664 touch controller. We proposed a device dynamic power management design starting from solving the problem how to figure out when to put device to low power mode. We adopted an idle time prediction mechanism instead of fixed timeout to improve power management efficient and lower power consumption even further by reduce overhead from entering and exiting low power mode frequently.

4. With testing on the Qualcomm MSM8996 and APQ8074 hardware platform, we verified improvement on CPU idle time and high CPU frequency time with the energy aware scheduler design. We also verified improvement on MXT1664 power consumption from the device dynamic power management design..

In this paper, we firstly introduced background of this research and literature survey in Embedded Linux system power management. Secondly, raised problems in existing technologies. Then, we proposed our improvement design on both CPU scheduling and device dynamic power management with implementation in source code. At last, verified effectiveness of our design by test data on hardware platform.

**Keywords** Embedded Linux, power management, scheduler, Runtime PM

**目 录**

[1 绪 论 1](#_Toc471381652)

**[1.1](#_Toc471381653)** [研究背景 1](#_Toc471381653)

**[1.2](#_Toc471381654)** [研究现状 1](#_Toc471381654)

**[1.3](#_Toc471381655)** [本文研究工作 2](#_Toc471381655)

**[1.3.1](#_Toc471381656)** [研究目标 2](#_Toc471381656)

**[1.3.2](#_Toc471381657)** [研究内容与取得的成果 3](#_Toc471381657)

**[1.3.3](#_Toc471381658)** [创新点 3](#_Toc471381658)

**[1.4](#_Toc471381659)** [论文结构 4](#_Toc471381659)

**[1.5](#_Toc471381660)** [本章小结 4](#_Toc471381660)

[2 嵌入式Linux系统电源管理技术分析 5](#_Toc471381661)

**[2.1](#_Toc471381662)** [嵌入式Linux系统电源管理研究综述 5](#_Toc471381662)

**[2.1.1](#_Toc471381663)** [电源管理架构 5](#_Toc471381663)

**[2.1.2](#_Toc471381664)** [静态电源管理 7](#_Toc471381664)

**[2.1.3](#_Toc471381665)** [动态电源管理 7](#_Toc471381665)

**[2.2](#_Toc471381666)** [嵌入式Linux系统电源管理的需求和存在问题 9](#_Toc471381666)

**[2.2.1](#_Toc471381667)** [电源管理需求 9](#_Toc471381667)

**[2.2.2](#_Toc471381668)** [存在的问题 10](#_Toc471381668)

**[2.3](#_Toc471381669)** [本文的研究路线 12](#_Toc471381669)

**[2.3.1](#_Toc471381670)** [有节能意识的调度器 12](#_Toc471381670)

**[2.3.2](#_Toc471381671)** [设备动态电源管理 12](#_Toc471381671)

**[2.4](#_Toc471381672)** [本章小结 12](#_Toc471381672)

[3 嵌入式Linux系统电源管理方法的改进 14](#_Toc471381673)

**[3.1](#_Toc471381674)** [一种有节能意识的调度器设计 14](#_Toc471381674)

**[3.1.1](#_Toc471381675)** [负载追踪的改进 14](#_Toc471381675)

**[3.1.2](#_Toc471381676)** [任务放置的改进 18](#_Toc471381676)

**[3.1.3](#_Toc471381677)** [负载均衡的改进 21](#_Toc471381677)

**[3.1.4](#_Toc471381678)** [与CPUFreq框架的协同 22](#_Toc471381678)

**[3.1.5](#_Toc471381679)** [可调整参数 23](#_Toc471381679)

**[3.2](#_Toc471381680)** [设备驱动的动态电源管理设计 27](#_Toc471381680)

**[3.2.1](#_Toc471381681)** [Runtime PM设计思想 27](#_Toc471381681)

**[3.2.2](#_Toc471381682)** [Atmel MXT1664触摸屏控制器驱动的Runtime PM设计 29](#_Toc471381682)

**[3.2.3](#_Toc471381683)** [设备进入低功耗状态时机的确定 33](#_Toc471381683)

**[3.3](#_Toc471381684)** [本章小结 34](#_Toc471381684)

[4 嵌入式Linux系统电源管理方法的实现 36](#_Toc471381685)

**[4.1](#_Toc471381686)** [一种有节能意识的调度器的实现 36](#_Toc471381686)

**[4.1.1](#_Toc471381687)** [update\_task\_demand()函数实现 37](#_Toc471381687)

**[4.1.2](#_Toc471381688)** [update\_history()函数实现 39](#_Toc471381688)

**[4.1.3](#_Toc471381689)** [update\_cpu\_busy\_time()函数实现 39](#_Toc471381689)

**[4.2](#_Toc471381690)** [设备驱动的动态电源管理实现 41](#_Toc471381690)

**[4.2.1](#_Toc471381691)** [MXT1664驱动实现概述 41](#_Toc471381691)

**[4.2.2](#_Toc471381692)** [Device tree配置数据 41](#_Toc471381692)

**[4.2.3](#_Toc471381693)** [I2C Client驱动实现 42](#_Toc471381693)

**[4.3](#_Toc471381694)** [本章小结 51](#_Toc471381694)

[5 系统功耗测试 52](#_Toc471381695)

**[5.1](#_Toc471381696)** [测试环境 52](#_Toc471381696)

**[5.2](#_Toc471381697)** [测试方法和结果 53](#_Toc471381697)

**[5.2.1](#_Toc471381698)** [有节能意识的调度器测试 53](#_Toc471381698)

**[5.2.1](#_Toc471381699)** [设备动态电源管理测试 54](#_Toc471381699)

**[5.3](#_Toc471381700)** [本章小结 56](#_Toc471381700)

[6 总结与展望 57](#_Toc471381701)

**[6.1](#_Toc471381702)** [本文总结 57](#_Toc471381702)

**[6.2](#_Toc471381703)** [未来工作展望 57](#_Toc471381703)

[附 录 61](#_Toc471381704)

[附录I. update\_task\_demand()函数代码 61](#_Toc471381705)

[附录II. update\_history()函数代码 62](#_Toc471381706)

[附录III. update\_cpu\_busy\_time()函数代码 64](#_Toc471381707)

[附录IV. mxt\_runtime\_suspend()函数代码 70](#_Toc471381708)

[附录V. mxt\_runtime\_resume()函数代码 71](#_Toc471381709)

[附录VI. mxt\_io\_complete()函数代码 72](#_Toc471381710)

[附录VII. mxt\_runtime\_idle()函数代码 72](#_Toc471381711)

[致 谢 75](#_Toc471381712)

[攻读学位期间发表的学术论文目录 77](#_Toc471381713)

**1 绪 论**

**1.1** 研究背景

嵌入式Linux(Embedded Linux)是一类嵌入式操作系统的概称，这类操作系统都是以Linux内核为基础的，被设计用来使用于嵌入式设备，诸如移动电话、个人数字助理(Personal Digital Assistant, PDA)、媒体播放器以及众多的消费类电子装置中。到2015年第三季度为止，根据研究机构Gartner的数据，以安卓(Android)系统为代表的嵌入式Linux设备在全球智能手机市场的份额达到了84.7%。在智联汽车(Connected Car)领域，Tizen嵌入式Linux系统成为了主要的实现平台。在物联网和家庭自动化领域，家庭自动化中心和工业物联网网关均以嵌入式Linux为主。此外，在新兴的可穿戴设备、无人机和机器人领域，嵌入式Linux也日趋成为主流的产品基础。

然而，智能手机等嵌入式设备的电池续航时间已经远远落后于用户需求。根据全球领先的锂电池制造公司之一松下公司估计，相同体积条件下锂电池容量每年的提升仅为11%，在新的电池技术出现之前，现有电池技术的发展已经成为制约嵌入式Linux设备应用推广的一个关键因素。一个突出的例子就表现在目前基于嵌入式Linux的智能手机在功能和电池续航时间上的突出矛盾。因此，如何提高嵌入式Linux系统的电源管理技术、降低设备的功耗、延长设备的电池续航时间等研究就显得极为重要和迫切了。

**1.2** 研究现状

嵌入式Linux系统的电源管理技术并不是单一的技术，而是一个嵌入式系统在各个层面上多种技术的组合，总体上可以分为静态电源管理技术和动态电源管理技术两大类。

静态电源管理技术主要是指在整个嵌入式系统进入空闲状态后，切换到一个更低功耗模式并在需要时唤醒的技术，主要包括基于Linux核心电源管理的系统睡眠与唤醒。在这种情况下，嵌入式系统通常需要将上下文保存到RAM中，然后将CPU切换到睡眠模式。从睡眠状态下唤醒系统通常是通过外部硬件中断，比如电源按键。

动态电源管理技术主要指在系统运行时，通过对系统组件包括CPU和外设等的重新配置，达到以最小的功耗成本满足系统需求的技术。这里的重新配置包括对组件的时钟、电源电压和工作频率的调整。按照设备运行状态的不同，动态电源管理技术又可以分为设备运行时的电源管理技术和设备空闲时的电源管理技术。

设备运行时的电源管理技术主要包括Linux上对动态电压频率调整 (Dynamic Voltage Frequency Scaling, 简称DVFS) 技术[1]的实现，如针对CPU电压和工作频率的CPUFreq框架[2]和操作性能点 (Operating Performance Points，简称OPPs) 机制。

设备空闲时的电源管理技术可分为针对CPU空闲状态的CPUidle框架[3]实现以及在此基础上为减少CPU空闲状态下唤醒源的一系列技术，如无时钟中断空闲(Tickless Idle)[4]和针对外设空闲状态管理的Runtime PM框架[5]，用以将处于空闲状态的外设及时切换到低功耗模式。为更好地实现Runtime PM机制，Linux还引入了电源管理域 (PM domains) 概念，将需要同时进行电源管理的设备分组进行管理。

除了上面介绍的在Linux内核中实现的电源管理技术之外，嵌入式Linux电源管理技术还包括用户空间的应用层面上针对功耗的优化。比如安卓系统中对运动传感器数据读取的批处理技术(Sensor Batching) [6] 和在设备处于静止状态下控制后台服务进程的Doze机制 [7] ，根据液晶显示屏显示的图像内容和环境光强度适配背光亮度和色彩的CABC和LABC技术 [8] 。

实践证明，这些技术对嵌入式Linux系统设备的电池续航和功耗方面的提高有显著作用。然而，随着这个领域硬件的不断发展，电源管理对软件技术也提出了新的要求。当高性能嵌入式系统的CPU从对称多处理(Symmetrical Multi-Processing, 简称SMP)发展到异构多处理(Heterogeneous Multi-Processing, 简称HMP)，如ARM的Big. Little [9] 架构，现有的针对SMP设计的Linux内核调度器已经不能满足需要。此外，目前大多数采用固定超时机制的设备Runtime PM管理也需要改进，以更精准地使处于空闲状态的设备进入低功耗模式。关于这些问题，本文在2.2.2节中将详细描述。

**1.3** 本文研究工作

**1.3.1** 研究目标

本文致力于解决目前嵌入式Linux系统电源管理在HMP CPU架构的调度器设计问题，并提出一种更高效的设备动态电源管理设计。对HMP CPU架构的调度器设计需要解决目前调度器没有与各CPU核功耗方面考虑结合起来的问题。对于设备的动态电源管理，需要解决目前基于固定超时机制造成的能源浪费和效率不高的问题。

**1.3.2** 研究内容与取得的成果

(1) 有节能意识的调度器设计

设计了一种基于当前完全公平(Completely Fair Scheduler, 简称CFS)调度器且可以与CPUFreq框架结合，适用于当前HMP CPU架构的有节能意识调度器。该设计通过基于窗口历史的负载追踪、根据各个CPU核的功耗成本进行任务放置和负载均衡、与CPUFreq框架协作动态调整CPU核的工作电压和频率、提供可调整参数在运行时进行上述功能的调整等五个方面达到降低CPU整体功耗的目的。

(2) 设备驱动的动态电源管理

基于现有的Runtime PM电源管理框架，设计并实现了设备在进入Runtime PM空闲状态、Suspend状态和Resume状态的逻辑；设计并实现了一种通过预测设备空闲时间从而达到适配设备进入低功耗模式时间目的的方法。动态适配的方法避免频繁进出低功耗模式带来的额外开销，同时也解决了固定超时机制带来的能源浪费的问题，提高了动态电源管理效率。

(3) 电源管理改进方案的验证

在高通MSM8996四核处理器硬件平台+Android 6.0.1操作系统环境下，通过BBench和PowerTop工具测试各CPU核在不同功耗和运行频率状态下的时间，验证了具有节能意识的调度器设计在增加CPU空闲时间和降低CPU高频率运行时间上的改进。

在高通APQ8074硬件平台上，通过浏览网页、阅读电子书和游戏三种触摸屏测试，记录MXT1664触摸屏控制器处于低功耗模式的时间，验证了基于Runtime PM的设备动态电源管理设计能够带来设备处于低功耗时间的增加。此外，采用预测空闲时间的方法相比固定超时机制可以有效避免频繁进出低功耗模式带来的额外开销。

**1.3.3** 创新点

在CFS调度器中加入对功耗成本的概念并且将目前互相独立的调度器和CPUFreq框架结合起来，提出一种统筹任务调度和CPU工作频率的具有节能意识的调度器设计

**1.4** 论文结构

本论文共分为五章，内容安排如下：

第一章是绪论。讲述了嵌入式Linux系统电源管理的研究背景、国内外的研究现状、本文的研究目标、研究内容与取得的成果。

第二章是嵌入式Linux系统电源管理现有技术的介绍和存在问题的分析，在此基础上阐述了本文的技术路线。

第三章是对嵌入式Linux系统电源管理技术的改进和设计，深入研究了一种有节能意识的调度器设计和设备动态电源管理设计两大关键技术。

第四章对第三章中提出的设计进行了实现并进行了性能测试。

第五章是结论与展望，对全文进行了总结并提出了一些尚需研究和解决的问题。

**1.5** 本章小结

本章介绍了嵌入式Linux系统电源管理技术的研究背景和现状，指出了改进电源管理技术的必要性。在此基础之上，介绍了本文的研究工作和目标以及取得的成果和创新点，同时给出了论文的结构。

**2 嵌入式Linux系统电源管理技术分析**

在第一章介绍嵌入式Linux系统电源管理技术研究背景和现状的基础上，本章将综述现有电源管理技术架构和技术，针对嵌入式Linux系统的发展趋势提出对电源管理技术的新的需求，以及在CPU调度和设备电源管理两方面存在的问题，进而提出本文的技术路线，即针对CPU调度的有节能意识的调度器设计和基于Runtime PM的设备动态电源管理设计。

**2.1** 嵌入式Linux系统电源管理研究综述

**2.1.1** 电源管理架构

嵌入式Linux系统的电源管理架构是在通用Linux内核电源管理基础之上加入了用户空间电源管理机制形成的，如图2-1表示。其中，从下往上依次为硬件层、Linux内核层和用户空间层，而Linux内核层又可以根据与硬件的依赖关系分为硬件SoC平台相关和平台通用两层。

硬件层通常指的是作为嵌入式Linux系统硬件核心的SoC硬件以及与之相配套的电源管理芯片PMIC。SoC通常在硬件上提供了一系列电源管理特性，比如处理器核的低功耗状态、处理器核心和外设总线时钟频率的设置寄存器、门控时钟(clock gating)等。PMIC通常由SoC硬件以及Linux内核中的PMIC驱动协调控制实现系统中各组件供电的打开、关闭以及电压的调整，可以实现处理器核心在低功耗状态下相应的电压调整。除此之外，在现代SoC集成度越来越高的趋势下，越来越多的处理器核如无线通信基带处理器、数字信号处理器、Wi-Fi／蓝牙／GPS通信处理器等都集成到了同一个SoC系统内，因此SoC内也出现了专门负责资源管理的处理器核心，来统一调度SoC内不同处理器核的电源管理状态。

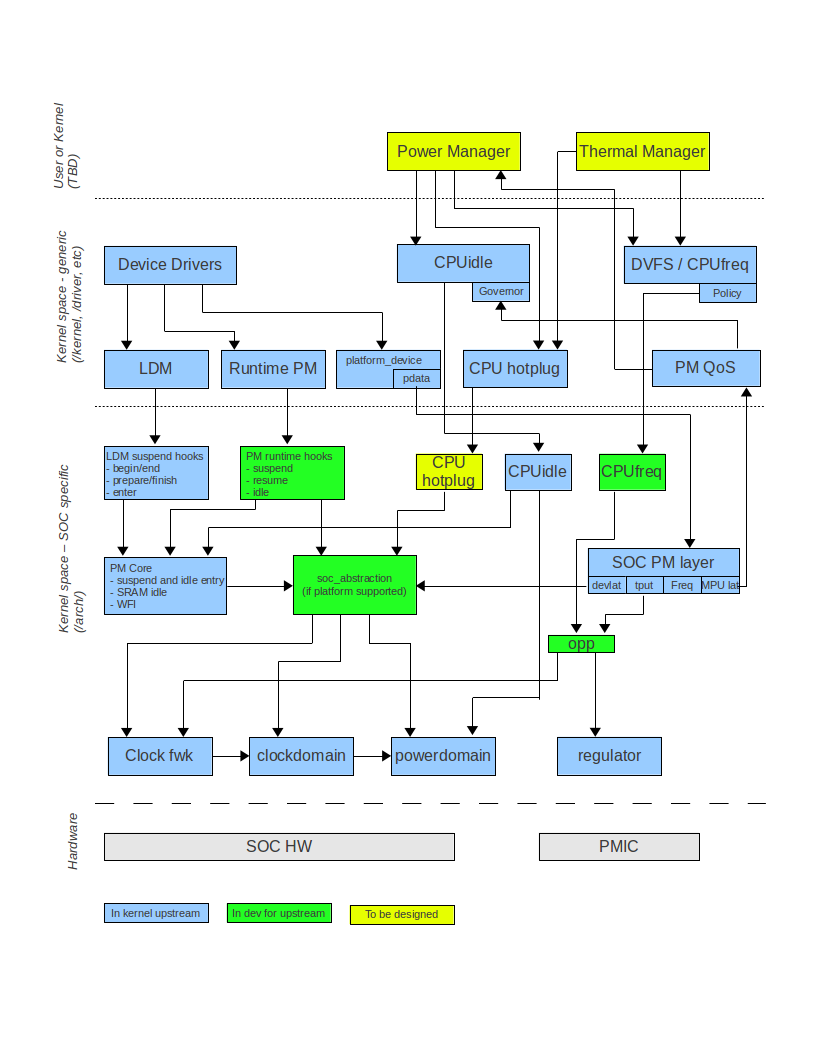


图 2-1 Linux系统电源管理架构[10]

Figure 2-1 Linux system power management architecture [10]

在硬件层之上是嵌入式Linux系统的内核。在3. X版本上它包含了实现电源管理所需要的时钟框架和电源驱动框架(Linux voltage and current regulator framework)，并在此基础之上提供了静态和动态电源管理两方面的功能。静态电源管理包括系统休眠与唤醒机制，动态电源管理主要包括基于Runtime PM框架的设备电源管理和一系列针对处理器核心的动态电源管理机制，如CPUidle[3]，CPUFreq[2]和CPU Hotplug[11]。

Linux内核中的电源管理架构设计遵从Linux内核的一般设计准则，即分为上层硬件平台无关和底层硬件平台相关这两层。通常硬件平台无关层通过sysfs这样的接口向用户空间提供控制接口，使得用户空间的应用程序可以控制系统的电源管理行为。而硬件平台相关层则由系统SoC硬件平台代码实现，用以向各个电源管理模块的硬件平台无关层提供控制特定SoC硬件平台的接口，如调节处理器核的运行频率和电压。本节下面的内容将从静态和动态电源管理两个方面介绍。

在Linux内核之上运行的是用户空间的电源管理模块，通常可以分为两类：一是负责控制系统电源状态的电源管理模块，比如控制系统的运行和休眠状态切换；二是负责调节系统的散热性能的热能管理模块[25]，比如在系统温度过高时降低处理器核的运行频率和电压以达到散热的目的。

**2.1.2** 静态电源管理

静态电源管理主要负责控制内核层面上的系统睡眠与唤醒。这里的睡眠指系统从正常工作状态切换到某一低功耗状态。在3.x版本Linux内核中，支持三种低功耗状态：待机(Standby)、睡眠(Suspend-to-RAM)和休眠(Hibernate-to-disk)[12]。这三种状态功耗依次降低，但从该状态返回所需要的延迟时间也依次增加。

待机状态下处理器核处于停止状态而大部分硬件设备也处于空闲状态。这种状态带来的功耗降低效果最少，但回到正常工作状态但延迟最短，通常在1秒内。

睡眠状态也被称为Suspend-to-RAM。在这种状态下，大部分硬件设备都处于电源关闭或者低功耗状态，处理器核的电源通常是关闭的。程序上下文保存在DRAM中，DRAM处于低功耗的自刷新状态。整个系统只保留用以唤醒系统的模块供电。这些模块将用以在特定条件下，比如电源键被按下，唤醒处理器核并从DRAM中恢复程序上下文。与待机状态相比，睡眠状态带来的功耗降低效果大很多，通常对于嵌入式Linux系统来说，整个系统的耗电在10mA以下。从睡眠状态恢复到正常状态的延迟与待机状态相比略微增加，但仍可以保持在1秒内。

休眠模式通常被称为Hibernate-to-disk。在该状态下，程序上下文保存在外部非易失性存储设备（比如硬盘）中。因此，整个系统可以完全关闭电源从而达到最低功耗状态。但同时返回正常工作状态但延迟也最长，可以达到睡眠状态的数十倍。

对于嵌入式设备，尤其是手持智能终端设备来说，时刻保持对用户对高度响应至关重要。因此，在这类系统里，Linux核心电源管理主要负责正常工作状态和睡眠状态之间的切换。

**2.1.3** 动态电源管理

如上文所述，嵌入式Linux系统的动态电源管理包括了一系列针对CPU和外设在运行过程中的电源管理机制。动态电源管理的适用基础在于系统和系统组件在运行过程中工作负荷的不持续性，即并不是在任何时候都有工作负荷。此外，工作负荷的分布情况是可以做一定程度的预测的，同时预测工作负荷不能有显著的功耗。下面将从系统在运行时和空闲时两个方面介绍这些机制。

(1) 运行时电源管理机制

在嵌入式系统中，半导体器件在运行时的功耗可以用公式2-1表示 [13]。

 (2-1)

其中，为活动因子，C表示对应的充放电电容，表示工作电压，表示工作频率。从公式2-1中可以看出，器件的功耗与其工作电压的平方和工作频率均成正比。因此，要降低功耗，就要降低工作电压和频率。当然，这通常需要在性能和功耗之间做到平衡。

Linux内核的CPUFreq框架就是基于以上原理，通过动态调整CPU的工作电压和频率达到降低CPU在运行时功耗的目的 [2] 。CPUFreq由Governor即频率调整策略和具体CPU的频率电压调整表两个主要部分组成。在调整CPU频率和电压时，CPUFreq依赖于内核提供的时钟(Clocks)和电源(Regulators)驱动框架。

CPUFreq框架的体系架构由CPUFreq governor、体系结构无关的CPUFreq核和底层的CPU-Specific Driver组成。CPU-Specific Driver位于CPUFreq框架的最底层，其功能是实现CPU的调频等一系列与之相关操作的驱动程序。CPUFreq核位于CPU-Specific Driver的上层，具有硬件不相关性，封装了变频技术以及根据工作负载选择合适的运行频率，为机制与策略提供了规范的通信接口。CPUFreq governor位于CPUFreq核的上层，是CPUFreq框架的核心，它根据策略在必要的时间选择出一个CPU适合的工作频率。目前的Linux内核(自2.6.0以后)中提供了Performance、Powersave、 Userspace、 Ondemand[16] 和 Conservative等５种调控器供用户选择使用。

* Performance: CPU工作在它所支持运行的最高频率上
* Powersave: CPU工作在它所支持运行的最低频率上
* Userspace: 用户应用程序通过它提供的接口按照需求调节CPU的运行频率
* Ondemand: Ondemand只能在内核态下运行，能够以更加小的时间间隔对系统的负载采集并分析
* Conservative: Conservative是在Ondemand算法的基础上改进而来的，它在降低频率时会从可供选择的频率中选出确保CPU运行在80%以上的负载频率。在这个前提下，Conservative选择CPU所支持运行的最低频率。大量测试数据表明Conservative能够在不影响系统性能的前提下实现更加高效的节能。

(2) 空闲时电源管理

空闲时电源管理主要分为CPU和外设两个方面。对于CPU的空闲电源管理，Linux内核提供了CPUidle [3] 和CPU Hotplug [11] 框架。而对于设备，主要使用Runtime PM机制[5]。在现代嵌入式Linux系统中，CPU大多按功耗程度的不同包含多个空闲状态。虽然在理论上希望CPU尽可能多地处于功耗最低的状态，但是实际上，由于各个状态进入和退出需要一定的代价[17]，即额外的功耗，需要根据实际情况（通常是可预测的空闲时间）选择合适的状态进入。要达到这一目标，就需要掌握CPU各个空闲状态的特性包括进入和退出的功耗或延时、进入或退出该状态的具体方法。CPUidle框架就是为了提供这些而设计的。在此基础上，CPUidle还包含了一个控制状态切换的策略，即CPUidle governor[3]。

随着处理器技术的不断发展，嵌入式领域越来越多的CPU开始包含双核、四核甚至八核来满足CPU的高负载计算。相对地，嵌入式Linux系统可以在系统负载较低的情况下关闭一个或多个CPU核来更进一步降低整个CPU的功耗。CPU Hotplug框架就是为实现这一目的而设计的。对应于CPU，设备的动态电源管理是通过Runtime PM [5] 框架实现的。考虑到SoC上的设备是有相关性的，特别是在空闲状态的管理上，存在依赖关系和共同的电源开关，比如一个SoC集成图形处理子系统部分的存储和时钟部分。因此在Runtime PM的基础上又引入了电源管理域 (PM domains) [27] 的概念。同时，引入PM QoS[27]的概念用于表示不同种类的设备或CPU对于电源管理状态的约束，用于让Runtime PM框架决定应该进入哪一个状态。

**2.2** 嵌入式Linux系统电源管理的需求和存在问题

**2.2.1** 电源管理需求

在静态电源管理上，嵌入式系统往往因为用户体验，需要智能地判断系统什么时候处于空闲状态从而可以进入睡眠而不影响用户对系统的正常使用。比如即使屏幕关闭，我们仍然期望智能手机等到后台的Wi-Fi数据网络下载任务完成后再进入睡眠。而在系统唤醒时，比如用户在系统睡眠状态下按下电源键，也需要尽快完成这一过程来提高用户体验。

在CPU的动态电源管理方面，需要根据系统的应用场景来选择合适的CPUFreq策略。比如根据智能手机的特点，选择更注重用户交互体验的interactive CPUFreq governor。对于CPU空闲状态的管理，需要更精确的预测出CPU可以空闲的时间，避免频繁进入和退出空闲状态带来的额外开销，另一方面也可以根据预测的时间选择适当开销的空闲状态，这一点也同样适用于外设硬件的动态电源管理。同时，在CPU或外设进入空闲状态后，如何减少不必要的唤醒也同样重要，空闲状态下无时钟中断机制就是通过移除空闲状态下的时钟中断来减少不必要的CPU唤醒的。

此外，随着嵌入式Linux系统上CPU处理器技术的不断提高和新的外设硬件种类的不断出现，对电源管理的需求也在不断更新。例如，CPU从单核发展多双核四核甚至多核、从SMP发展到HMP，每个CPU核支持多个工作电压和频率以适应不同的工作负载需求，这就需要CPU调度器能实现以下两个方面的平衡：一是尽可能地将任务集中到一个CPU核或CPU簇以便让更多的CPU核处于空闲状态；二是根据任务对CPU的负载要求调整CPU核的工作电压和频率，选择合适的CPU核来运行任务。

需要注意的是，这里所说的CPU的负载应该是和CPU的工作频率相关的。理论上同样一个任务在CPU频率提升一倍的情况下所需要的CPU时间是原先的一半。要实现上面的平衡，就需要调度器不仅仅只是完成任务的调度，更是具有节能意识的(Energy-Aware Scheduling，简称EAS)。

**2.2.2** 存在的问题

（1）目前电源管理系统中第一个问题在于任务的调度和CPU频率分别是由Linux内核的调度器和CPUFreq各自管理的，没有共享数据。

如2.1.3节所描述，CPUFreq是通过各种插件形式的Governor工作的。而目前常见的Governor都是通过基于对CPU忙碌时间的采样来决定下一步CPU频率的调整。这样的方法在某些情况下会造成不正确的频率调整。

例如，CPU的最小频率fmin=100MHz, 最大频率fmax=1000MHz, Governor的策略是下一步CPU的频率f=util \* fmax+fmin, 其中util为上一个采样周期内CPU忙碌时间的百分比。图2-2显示了一个在8个采样周期内的CPU频率调整过程。

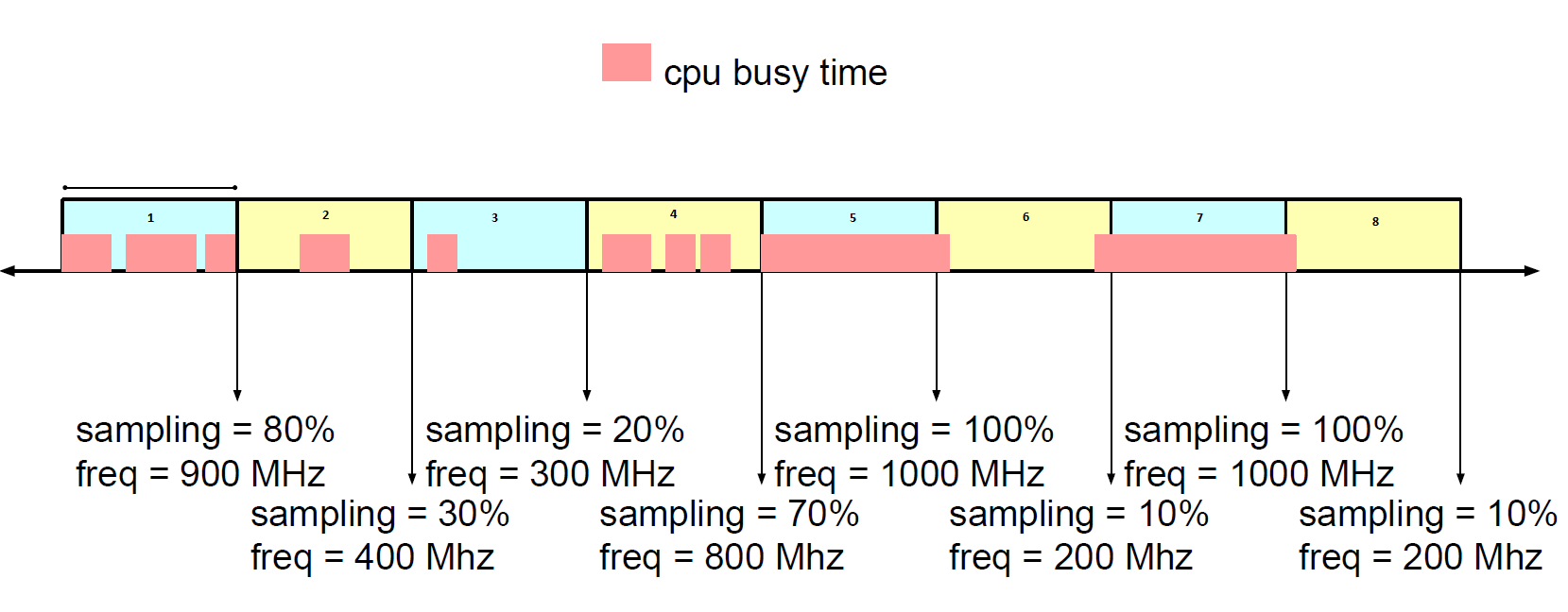


图 2-2 基于采样的CPU频率调整的问题

Figure 2-2 Problem with sampling based CPU frequency adjustment

从图中可以看到，在采样周期7和8内，CPU分别是以200MHz和1000MHz运行的，这是因为在采样周期6和7中，统计的CPU忙碌时间百分比分别为10%和100%。然而实际上，希望在周期7内，CPU以fmax运行，而在周期8内，CPU以200MHz运行。造成这一问题的原因在于采样周期与任务调度时机的信息不匹配。在周期6即将结束而周期7还未开始时，CPU上的可运行任务数量增加了。由于周期6即将结束，这些任务对CPU的要求无法体现在采样上，这就导致了周期7内CPU依然按照周期6内较低的CPU忙碌时间百分比来用较低的200MHz来运行，从而大大降低了系统的整体性能。同样的道理，虽然在周期8刚开始不久，CPU上的可运行任务就没有了，但CPUFreq依然按照上一个周期7内的采样结果让CPU以最高的1000MHz运行，浪费了系统性能并造成了功耗的增加。这样的问题在新的任务创建，任务在CPU核之间升级(migration)调度时尤为突出。

（2）第二个问题是动态电源管理中如何避免进入和退出低功耗状态带来的额外开销。

如上面所分析的，解决这个问题的关键在于选择正确的时机切换状态。对设备的动态电源管理来说，目前大部分的实现方法是基于固定的超时机制的，即在设备处于空闲状态后，判断经过一定时间后设备是否仍然处于空闲状态。如果是则通过Runtime PM框架将设备切换到低功耗状态，否则就保持运行状态。这种机制的优点在于实现简单而缺点则是灵活性差。因为这一定时间的超时是固定的，在这段时间内设备很有可能实际上处于空闲但不能把它处于低功耗模式。 一个改进的办法是预测设备可以处于空闲状态的时间，根据预测出的时间选择适当开销的空闲状态。

本文在第三章中就将提出针对上面两个问题的解决和改进的方法。除此之外，还需要持续改进静态电源管理中如何判断系统空闲状态、动态电源管理中CPU频率调整的策略以及减少CPU在空闲状态下不必要的唤醒等问题。

**2.3** 本文的研究路线

**2.3.1** 有节能意识的调度器

如上文中所分析，有节能意识的调度器需要根据任务对CPU的负载要求选择功耗成本最低的CPU或CPU核来运行。因此，要实现这样的调度器，首先要实现能准确反映CPU负载的机制。这在Linux目前的内核调度器中是通过负载追踪 (Load Tracking，简称LT) 来实现的。为此，本文将在第三章中提出一种对目前Linux内核中默认的CFS调度器[14]的负载追踪算法的改进方法。其次，为实现2.2.1节中所提出的需求，我们需要对调度器的任务放置(Task Placement)和负载均衡(Load Balancing)机制作出改进，以达到选择功耗成本最低的CPU核来运行任务的目的。最后，为实现与CPUFreq框架的协作，将CPU的负载信息反馈给CPUFreq governor，还需要实现调度器和CPUFreq框架之间的协同机制。此外，为满足嵌入式系统不同应用场景的需要，调度器设计中还包括了对调度器算法的可调整参数。用户可以根据实际需要调整这些参数并从实际测试结果中选出最合适的参数。

**2.3.2** 设备动态电源管理

对于设备的动态电源管理，本文以Atmel MXT1664电容触摸屏控制器为例，首先明确硬件上定义的各种运行模式并在此基础上确立动态电源管理的目标。然后，基于Runtime PM机制设计进入和退出低功耗模式的算法。同时基于一种设备空闲状态的预测机制实现对进入低功耗状态切换的控制。在实际运用中根据不同预测结果调整预测参数以提高预测准确度。

**2.4** 本章小结

本章首先从架构、静态和动态电源管理三方面综述了嵌入式Linux系统电源管理技术，然后提出了电源管理上新的需求和目前存在的问题。在此基础之上给出了本文的研究路线，为本课题的后续设计提供支持。

**3 嵌入式Linux系统电源管理管理方法的改进**

从前面第二章的分析可以看出，嵌入式Linux系统电源管理在CPU和设备两方面都存在新的需求和问题。在CPU方面，针对2.2.2节中的第一个问题，本章将提出一种有节能意识的调度器设计；而在设备方面，针对2.2.2节中的第二个问题，本章将提出一种基于Runtime PM的设备动态电源管理设计，从而对这两方面问题提出解决方案。

**3.1** 一种有节能意识的调度器设计

**3.1.1** 负载追踪的改进

1. 现有负载追踪技术的分析

Linux内核CPU调度器的设计难点之一就是需要将系统处理器资源以一种公平和能及时响应的方式分配给系统内各个进程，并且在这一过程中保证系统吞吐量的最大化和功耗的最小化，但这些目标往往是互相冲突的。内核调度器的设计随着Linux的发展经历了好几种版本，直到内核2.6.23上采用了CFS调度器之后才稳定了下来，然而调度器内部的组件设计的演进一直没有停止，负载追踪机制就是其中一个例子。

理想的CPU调度需要预测系统内每一个进程对它的资源要求数量以及要求的时间。当内核了解了这些需求之后才能决定如何优化调度这些进程。在能够预测这些需求的硬件面世之前，调度器必须能够实现一定程度上这样的预测方法，比如通过追踪基于过去一段时间的CPU负载情况，这就是负载追踪机制。

最初，CFS使用的是一种基于进程运行队列(run-queue)的负载追踪机制。每一个CPU至少有一个这样的run-queue。当有需要时，调度器会计算每个run-queue对CPU负载的贡献，但这只能做到整个run-queue层面，无法做到精确了解队列中每个进程的贡献。从内核3.8版本开始，CFS的负载追踪是一种基于entity的机制 (per-entity load tracking，简称PELT) [18] 。在PELT中，每一个进程或进程组就是一个entity。PELT的特点就在于将负责追踪建立在entity基础上，从而解决了之前基于run-queue的负载追踪机制无法知晓精准的负载来源和CPU负责相对稳定情况下容易产生大偏差的负载评估的问题。

PELT将时钟时间分为一系列连续的1ms片段，一个entity在其中一个片段*pi*对系统负载的贡献只考虑它在这段时间内处于可运行状态(包括实际运行和等待CPU时间)的部分。系统当前负载L可以用公式3-1表示为

 (3-1)

其中，*Li*表示在*pi*时间内的系统负载，*y*是衰减因子。在目前的内核代码实现中，*y*的取值使得当*y32*等于0.5。因此，从负载衰减的角度来说，32ms之前的系统负载值对当前系统负载评估的贡献大约是50%。

PELT能够让调度器在没有增加额外开销的情况下更清楚地了解每个进程对CPU负载的贡献。这可以给调度器的设计带来好处，其中一个明显的例子就是对负载均衡的改进。在这种情况下，调度器可以让每一个CPU的负载差不多相同，最大化多CPU架构系统的整体性能。

然而，PELT是针对桌面计算机，工作站和服务器这样的通用计算环境设计的，并没有针对有节能意识的任务调度和移动设备负载作特别的考虑。和这些环境相比，移动设备的CPU负载具有明显的突发性和不确定性的特征。在实际测试中，在默认的调整参数下，发现计算密集型任务需要花费75ms才能从0%缓慢提高到80%，而空闲任务需要100ms才能从100%缓慢下降到10%[19]。为了解决这些问题，更好地适用在移动产品如手机操作系统中，这里我们提出一种新的基于窗口的负载追踪机制(Window based Load Tracking，简称WinLT)。

1. WinLT设计思想

对于每一个调度任务，把它过去在CPU上运行的时间分为无数个相同大小的窗口并追踪它的N个最近的时间窗口。N的大小和窗口的时间长度都是可配置的。考虑到移动设备上负载的突发性和不确定性，N的取值不能太大，实际应用中，我们取N=5。这些时间窗口只包括那些任务处于实际运行状态的时间，不考虑被阻塞和没有得到CPU的时间。

这N个时间窗口的负载数据历史和每一个任务相关联。这些历史数据在以下任务调度事件发生时更新：

* 任务开始在CPU上运行；
* 任务在CPU上运行结束；
* 任务被从睡眠状态唤醒；
* 任务完成退出时；
* 任务从一个CPU转移到另外一个CPU运行；
* 调度器时钟中断发生时；
* 硬件中断处理完成时；
* CPU频率发生变化时。

每一个任务的CPU负载要求计算要依据不同的策略执行。不同的策略可以适用于不同的实际运用场景，比如移动设备、桌面计算和服务器。这些策略包括：

* 最大值(max)，即取N个窗口的负载值中的最大值；
* 平均值(avg)，N个窗口的负载值求平均；
* 最近值和平均值的最大值(max(recent, avg))，取最近一个窗口和N个窗口的平均值中的大者。

时间窗口的起止时间在各个CPU上是同步的，CPU负载*Load*的计算是归一化的。归一到CPU运行频率和多核CPU拓扑架构上，这可以用公式3-2表示。

 (3-2)

其中，*t*是窗口内的任务执行时间，*fcur*是任务运行时的CPU核当前运行时钟频率，*fmax*是系统中CPU核的最大可能运行时钟频率，*Smax*是多核CPU架构中最高能力核相对该CPU核的能力倍数。例如，在一个有2个Cortex A57核和2个Cortex A53核的四核CPU拓扑架构中，Cortex A57核的最高运行时钟频率是2GHz，Cortex A53核的最高运行时钟频率是1GHz。在该系统中，最高能力的Cortex A57核相对Cortex A53核的能力是2GHz / 1GHz = 2倍。因此，当一个任务在Cortex A53核上以1GHz运行10ms时，它的归一化CPU负载就是10ms \* (1GHz / 2GHz) \* (1 / 2) = 2.5ms。

1. WinLT核心算法

每一个任务的CPU负载计算是WinLT的核心算法，即在特定的调度时间发生时更新任的负载统计值。其算法可表述为：

* 判断当前时间对应的窗口是否是任务的一个新窗口；
* 如果当前窗口是活跃窗口，更新当前窗口的负载值；
* 如果不是新窗口，结束；
* 用当前1个新窗口的负载值更新任务的历史负载纪录；
* 如果当前时间超过一个新窗口，用剩余的新窗口负载值更新任务的历史负载纪录；
* 返回算法开始。

其中，用n个新窗口的负载值更新任务的负载历史纪录算法可以表述为：

* 如果该任务在这个窗口内运行时间为0，忽略这个窗口，返回；
* 将负载历史数组中的前N－n个窗口内的值右移n个元素;
* 将新纪录的n个窗口内的负载值加到数组头上；
* 根据所选择的统计策略，计算历史更新后的N个窗口负载统计值；
* 如果该任务在调度器的运行队列上，运行队列的累计可运行时间减去历史更新前的N个窗口负载统计值，加上历史更新后的统计值。

1. WinLT与PELT的比较

PELT和WinLT的特性对比如表3-1所示。

表3-1 PELT与WinLT特性比较

Table 3-1 Comparison between PELT and WinLT

|  |  |  |
| --- | --- | --- |
|  | **PELT** | **WinLT** |
| 负载追踪 | 负载计算根据几何级数求和。32ms之前的系统负载值对当前系统负载评估的贡献大约是50% | 负载计算根据一定的统计策略基于过去的N个时间窗口的负载值。不同的策略可以提高不同使用场景的负载值追踪精度。 |
| 阻塞任务负载追踪 | 在任务进入阻塞状态时，其负载作为运行队列负载统计的一部分而衰减。任务的平均负载在任务重新进入运行队列时适当衰减。 | 阻塞时间不计入运行队列对系统负载的贡献。 |
| 阻塞任务负载恢复 | 运行队列对系统负载的贡献始终包括阻塞任务的负载 | 任务从阻塞状态切换到可运行状态时，恢复其负载对系统负载的贡献。 |
| 对CPUFreq Governor的负载报告的影响效果 | 负载报告中的负载迟缓提升导致频率的缓慢提升。 | 可实现移动设备使用场景中的快速负载提升。大多数情况下不需要额外的调度器调整策略。更多的突发性负载变化报告，可能需要仔细选择窗口统计策略。 |

在实际测试中，相比PELT需要约100ms才能在计算密集型任务情况下将CPU负载提升到100%，WinLT只需要约10ms就可以提升至100%CPU负载。而在一个在Chrome浏览器中加载engadget.com网站并向下拉动滚动条的测试中，PELT需要约120ms提升CPU负载而WinLT只需要约40ms[19]。

**3.1.2** 任务放置的改进

1. 任务放置分析

在多核多CPU架构下，调度器的任务放置在大多数情况下是一个功耗与性能之间的妥协。在低系统负荷情况下，如果任务可以集中到一个CPU上，从而使得其它CPU可以处于空闲状态，这可以降低系统内CPU的整体功耗。然而目前CFS调度器默认的配置是试图将任务平均分散到多个CPU的多个核上，这样可以获得更好的吞吐量，从而提高系统性能。

为平衡这两方面的需求，本文的设计目标是针对那些对CPU负载要求低的任务集中放置到功耗成本最低的CPU上，而对于负载要求高的任务分散放置到系统内的各个CPU上以获取性能的最大化。当有了从WinLT提供的基于每个任务的CPU负载需求信息后，加上各个CPU核支持的运行频率以及运行频率的mw/MIPS信息，就可以对CFS调度器设计中的任务放置进行改进。

1. 任务放置时机

任务放置的主要目的是为所要调度的任务选择最合适的CPU。在本文的设计中，它在以下时刻进行：

* 一个任务被唤醒；
* 调度时钟中断发生；
* 当一个新的任务产生；
* 进行负载均衡(Load balancer)时。负载平衡将在3.1.3节中描述。

1. 核心算法

任务放置的核心算法流程如图3-1所示。

* 找到一个中断负载低于高中断负载阈值的合适CPU。加入新的任务不会导致其负载超过阈值进而导致新的任务被放置到其他CPU簇内的CPU上(inter-cluster migration)。
* 如果可以找到这样一个CPU，判断加入的任务是否会导致超过升级阈值(up-migrate threshold)。 这是通过检查任务的WinLT纪录对CPU使用率的要求和CPU的最高运行时钟频率判断的。如果超过up-migrate threshold，那么这个CPU将被忽略。
* 在所有符合上面要求的CPU中，选择那个功耗成本最低的CPU来运行任务。功耗成本是先依据计算CPU的总负载对于频率的要求，然后从CPU的power table里根据以这个频率运行得出来的。Power table里存储的是对应每一个CPU频率的功率。
* 如果找不到符合上面所有条件的CPU，选择系统中能力值最高的CPU，即拥有最高运行时钟频率的CPU。

其中，我们针对任务的特定需求设计了两个标志：

* sched\_boost

当sched\_boost标志置位时，CPU选择逻辑将忽略功耗成本。设计这个标志的目的是为了找出所有CPU中负载最小的那个。

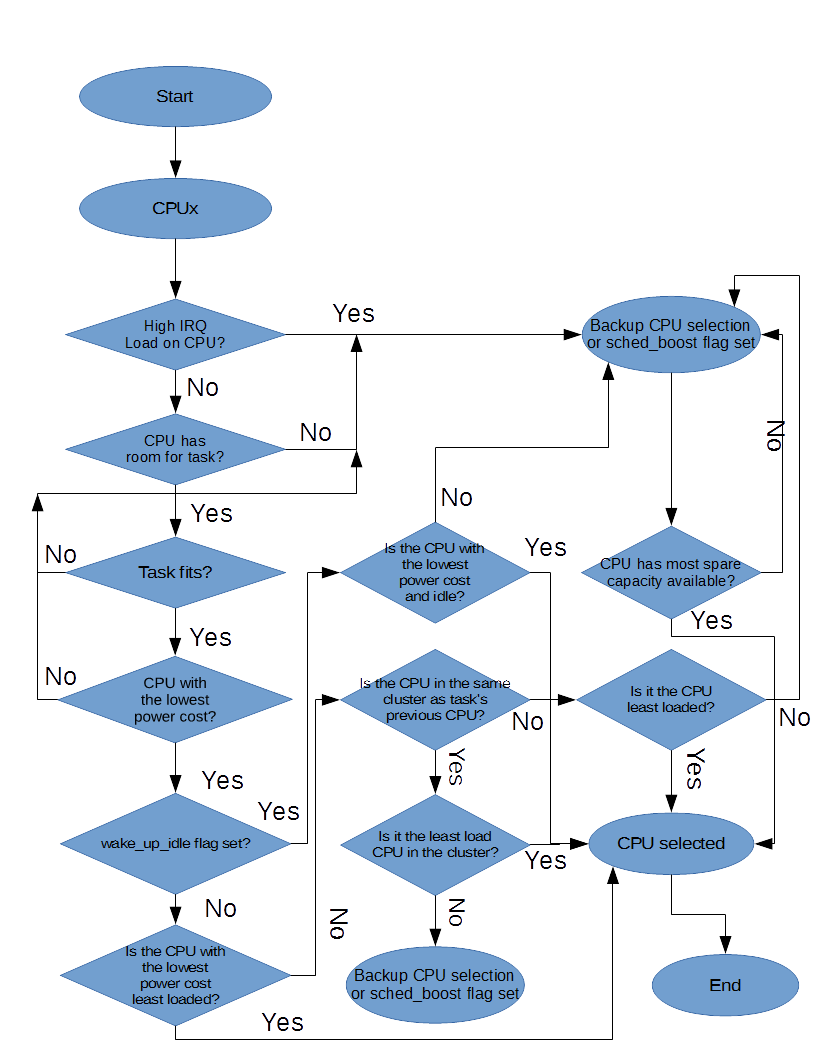


图 3-1 任务放置算法流程图

Figure 3-1 Task Placement flow

其中，我们针对任务的特定需求设计了两个标志：

* sched\_boost

当sched\_boost标志置位时，CPU选择逻辑将忽略功耗成本。设计这个标志的目的是为了找出所有CPU中负载最小的那个。

* wake\_up\_idle

wake\_up\_idle标志置位的任务需要唤醒一个处于空闲状态的CPU，除非所有的CPU都不处于空闲状态。

**3.1.3** 负载均衡的改进

负载均衡同时在CPU簇水平和CPU核水平上运行。它的基本设计思想是，当在CPU簇水平上运行时，将其他簇上的任务转移到正在进行负载均衡的CPU簇上；当在CPU核水平上运行时，将同一个簇内其他CPU核上当任务转移到正在进行负载均衡到CPU核上。

1. 负载均衡算法

负载均衡算法的步骤可表述为：

* 尝试找出最忙碌的CPU簇(在CPU簇水平上运行时)或者CPU核(在CPU核水平上运行时)。这一过程依赖于负载追踪算法的统计值。
* 对于最忙碌的CPU簇，找出该簇内最忙碌的CPU核。
* 找到最忙碌的CPU核之后，遍历该CPU核上的所有任务，找出那些适合转移到进行负载均衡的CPU上的任务并进行转移。

在这个改进的调度器设计中，衡量“最忙碌”的标准是CPU上累计的任务可运行时间，包括实际运行和等待的时间。这体现了任务对于CPU时间的实际需求。相比现有的负载均衡算法，用的是基于任务优先级和cgroup份额的负载衡量。

当一个CPU切换到空闲状态时，空闲均衡算法会确保空闲CPU更多地帮助处于忙碌的CPU以避免后者因过于忙碌而触发任务调度升级到更高能力的CPU从而达到降低功耗的目的。负载均衡算法支持溢出阈值。例如在一个2个Cortex-A57 和2个Coretx-A53架构中，当 Cortex-A53簇的负载没有达到溢出阈值时，任务不会被均衡到Cortex-A57簇上。对于处于tickless idle模式的CPU，它们的负载均衡会用另外一个合适的CPU来代替。

1. 实时(Real Time,简称RT)任务调度

在现有的RT任务调度中，任务在唤醒后总是在睡眠之前的运行的CPU上继续运行，除非以下例外：

* 那个CPU正在运行另外一个绑定在该CPU上的RT任务。
* 那个CPU正在运行另外一个优先级更高的任务，并且这个唤醒的RT任务并没有绑定在该CPU上。

当这些条件满足时，调度器会查找那些运行最低优先级任务的CPU并从中挑选和睡眠之前运行的CPU拓扑最接近的一个来运行该RT任务。在改进的这个调度器中，我们将不考虑总是使用睡眠之前CPU这一条件。相应地，调度器将找出那些运行最低优先级任务的CPU，并从中找出负载最小的CPU来运行RT任务。

**3.1.4** 与CPUFreq框架的协同

与CPU调度器类似地，CPUFreq governor (Ondemand)也统计系统内的负载来设置每个CPU的运行频率。参见2.1.3节的分析，CPUFreq通过设置可以满足系统内任务运行要求的最低频率且不影响系统整体性能，从而达到降低系统功耗的目的。然而，在目前的Linux内核中设计中，CPUFreq和CPU调度器是两个互相独立的设计，这就导致下面一系列的问题[26]：

1. CPUFreq加入了大量的代码通过间接的方法来评估实际的CPU负载，包括使用试探法来避免预测错误。然而，这些信息在调度器中已经获得了。
2. 调度器可以精确地获得每个进程对CPU负载的贡献，但CPUFreq没有这样的能力。当出现任务升级或唤醒情况时，CPU调度器可以提前计算出CPU负载变化的情况而CPUFreq只能注意到负载的增加并且在它实际发生后做出反应。
3. 为了确保所有任务调度的公平性，调度器记录了每个任务的执行时间。但是，因为调度器不知晓CPU频率的变化，当一个任务运行在一个频率降低的CPU上时，相比运行在一个频率更高的CPU上的相同任务，它的执行时间就会变长，这就导致公平性无法保证。

为了解决这些问题，需要调度器和CPUFreq更紧密的协同工作，即在需要对CPU频率做出调整时通知CPUFreq governor进行调整以确保CPU总是以最合适对频率运行。这样做的好处在于CPUFreq可以获得积极主动调整时钟频率的能力而不再只是单纯做出反应。同时，和调度器的其他活动，比如负载均衡也可以协调得更好[26]。因此，在本文所阐述的设计中，包括了在以下事件发生时对CPU频率是否需要调整进行检查的机制：

* 唤醒一个任务
* 任务被升级调度到另外一个CPU
* 进行负载均衡

为避免频繁调整CPU频率带来的开销影响整体功耗，在本设计中调度器需要定义负载增加时所需CPU频率升高阈值sysctl\_sched\_freq\_inc\_notify和负载减小时CPU频率降低阈值sysctl\_sched\_freq\_dec\_notify。这两个阈值可以通过sysctl接口进行调整。

当进行CPU频率检查时，先根据上一次查询CPU负载时的值计算出当前的等价CPU频率，再根据最近一次负载更新时的统计值计算出未来所需要的等价CPU频率。如果两者相差大于预先定义好的CPU频率调整阈值，就通知CPUFreq governor并指明可能需要调整频率的CPU序号。在CPUFreq governor中，当收到来自调度器的通知后，通过调度器提供的sched\_get\_cpus\_busy()接口得到当前CPU基于WinLT并归一化到CPU频率的负载值并根据这个负载重新计算新的CPU频率。

**3.1.5** 可调整参数

除了上述改进之外，本设计还提供了一系列可调整参数用来在运行时调整调度器的工作方式和策略。sched\_ravg\_window描述了WinLT中每一个时间窗口的长度。它的取值范围通常在10ms到1s，默认值为10ms。sched\_ravg\_window的大小决定了WinLT算法中统计CPU负载的最小单位时间，它的值需要根据系统实际使用场景来调整。在像智能手机这样的应用场景中，如果取值太大，则不能反映出用户在不同时间使用过程中CPU负载的突发性和不确定性；如果取值太小，则会带来统计负载的额外开销。sched\_ravg\_window的定义如表3-2所示。

表3-2 sched\_ravg\_window可调整参数

Table 3-2 sched\_ravg\_window tunable parameter

|  |  |
| --- | --- |
| 名称 | sched\_ravg\_window |
| 位置 | 内核命令行参数 |
| 默认值 | 10ms |

RAVG\_HIST\_SIZE描述了WinLT中统计CPU负载所用的最近时间窗口的个数。同样地，它的大小也需要根据实际使用场景来调整。如果取值太大，不能实时反映出CPU负载情况的变化的突发性；如果取值太小，会带来统计的额外开销。RAVG\_HIST\_SIZE的定义如表3-3所示。

表3-3 RAVG\_HIST\_SIZE\_MAX可调整参数

Table 3-3 RAVG\_HIST\_SIZE\_MAX tunable parameter

|  |  |
| --- | --- |
| 名称 | RAVG\_HIST\_SIZE\_MAX |
| 位置 | 预编译宏 |
| 默认值 | 5 |

sched\_window\_stats\_policy决定如何根据任务历史窗口的负载值计算总体任务负载值的策略。与之前的sched\_ravg\_window和RAVG\_HIST\_SIZE一样，计算总体任务负载的策略也需要依据实际使用场景来调整，以使得计算出的历史负载能最准确的反映过去RAVG\_HIST\_SIZE个时间窗口内CPU负载的情况。sched\_window\_stats\_policy的定义如表3-4所示。

表3-4 sched\_window\_stats\_policy可调整参数

Table 3-4 sched\_window\_stats\_policy tunable parameter

|  |  |
| --- | --- |
| 名称 | sched\_window\_stats\_policy |
| 位置 | /proc/sys/kernel/sched\_window\_stats\_policy |
| 默认值 | WINDOW\_STATS\_MAX\_RECENT\_AVG |

sched\_init\_task\_load定义了对于一个CPU上新创建的没有历史记录的任务的初始负载百分比。由于没有历史记录，WinLT算法无法得出一个负载统计值给它。它的取值范围是0到100。它的值会影响该任务所在的CPU的历史负载统计值。sched\_init\_task\_load的定义如表3-5所示。

表3-5 sched\_init\_task\_load可调整参数

Table 3-5 sched\_init\_task\_load tunable parameter

|  |  |
| --- | --- |
| 名称 | sched\_init\_task\_load |
| 位置 | /proc/sys/kernel/sched\_init\_task\_load |
| 默认值 | 0 |

sched\_account\_wait\_time定义了是否将任务的等待时间计入其对CPU时间要求的标志。0表示不计入，其他值表示计入。sched\_account\_wait\_time的定义如表3-6所示。

表3-6 sched\_account\_wait\_time可调整参数

Table 3-6 sched\_account\_wait\_time tunable parameter

|  |  |
| --- | --- |
| 名称 | sched\_account\_wait\_time |
| 位置 | /proc/sys/kernel/sched\_account\_wait\_time |
| 默认值 | 1 |

如果一个任务对CPU的使用率超过sched\_upmigrate所指定的百分比，那么该CPU将被认为能力不够。需要找一个更高能力（频率）的CPU来运行任务。它的取值范围为0到100。它的值越小，表示任务越容易触发up migrate。同样地，它的取值也要根据系统使用场景来调整，默认值是80。sched\_upmigrate的定义如表3-7所示。

表3-7 sched\_upmigrate可调整参数

Table 3-7 sched\_upmigrate tunable parameter

|  |  |
| --- | --- |
| 名称 | sched\_upmigrate |
| 位置 | /proc/sys/kernel/sched\_upmigrate |
| 默认值 | 80 |

sched\_spill\_nr\_run规定了一个CPU能够同时容纳的可运行任务的最大个数。它的值会影响任务在各CPU核之间的调度频率，需要根据CPU核硬件能力进行调整，默认值为10。sched\_spill\_nr\_run的定义如表3-8所示。

表3-8 sched\_spill\_nr\_run可调整参数

Table 3-8 sched\_spill\_nr\_run tunable parameter

|  |  |
| --- | --- |
| 名称 | sched\_spill\_nr\_run |
| 位置 | /proc/sys/kernel/sched\_spill\_nr\_run |
| 默认值 | 10 |

sched\_spill\_load参数规定了一个CPU能够承受的最大使用率百分比。当加入一个新的任务导致该CPU超过sched\_spill\_load所指定的值，将触发一个CPU簇间的升级调度。它的取值范围是0到100。它和sched\_spill\_nr\_run一起作用将影响任务在CPU核间的调度。默认值是100。sched\_spill\_load的定义如表3-9所示。

表3-9 sched\_spill\_load可调整参数

Table 3-9 sched\_spill\_load tunable parameter

|  |  |
| --- | --- |
| 名称 | sched\_spill\_load |
| 位置 | /proc/sys/kernel/sched\_spill\_load |
| 默认值 | 100 |

cpu\_high\_irqload用于描述判断CPU是否处于高中断负载的标准。如果该CPU的中断负载高于cpu\_high\_irqload，那么它将被视为高中断负载。cpu\_high\_irqload的定义如表3-10所示。

表3-10 cpu\_high\_irqload可调整参数

Table 3-10 cpu\_high\_irqload tunable parameter

|  |  |
| --- | --- |
| 名称 | cpu\_high\_irqload |
| 位置 | /proc/sys/kernel/cpu\_high\_irqload |
| 默认值 | 10 \* NSEC\_PER\_MSEC |
| 描述 | 判断CPU是否处于高中断负载的标准 |

* 1. 设备驱动的动态电源管理设计
     1. Runtime PM设计思想

从第二章的分析可知，在嵌入式Linux系统中，设备的动态电源管理是由Linux内核的Runtime PM框架实现的。它是按照Linux内核中的设备模型设计的。在该设备模型中，设备之间是一个抽象的总线树形的关系：设备挂载在某一类型的总线上，它们是总线控制器的子设备，而它们本身又是下一级总线设备的父设备。基于这一模型，Runtime PM为每一个设备定义了两个计数器：usage\_count和child\_count[5]。其中，usage\_count用来纪录当前设备的使用量计数；child\_count用来纪录当前设备的活跃(active)子设备的计数。

按照设备功耗的不同，分为三个电源管理状态：活跃(active)、空闲(idle)和休眠(suspended)[5]。Runtime PM在初始化时认为每一个设备都是处于suspended状态。当设备被用户开始使用后，设备的状态由suspended转换为active，同时相应地增加usage\_count和当前设备的父设备的child\_count。如果父设备此时处于suspended或idle状态，父设备也将转换为active状态，依此类推。当设备的所有用户使用结束，usage\_count减少到0并且同时child\_count也是0的话，设备将由active状态转换到idle状态。如果在idle状态下设备可以休眠，Runtime PM将会把设备切换到suspended状态。如果此时又出现用户使用当前设备或出现active状态的子设备，当前设备将切换到active状态。通过对这三个状态的动态切换达到降低设备功耗的目的。如图3-2所示。

在上述状态述切换过程中，设备驱动可以实现三个相应的回调函数(callback function)：runtime\_suepnd()，runtime\_resume()和 runtime\_idle()。函数的原型声明如下[5]：

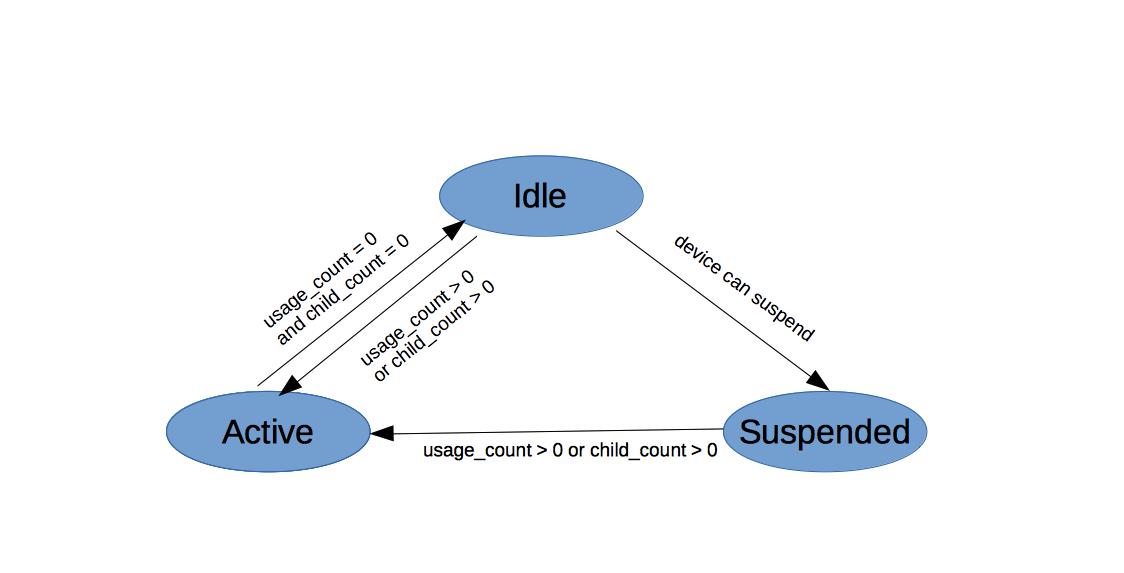


图 3-2 任务放置算法流程图

Figure 3-2 Task Placement flow

在上述状态述切换过程中，设备驱动可以实现三个相应的回调函数(callback function)：runtime\_suepnd()，runtime\_resume()和 runtime\_idle()。函数的原型声明如下[5]：

struct dev\_pm\_ops {

...

int (\*runtime\_suspend)(struct device \*dev);

int (\*runtime\_resume)(struct device \*dev);

int (\*runtime\_idle)(struct device \*dev);

...

};

其中，runtime\_suspend()用于设备在切换到suspended状态时将其切换到低功耗状态。这通常包括关闭不需要的电源和时钟输入和降低设备运行频率。runtime\_resume()用来恢复runtime\_suspend()中所做的操作，将设备切换到正常使用状态。runtime\_idle()的实现是可选的，当设备驱动认为设备可以切换到suspended状态时，该函数返回0，否则返回非0值。在本文的研究设计中，我们以Atmel MXT1664电容触摸屏控制器驱动为例，设计了基于Runtime PM框架的设备动态电源管理流程。

* + 1. Atmel MXT1664触摸屏控制器驱动的Runtime PM设计

1. MXT1664的硬件特性

在该嵌入式系统项目中，我们使用的是Atmel公司的MXT1664电容触摸屏控制器。MXT1664的硬件特性包括[20]：

* maXTouch™触摸屏解决方案
  + 支持双触摸屏。每个触摸屏支持12-bit多点触摸报告和最多16个触摸并发的实时坐标跟踪
  + 支持尺寸为8到14英寸的触摸屏
* 扫描速度
  + 单点触摸最高250Hz
  + 16点触摸最高100Hz
* 响应时间
  + idle状态下的首次触摸延迟小于15ms
* 接口
  + I2C总线接口，支持400kHz快速模式以及最高1.7MHz的高速模式
  + USB 2.0标准兼容接口, full speed 12Mbps
  + Microsoft Windows 8的HID-I2C接口
* 电源
  + 最低1.8V的数字供电
  + 2.7到3.3V的模拟供电

在该系统中，我们采用MXT1664的I2C总线接口，挂载在APQ8074AB的I2C总线上。数字电源采用1.8V VDDIO和3.3V DVDD，模拟电源为3.3V AVDD。VDDIO由系统PM8941的1.8V低压开关电源 (Low voltage switch mode power supply，简称SMPS) 提供；DVDD由电池电源VPH\_PWR通过低压降稳压器 (Low-Dropout Linear Regulators, 简称LDO) 提供，通过TP\_DVDDIN\_EN使能信号控制；AVDD由电池电源VPH\_PWR通过另一LDO提供，由TP\_AVDD\_EN使能信号控制。通过APQ8074AB的GPIO连接芯片的控制和状态管脚如复位和中断。其接口部分原理图如图3-3所示。

MXT1664有三种工作模式: Active, Idle和Deep Sleep [20] 。其中功耗最低的工作模式是Deep Sleep。各种模式下的功耗都分为三部分: 模拟电源功耗 (Analog Supply, AIdd), 数字电源 (Digital Supply, DIdd) 和X Drive Supply (XIdd) 。三种模式下的功耗如表3-11所示。

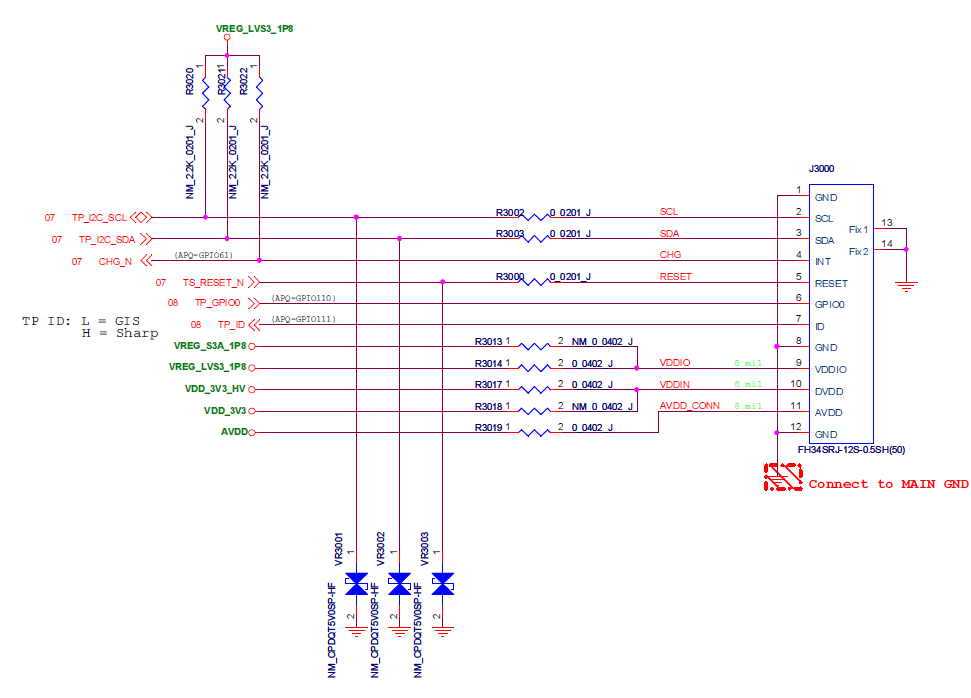


图 3-3 MXT1664接口原理图

Figure 3-3 MXT1664 interface schematics

MXT1664

表3-1011 MXT1664S三种模式下耗电情况（单位：mA）

Table 3-1011 MXT1664S power consumption under 3 modes (unit: mA)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Active | Idle | Deep Sleep |
| **AIdd** | **16.5** | **2.42** | **0.003** |
| **DIdd** | **8.58** | **1.13** | **0.33** |
| **XIdd** | **0.53** | **0.30** | **0.26** |
| 合计 | **25.61** | **3.85** | **0.593** |

1. 软件设计目标

从表中可以看出，Deep Sleep模式下耗电最低，Idle模式其次。但在Deep Sleep模式下MXT1664无法响应触摸。因此，动态电源管理的目标是让MXT1664在空闲模式下尽快进入Idle模式降低功耗而在用户不使用触摸屏时，比如LCD屏幕关闭后，让MXT1664进入Deep Sleep模式，从而将功耗降至最低。

由于用户的触摸动作是随机的，每两次触摸事件之间的设备空闲时间间隔不固定。考虑到MXT1664进入和退出Idle模式都需要一定都功耗开销，我们需要在只有预测设备的空闲时间足够长到可以补偿进入和退出Idle模式所花费的功耗时才进入Runtime suspended状态，否则无法达到降低功耗的目的。关于如何预测设备的空闲时间是否足够长，将在3.2.3节中详细描述。

1. 驱动架构

MXT1664驱动的动态电源管理相关部分架构如图3-4所示。其中绿色部分是需要实现的。这里只画出和动态电源管理相关部分。在底层硬件之上，Linux的Device Tree中的设备节点配置了MXT1664触摸屏控制器在系统中所用到的资源，包括I2C总线控制器，I2C设备地址，复位、中断GPIO pin脚和电源regulator。这些硬件资源的配置是和系统原理图的设计一致的。

在实际驱动层，驱动程序通过配置I2C总线资源实现读写硬件寄存器，通过控制复位GPIO pin脚操作和电源regulator来控制电源对硬件进行复位并实现上电初始化。硬件通过中断GPIO pin脚的状态触发中断处理程序从而使驱动程序获取硬件状态和触摸事件控制。

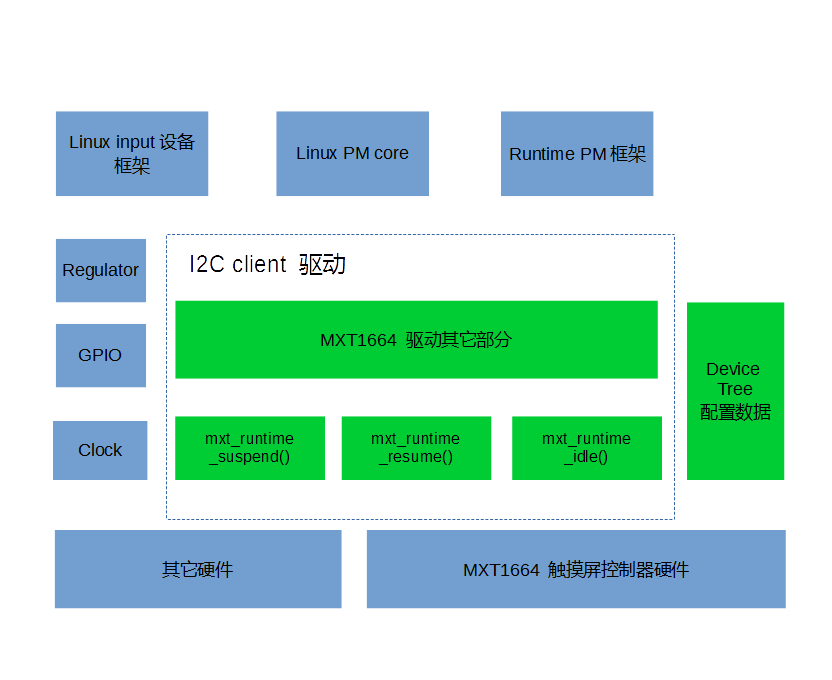


图 3-4 MXT1664驱动架构(动态电源管理相关)

Figure 3-4 MXT1664 driver architecture (dynamic power management)

在上层，驱动通过注册为Linux input设备与设备框架进行交互，将底层触摸事件的数据坐标转换成标准的input设备事件供用户空间的应用程序框架读取。应用程序通过Linux内核的系统调接口可以打开或关闭MXT1664的input设备节点，并从节点读取触摸事件的坐标。将坐标与屏幕上显示的应用程序用户接口(UI)元素布局坐标比较，就可以获取发生触摸事件的具体元素从而触发相应的用户触摸事件。

在设备电源管理方面，驱动程序实现了动态电源管理和静态电源管理。在动态电源管理方面，通过实现Runtime PM框架的runtime\_suspend、runtime\_resume和runtime\_idle回调函数来实现从Runtime Active到Idle再到Suspended状态的转换。整个驱动以及runtime\_suspend, runtime\_resume和runtime\_idle回调函数的流程将在4.2.3节中详细描述。在静态电源管理方面，通过注册Linux内核PM Core框架的suspend和resume回调函数，实现设备驱动的Suspend-to-RAM静态电源管理机制。根据Suspend-to-RAM机制的设计，suspend回调函数用于在整个系统进入suspend状态时将触摸屏切换到最低功耗状态；而resume回调函数则用于在系统从suspend状态切换回正常工作状态时将触摸屏恢复到suspend之前的状态。为达到系统suspend状态下触摸屏的最低功耗，suspend回调函数采用切断触摸屏供电的方式。

**3.2.3** 设备进入低功耗状态时机的确定

从第二章的分析可知，虽然在理论上动态电源管理希望设备在一结束运行状态就关闭，然而实际上在大多数应用场景里都不是这样。因为设备进入和退出低功耗模式都需要一定的开销。如果设备不能在空闲状态保持足够长的时间，设备在低功耗状态下省下的功耗将不能抵销频繁进入和退出低功耗状态所带来的开销。设备也无法从动态电源管理中得益。

在本文的设计中，我们采用Hwang等人[21]的研究确定MXT1644进入Runtime suspended状态所需要的空闲时间阈值并根据设备的历史空闲时间纪录预测当前空闲时间。只有当预测空闲时间大于等于空闲时间阈值时才让设备进入suspended状态。此外，为控制预测误差，引入看门狗机制来监视空闲时间预测值。要使动态电源管理能实际带来功耗上的降低，设备空闲时间的最小值*Ith*可以用公式3-3表示。其中，*E*为设备进入低功耗模式所需要的时间；*W*为设备退出低功耗模式所需的时间；*PEW*是进入和退出低功耗模式过程中所消耗的平均功率；*PR*是设备处于运行状态时的平均功率；*PS*是设备处于低功耗装下的平均功率。

在本文的设计中，*PR*和*PS*分别对应MXT1664的Active和Idle模式下的功耗。结合表3-10中的数据和MXT664的内阻即可得出相应模式下的平均功率。从MXT1664的数据书册可知，要使设备进入Idle模式，需要往MXT1664的GEN\_POWERCONFIG\_T7寄存器（地址为2字节长）通过I2C接口写入2个字节的配置数据。为了得到E，我们在mxt\_runtime\_suspend()里写入数据之前和之后加上trace信息分别纪录下时间戳，两者相减作为E的值。芯片数据手册中并没有指明MXT1664从Idle转换到Active模式所需的时间以及转换过程中所消耗的功率，我们假设*W=E*，*PEW=PR*。在确定了*Ith*之后，我们用公式3-4预测设备下一次空闲时间*In＋1*。

其中，*in*是纪录的上一次实际设备空闲时间；*In*是上一次按照公式3-4预测的设备空闲时间；*a*是预测因子，取值范围是大于等于0且小于等于1。*a*值越趋于1，表示此次的预测值越接近上一次记录的真实值，*a*值越趋于0，表示此次的预测值越接近上一次的预测值。

公式3-4中的预测算法本质上是一个指数平均数，由它的特性可以知道该算法存在以下问题：

* 1. 由于公式3-4中的*a*是常量，需要根据不同的实验结果不断调整以适应实际使用场景。
  2. 如果在预测过程中遇到一个较长的空闲时间段序列，那么该算法需要较长的时间来调整才能得出和实际值误差较小的结果，并且在这过程中会产生很多预测不足。同样的道理，如果遇到一个很短的空闲时间序列，公式3-4会产生很多过度预测。

这两个问题存在的原因在于公式3-4中的*a*是常量。当负载特征变化大时，不能及时作出调整，需要一个过程才能适应变化后的负载。针对这个问题，在本文的设计中，我们采用一个自适应的*a*值。当上一次预测值误差较大时，使得*a*值变大；当上一次预测值误差较小时，使得*a*值变小。由此，我们将*a*设计为*in*和*In*的函数。考虑到真实值和预测值的偏离程度可以表示为d = min (*in ,In* ) / max (*in ,In*), 且d的值为0到1，d的值趋于0表示预上一次测偏差大，趋于1表示偏差小。因此*a*可以设计为1 - min (*in ,In* ) / max (*in ,In*)。

作为初始，我们采用*a*=0.5，并根据实际运行结果修正*a*的取值以使预测值接近实际值。为获得*in*的值，我们在mxt\_io\_complete()函数结束前纪录实际设备处于空闲状态的开始时间并在MXT1664 Runtime idle回调函数入口处纪录当前时间，两者相减作为上一次设备的实际空闲时间*in*。当*In＋1*大于*Ith*时，回调函数返回0以使Runtime PM通过mxt\_runtime\_suspend()接口将设备切换到低功耗模式。

* 1. 本章小结

本章首先提出了一种在Linux CFS调度器基础上改进而来的有节能意识的调度器设计，介绍了该设计在任务追踪、任务放置、负载均衡、与CPUFreq框架协同和可调整参数五个方面的改进和设计。其次，提出了一种基于Runtime PM框架的设备驱动电源管理设计，重点介绍了在该设计中对于动态电源管理中确定低功耗状态切换时机的方法。通过本章的设计，对第二章中提出的问题提出了解决方案。

**4 嵌入式Linux系统电源管理方法的实现**

**4.1** 一种有节能意识的调度器的实现

在Linux 3.18.20版本内核默认的CFS调度器和Android “interactive” CPUFreq governor基础上实现了具有节能意识的调度器。整个实现包括十几个源代码文件，3200多行代码的修改。

本节将介绍具有节能意识的调度器实现中的主要数据结构和算法实现。首先是ravg结构体，它代表了一个任务对于CPU时间的要求（归一化到时钟频率和拓扑结构），主要成员变量定义如表4-1所示。

表4-1 ravg结构体主要成员变量

Table 4-1 main members in structure ravg

|  |  |  |
| --- | --- | --- |
| **成员变量** | **类型** | **描述** |
| mark\_start | u64 | 标记了调度事件（任务被唤醒、任务开始执行、任务被抢占等）在一个时间窗口内等开始时间 |
| sum | u32 | 代表任务在一个时间窗口内处于可运行状态的时间，包括运行时间和等待CPU的时间。归一化到时钟频率上。 |
| sum\_history[RAVG\_  HIST\_SIZE\_MAX] | u32 | 任务在之前RAVG\_HIST\_SIZE\_MAX个时间窗口内的sum值历史纪录。WinLT算法将根据这些纪录得出任务对CPU负载的要求 |
| demand | u32 | 依据不同策略从sum\_history数组中得出的CPU负载要求。 |
| curr\_window | u32 | 当前窗口中任务对CPU忙碌时间统计的贡献值 |
| prev\_window | u32 | 上一个窗口中任务对CPU忙碌时间统计的贡献值 |

为关联任务的负载信息到每一个任务，ravg结构体加入到task\_struct结构体中。WinLT算法的入口是walt\_update\_task\_ravg()函数。它主要完成两个任务：通过update\_task\_demand()函数更新任务的CPU时间需求和update\_cpu\_busy\_time()更新任务的CPU忙碌时间。而update\_task\_demand()通过update\_history()函数纪录更新任务在最近的几个窗口内的负载。它们的关系如图4-1所示。

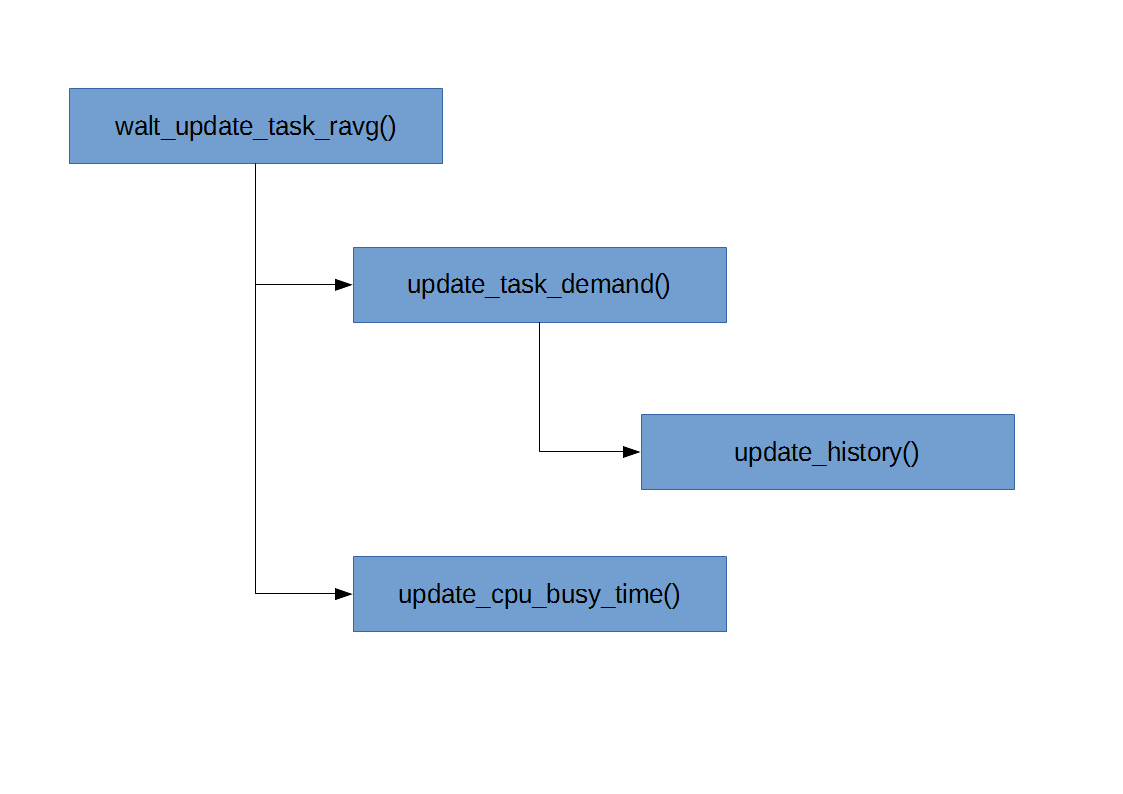


图 4-1 WinLT核心算法函数关系图

Figure 4-1 Function relationship in core WinLT algorithm

**4.1.1** update\_task\_demand()函数实现

WinLT的核心算法是update\_task\_demand()函数。它用来计算任务对CPU时间的需求和更新所记录的任务的CPU时间需求历史。这里会遇到图4-2中的三种情况：



图 4-2 update\_task\_demand()的三种情况

Figure 4-2 three cases in update\_task\_demand()

图中，ws表示一个时间窗口的开始，ms表示调度事件在一个窗口内的开始时间，wc表示当前时间。从图4-2中可以看到，第一种情况A：调度事件的发生在距离当前时间的一个窗口时间内，即ws < ms < wc。在这种情况下，任务在这个时间窗口内对CPU时间的请求应计算为wc - ms；第二种情况B：调度事件发生在两个时间窗口内，即ms < ws < wc。在这种情况下，任务在这两个时间窗口内的CPU时间请求分别为ws - ms和wc - ws；第三种情况，调度事件发生在超过两个窗口时间范围内。在这种情况下，首先更新调度事件发生的时间窗口内的CPU时间，即ws\_tmp - ms（ws\_tmp为ms之后的第一个窗口起始时间），然后计算出从ws到ms之间内完整的时间窗口个数n，更新在这n个窗口内的CPU时间请求，即n \* 窗口长度，最后更新第n+1个窗口内的时间，即wc - ws。函数代码实现见附录I。

**4.1.2** update\_history()函数实现

update\_task\_demand()算法的核心是update\_history()函数，用于纪录更新任务在最近的几个窗口内的负载。由于WinLT只考虑处于实际运行状态的任务， update\_history()首先判断这个任务在更新周期内是否有真正运行。如果没有，则直接返回，否则将此次统计的实际任务运行时间加到任务运行时间历史记录数组头上，并将老的记录向后移动相应个位置。接下来，update\_history()将根据指定的任务负载统计策略计算任务对CPU时间的要求并将它记录在任务结构struct task里作为未来调度器调度任务的依据。针对处于run queue的任务，update\_history()还将更新其累计的运行时间。它的流程实现如图4-2所示，完整代码见附录II。

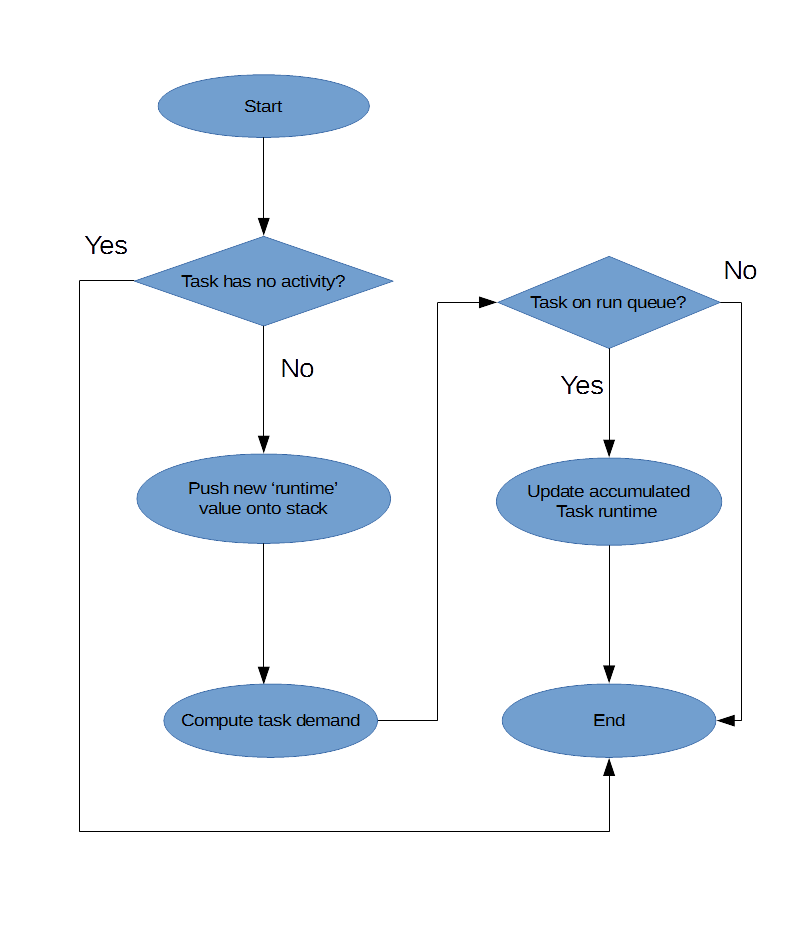


图 4-3 update\_history()流程图

Figure 4-3 update\_history() flow

**4.1.3** update\_cpu\_busy\_time()函数实现

update\_cpu\_busy\_time()函数是用来统计和记录任务在最近一次调度事件发生后的这一段时间内的CPU忙碌时间的。它的算法可描述如下。

首先，通过比较4.1.1节中的ms和ws的值判断从调度事件发生后到窗口开始是不是跨越了至少一个新的窗口。如果是，则计算出这过程中的完整窗口的个数。如果经历了新窗口并且当前任务不是处于空闲或者退出状态，更新ravg结构中的prev\_window和curr\_window的值。

接下来根据调度事件的类型判断是否需要更新CPU忙碌时间。对于空闲状态的任务来说，如果调度事件是选择下一个可运行的任务，则不需要更新CPU忙碌时间。对于其他的情况，如果调度事件是唤醒任务，也不需要更新CPU忙碌时间。而如果事件是放下前一个可运行任务或是中断和时钟周期更新，则需要更新忙碌时间。

对于不需要更新忙碌时间的情况，如果没有经历一个新窗口，则整个流程提前返回。否则，当当前任务是CPU正在运行的任务时，还需要更新该任务所在运行队列的累积可运行时间。对于需要更新CPU忙碌时间并且又没有经历新窗口的情况，需要在当前时间窗口内更新CPU忙碌时间。对于没有处于中断或是空闲或是等待IO完成的任务来说，新增加的忙碌时间就是当前时间与ms值的差值。否则，忙碌时间应为中断时间。将新增忙碌时间累加到当前运行队列的累积可运行时间上。对于不处于空闲或者退出状态的任务，还需要将新增忙碌时间累加到ravg结构的curr\_window值上，这样整个流程可以提前结束。

如果跨越了一个新的时间窗口，而且当前任务不是CPU正在运行的任务，需要分别更新上一个和当前时间窗口内的忙碌时间。对于没有跨过一个完整的窗口的情况，上一个时间窗口的忙碌时间是ws与ms值的差值，否则，就取一个完整时间窗口的长度。当前时间窗口内的忙碌时间是当前时间与ws值之间的差值。

如果当前任务是CPU正在运行的任务，需要更新针对当前窗口的忙碌时间。如果没有跨过一个完整的窗口，忙碌时间是ws与ms的差值；否则取一个完整时间窗口的长度。对于不处于空闲或退出状态的任务，将这个新增的忙碌时间记录到ravg结构的prev\_window上。另外，在这种情况下，新增忙碌时间还需要记录到运行队列的上一个累积可运行时间总和里。最后，当前窗口的忙碌时间取当前时间与窗口起始时间的差值并记录到运行队列的当前累积可运行时间总和。完整代码实现见附录III。

**4.2** 设备驱动的动态电源管理实现

**4.2.1** MXT1664驱动实现概述

在本设计中，我们以开源的Atmel maXTouch驱动[23]为蓝本，在高通APQ8074 AB SoC硬件平台（运行Linux内核3.18.20版本）上实现包含动态电源管理机制的MXT1664触摸屏控制器的设备驱动。MXT1664挂载在APQ8074AB的BLSP1的I2C总线上，由APQ8074的GPIO\_60和GPIO\_61分别连接MXT1664的RESET和CHG管脚作为复位和中断控制；由系统电源管理芯片PM8941提供MXT1664的VDDIO电源；由电池电源提供MXT1664所需的DVDD数字电路电源和AVDD模拟电路电源，分别由GPIO管脚TP\_VDDIN\_EN和TP\_AVDD\_EN作为开关来控制。具体电路设计原理图见图3-2。根据Linux设备驱动模型，该设备驱动分为两大部分：MXT1664的Device tree[24]配置数据和MXT1664的I2C Client驱动。

Device tree配置数据描述了MXT1664是如何连接在APQ8074硬件平台上的并且定义了MXT1664在该硬件平台上的特定配置选项，包括I2C Client地址和空闲时间预测因子a。MXT1664的I2C Client驱动给Linux内核提供了MXT1664的上电检测、复位、移除、电源管理等接口来操作控制该硬件。此外它通过注册Linux输入设备驱动来实现完整的一个作为输入设备的触摸屏控制器功能。Device tree配置数据将在下面的4.2.2节介绍，I2C Client驱动实现将在4.2.3节介绍。

**4.2.2** Device tree配置数据

Device tree是Linux内核启动过程中由Bootloader传递给内核用来描述该硬件平台的一系列参数。如前面4.2.1节中所述，MXT1664的Device tree配置参数描述了该触摸屏控制器在APQ8074硬件平台上的连接并定义了和硬件平台相关的特定配置选项。这些参数定义如下：

i2c@f9923000 {

status = "ok"

mxt1664@4b {

compatible = "atmel,maxtouch";

reg = <0x4b>;

interrupt-parent = <&msmgpio>;

interrupts = <61 0x2>;

atmel,reset-gpio = <&msmgpio 60 0x0>;

atmel,reg-dvdd = <&ts\_dvdd\_reg>;

atmel,reg-avdd = <&ts\_avdd\_reg>;

atmel,predict-const = <50> /\* set a = 0.50 \*/

};

/\* other I2C client definitions \*/

};

其中compatible = "atmel,maxtouch"用来匹配我们的I2C Client驱动；0x4b为MXT1664的I2C Client地址，用来给APQ8074的I2C总线控制器对MXT1664进行读写；interrupt-parent和interrupts用来描述中断管脚；"ateml,reset-gpio"用来指定复位管脚；"atmel,reg-dvdd"和"atmel,reg-avdd"分别用来指定DVDD电源和AVDD电源，这两个电源在Linux内核的静态电源管理下进入Suspend模式时被关闭以确保MXT1664的耗电降到最低。"atmel,predict-const"就是我们定义的空闲时间预测因子，其定义见在3.2.2节中详细描述。

**4.2.3** I2C Client驱动实现

如4.2.1节所述，MXT1664 I2C Client驱动是用来实现包括动态电源管理机制在内的完整驱动功能的。根据3.2节的设计，我们的动态电源管理将分为以下几个部分来实现。

(1) MXT1664驱动流程

MXT1664驱动基于Runtime PM的动态电源管理流程可以表述如下，整个流程如图4-3所示。

* 初始状态为active运行状态；
* 当检测到触摸动作时，从硬件寄存器读取坐标数据并上报给上层应用程序。继续保持active运行状态；
* 如果没有检测到触摸，进入Runtime PM的idle状态；
* 如果检测到触摸，读取硬件信息并切换到active状态；
* 如果没有检测到触摸且预测设备空闲时间超过suspended阈值，准备进入Runtime PM suspended状态；
* 如果可以进入suspend则继续suspend流程，否则回到idle状态；
* 如果进入suspended状态后检测到触摸，进入active状态，否则保持在suspended状态。

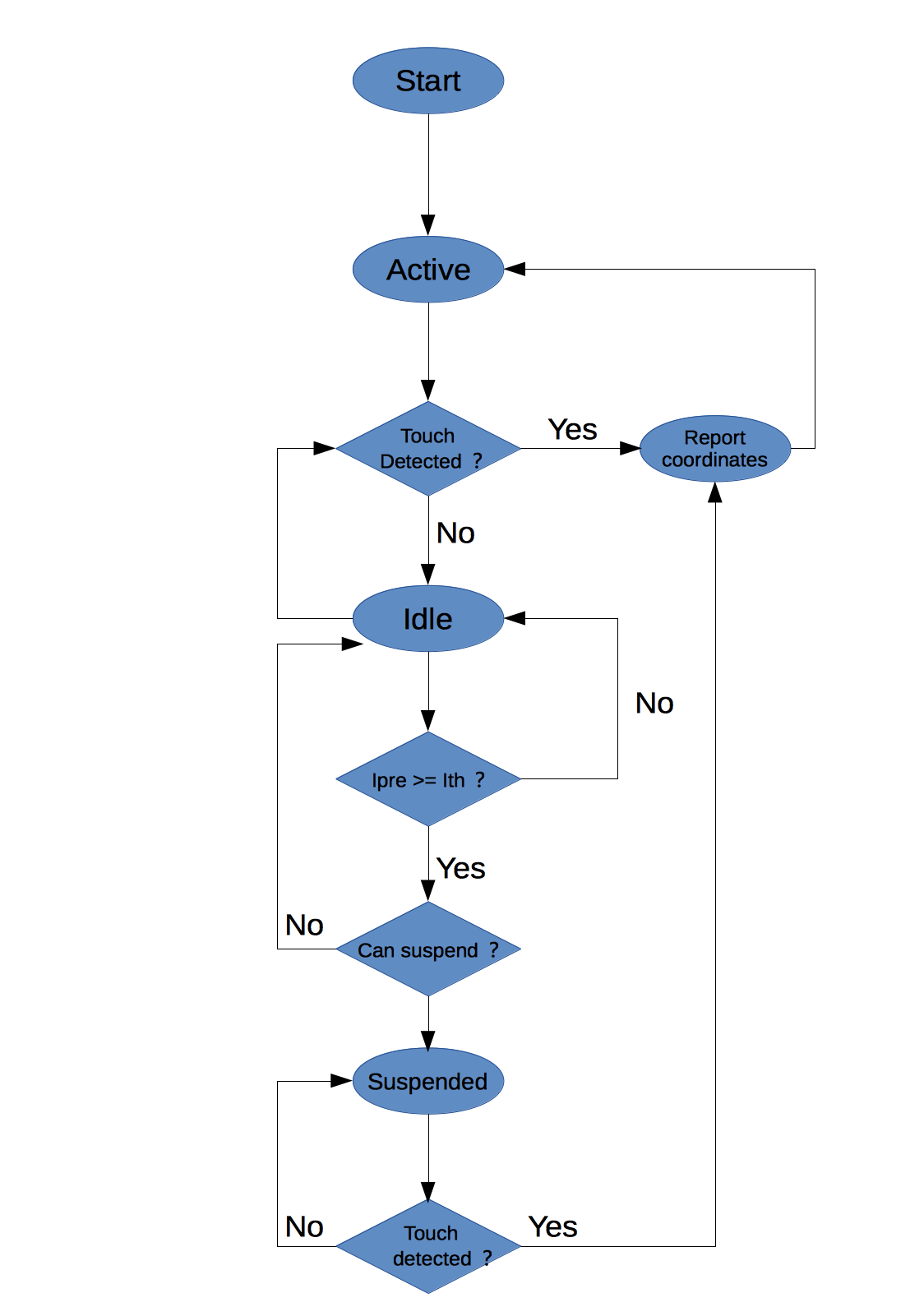


图 4-4 mxt\_runtime\_suspend()流程图

Figure 4-4 mxt\_runtime\_suspend() flow

其中，Ipre是当前预测的设备空闲时间，Ith是进入Runtime Suspended状态的空闲时间阈值。关于Ipre和Ith的计算，在3.2.3节中有具体描述。MXT1664驱动的runtime suspend回调函数mxt\_runtime\_suspend()的作用在于将处于Runtime PM idle状态的设备切换到硬件的idle状态以降低功耗。

(2) MXT1664 Runtime PM suspend回调函数

mxt\_runtime\_suspend()回调函数的作用是在MXT1664的使用量记数为0时将设备切换到硬件Idle状态以降低功耗。要完成这一任务，mxt\_runtime\_suspend()首先要获取状态同步锁sync\_lock。它的作用在于同步MXT1664在mxt\_runtime\_suspend(), mxt\_runtime\_resume()和mxt\_io\_complete()之间的状态切换，以避免竞争条件造成的状态异常。然后，判断当前是否用正在待处理的IO请求。如果有，则释放sync\_lock并返回-EBUSY错误代码给内核的PM runtime框架，表明当前设备正在处理IO，无法进入runtime suspend状态。PM runtime会在稍后再次尝试调用mxt\_runtime\_suspend()；如果没有，表明MXT1664目前的确可以进入runtime\_suspend状态。接下来就通过配置MXT1664的寄存器将它切换到硬件Idle模式，释放sync\_lock并返回0告诉PM runtime框架runtime\_suspend回调函数成功返回。它的流程如图4-5所示，完整代码参考附录IV。

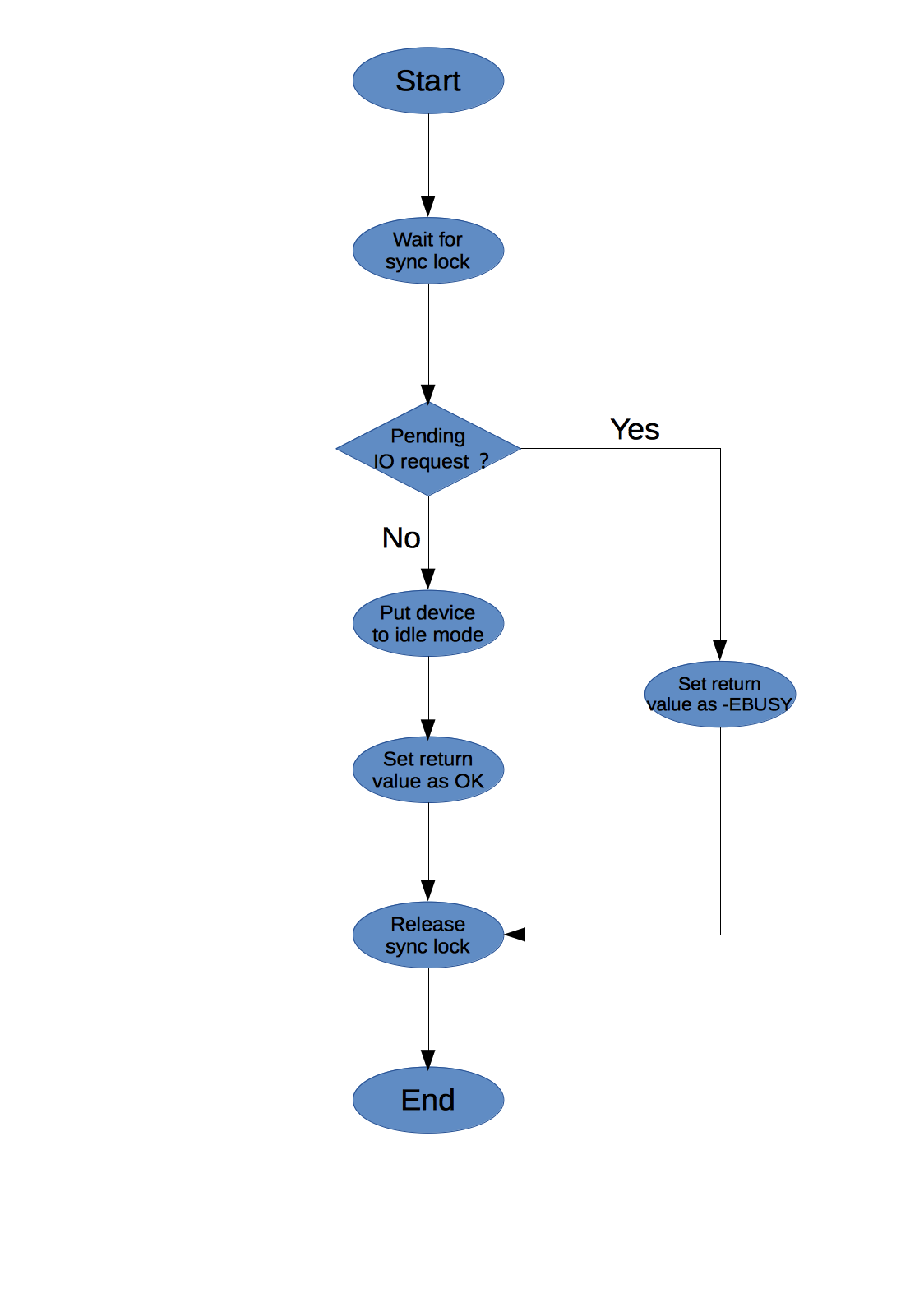


图 4-5 mxt\_runtime\_suspend()流程图

Figure 4-5 mxt\_runtime\_suspend() flow

(3) MXT1664 Runtime PM resume回调函数

mxt\_runtime\_resume()回调函数的作用在于当设备的使用量记数大于0时将设备切换回Active模式并处理还未处理的IO请求。为完成这一任务，mxt\_runtime\_suspend()需要先获取sync\_lock。然后配置MXT1664的寄存器使它切换到正常工作模式。由于在进入runtime\_suspend状态下，设备很可能是由于接收到新的触摸IO请求才会进入runtime\_resume状态。在这里需要判断是否有正在待处理的IO请求。如果没有，直接返回成功；否则，需要启动驱动中的IO处理线程再返回。它的流程如图4-6所示，完整代码参考附录V。

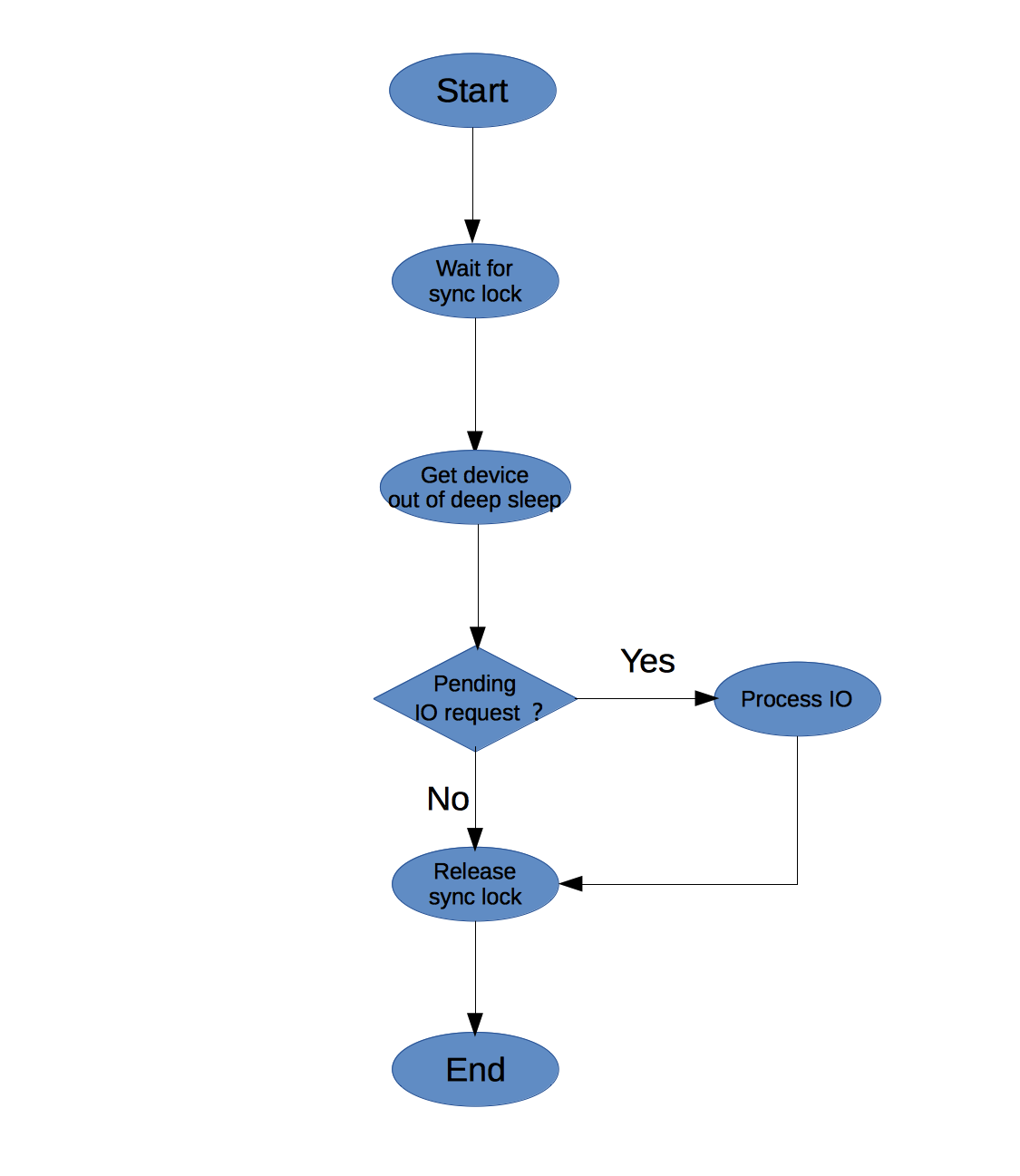


图 4-6 mxt\_runtime\_resume()流程图

Figure 4-6 mxt\_runtime\_resume() flow

(4) MXT1664 设备I/O完成

mxt\_io\_complete()回调函数用来完成设备I/O请求并记录设备开始空闲时间idle\_start。为完成这一任务，它需要先获取同步锁sync\_lock。然后判断当前还有待处理的IO请求。如果没有，说明MXT1664可以进入runtime idle状态，给设备的使用量记数减1；如果有，则需要启动IO处理线程。最后释放sync\_lock并返回。它的流程如图4-7所示，完整代码参考附录VI。

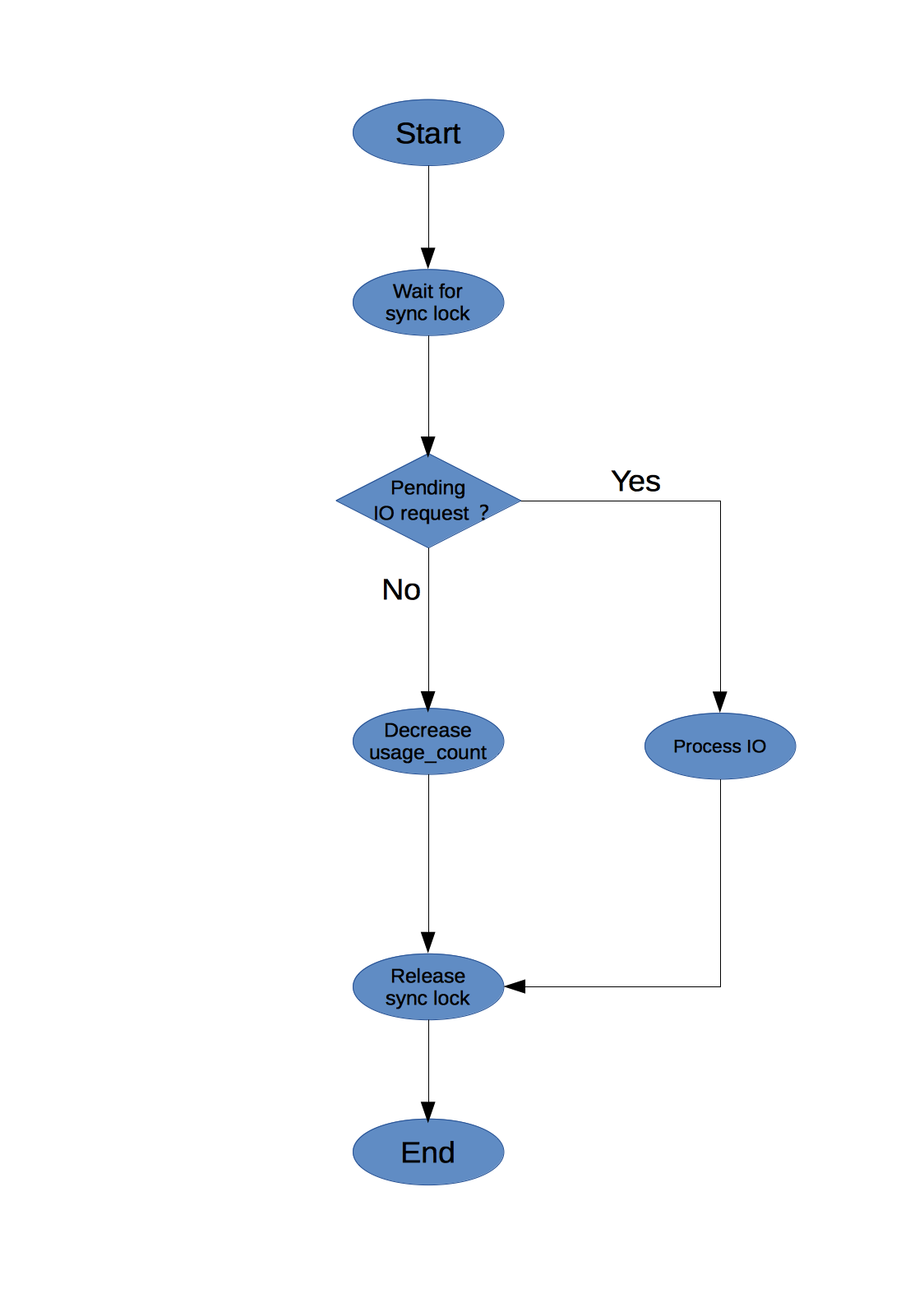


图 4-7 mxt\_io\_complete()流程图

Figure 4-7 mxt\_io\_complete() flow

(5) MXT1664 Runtime idle回调函数

根据3.2节的设计，MXT1664的mxt\_runtime\_idle()回调函数用来计算设备的实际空闲时间并根据这个时间来预测设备可能的空闲时间。如果预测时间大于Ith阈值，则允许设备进入Runtime suspend模式，否则返回-EAGAIN错误值以等待下次再进入Runtime idle模式重新计算预测时间。在我们的设计中，实际设备的空闲时间是从一次MXT1664设备I/O完成到进入Runtime idle回调函数为止所过去的时间。根据这一设计，mxt\_runtime\_idle()的流程如图4-8所示，完整代码参考附录VII。

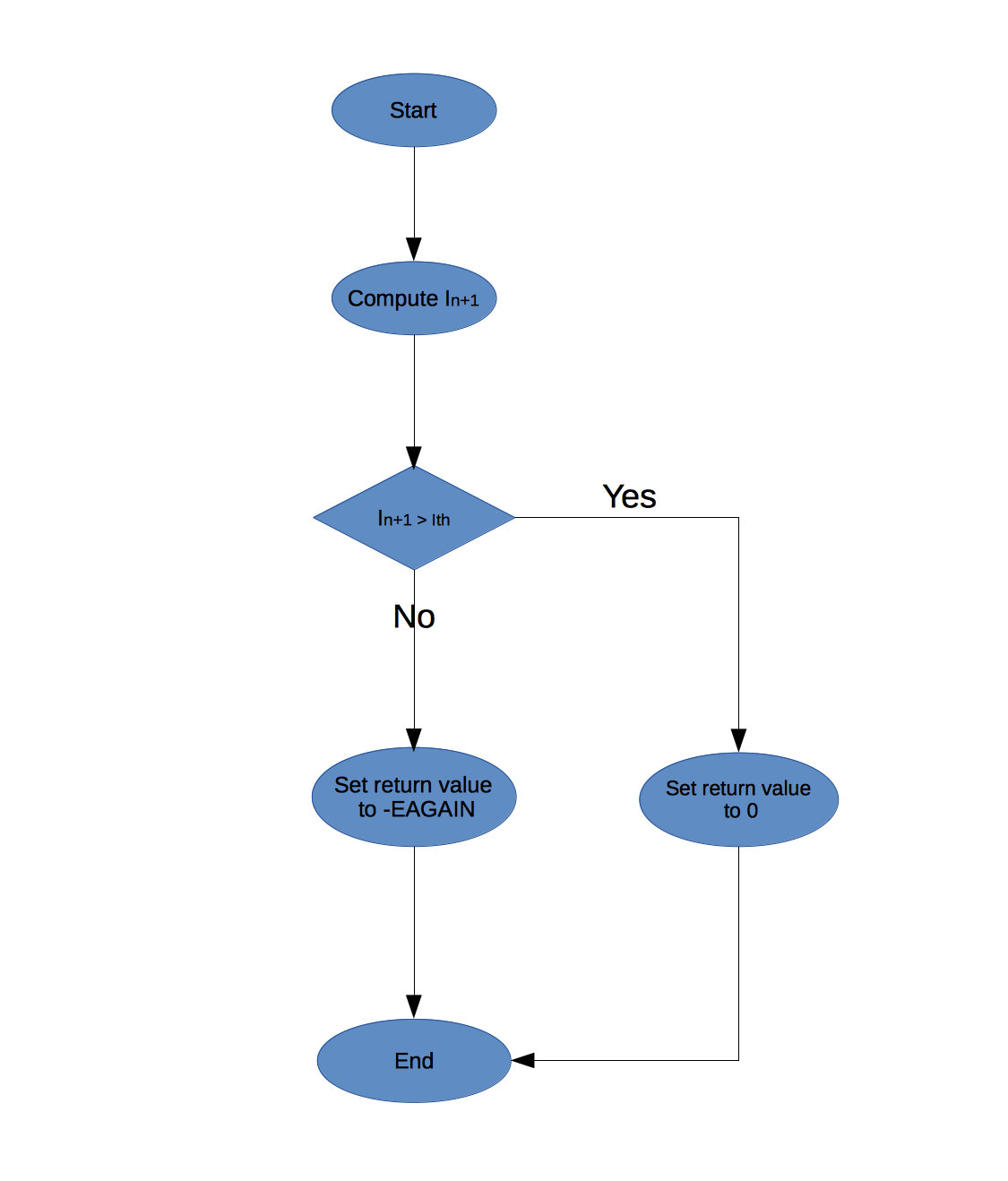


图 4-8 mxt\_runtime\_idle()流程图

Figure 4-8 mxt\_runtime\_idle() flow

**4.3** 本章小结

本章描述了有节能意识的调度器和基于Runtime PM框架的设备动态电源管理的实现并详细阐述了各数据结构和关键函数的实现细节。提供了节能意识调度器的测试和性能评估结果。

**5 系统功耗测试**

为了验证本文中提出的电源管理方法改进的设计的效果，检验设计是否达到要求，我们将在本章中对系统功耗进行对比测试。测试的重点是在轻量级工作负荷情况下，采用默认的CFS调度器和采用改进的具有节能意识的调度器两种情况下CPU处于不同频率的时间对比。一般来说，CPU处于高频率运行的时间越长，功耗就越大，反之，功耗就越小。这也是调度器改进的目标，即在日常轻量级工作负荷情况下减少CPU处于高频率运行状态的整体时间从而降低CPU的整体功耗。在设备的动态电源管理方面，测试主要在于在一定设备使用时间内统计采用预测设备空闲时间方法后对于设备实际处于runtime suspended状态的时间。

**5.1** 测试环境

为评估节能意识的调度器在降低系统功耗方面的成果，我们的硬件平台是高通MTP8996开发套件。MTP8996的CPU芯片组是高通MSM8996，包括4个ARMv8-A 64位架构14纳米制程的应用处理器Kryo核。其中2个Kryo核的最大时钟频率为2.15GHz，组成一个golden簇；另外2个Kryo核的最大时钟频率为1.6GHz，组成一个silver簇 [22] 。Golden簇主要面向高性能设计而silver簇主要面向低功耗。

软件方面操作系统为Android 6.0.1。采用了BBench和Powertop作为测试工具。BBench是一个由美国密歇根大学开发的自动化的网页渲染工具，可以让用户评估不同网页和网页技术的特性，评估浏览器性能。我们使用它的理由是它在渲染不同网页的过程中，可以产生一系列具有轻量级CPU负载的任务，具有一定的随机性，适合用于评估操作系统调度器性能。在测试中，我们使用的是Android版本的Firefox浏览器。

Powertop由Intel开发，是一个能够从多方面提供Linux系统功耗情况的工具。在我们使用的针对高通MSM硬件平台移植的Powertop 1.11版本上，它能够提供以下信息：

* CPU核处于各C-states的时间和百分比
* CPU核的P-states，即CPU核处于不同时钟频率的时间和百分比
* 各CPU核总计每秒钟从空闲状态唤醒的次数以及主要唤醒源统计

针对设备的动态电源管理设计验证，我们使用的硬件平台是搭载了Atmel MXT1664触摸屏控制器的HP Pro Slate 8平板电脑。该硬件平台以高通APQ8074应用处理器为核心。它的液晶显示屏尺寸达到了7.9英寸，分辨率达到了2048x1536。

**5.2** 测试方法和结果

**5.2.1** 有节能意识的调度器测试

我们的测试方法是，在Android系统启动完成后，启动Firefox浏览器进入BBench测试并设置网页库为默认的sites.txt。首先用单次测试测出完成一次测试在MTP8996上所需的时间t0，然后设置循环测试次数为5次。在Android shell命令行上用命令powertop -d -t <time>启动Powertop并同时开始BBench测试，time的值为5倍的t0 。做5次测试后计算出CPU核在各个状态时间百分比的平均值。作为对比，在没有调度器改进的情况下我们同样测试5次。测试结果的C-states和P-states平均用量如表5-1和表5-2所示。

表5-1 C-states平均用量对比（单位：百分比）

Table 5-1 C-states average usage comparison (unit: percentage)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **非节能意识调度器** | **节能意识调度器** | **两者相差** |
| C0 | 35.92 | 31.96 | -3.96 |
| C1 | 64.06 | 68.04 | 3.98 |

其中，C0为CPU正常运行状态，C1为CPU空闲状态。从表中数据可以看出，在有节能意识的调度器作用下，CPU处于运行状态的时间减少了约4个百分点而相应地处于空闲状态的时间增加约4个百分点了。因此，CPU的整体功耗得到了降低。

表5-2 P-states平均用量对比（单位：百分比）

Table 5-2 P-states average usage comparison (unit: percentage)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Silver Cluster | | | Golden Cluster | | |
| 时钟频率（单位：MHz） | 非节能意识调度器 | 节能意识调度器 | 两者相差 | 非节能意识调度器 | 节能意识调度器 | 两者相差 |
| 307 | 44.68 | 54.18 | 9.50 | 58.16 | 55.88 | -2.28 |
| 403 | N/A | N/A | N/A | 3.56 | 0.86 | -2.70 |
| 422 | 4.08 | 5.66 | 1.58 | N/A | N/A | N/A |
| 480 | 1.32 | 1.40 | 0.08 | 1.78 | 0.42 | -1.36 |
| 557 | 1.48 | 1.18 | -0.30 | 1.68 | 0.66 | -1.02 |
| 653 | 2.12 | 1.22 | -0.90 | 2.02 | 0.84 | -1.18 |
| 730 | 1.64 | 0.94 | -0.70 | 1.56 | 0.60 | -0.96 |
| 806 | N/A | N/A | N/A | 1.74 | 0.56 | -1.18 |
|  | | | | | | |
| 续表5-2 | | | | | | |
|  | Silver Cluster | | | Golden Cluster | | |
| 时钟频率（单位：MHz） | 非节能意识调度器 | 节能意识调度器 | 两者相差 | 非节能意识调度器 | 节能意识调度器 | 两者相差 |
| 845 | 2.26 | 1.36 | -0.90 | N/A | N/A | N/A |
| 883 | N/A | N/A | N/A | 1.46 | 0.56 | -0.90 |
| 960 | 13.76 | 16.02 | 2.26 | N/A | N/A | N/A |
| 1037 | 1.92 | 1.12 | -0.80 | 1.62 | 0.62 | -1.00 |
| 1114 | 2.02 | 1.10 | -0.92 | 1.12 | 1.42 | 0.30 |
| 1190 | 4.22 | 2.82 | -1.40 | 1.00 | 0.46 | -0.54 |
| 1229 | 1.42 | 0.64 | -0.78 | N/A | N/A | N/A |
| 1248 | N/A | N/A | N/A | 11.68 | 4.38 | -7.30 |
| 1325 | 1.74 | 0.74 | -1.00 | 1.38 | 0.46 | -0.92 |
| 1402 | 1.26 | 0.58 | -0.68 | 2.54 | 0.80 | -1.74 |
| 1478 | 2.04 | 0.88 | -1.16 | 2.82 | 0.54 | -2.28 |
| 1560 | N/A | N/A | N/A | 1.30 | 0.32 | -0.98 |
| 1600 | 14.12 | 10.20 | -3.92 | N/A | N/A | N/A |
| 1640 | N/A | N/A | N/A | 0.48 | 0.18 | -0.30 |
| 1710 | N/A | N/A | N/A | 0.34 | 0.10 | -0.24 |
| 1790 | N/A | N/A | N/A | 0.20 | 0.04 | -0.16 |
| 1830 | N/A | N/A | N/A | 0.00 | 0.04 | 0.04 |
| 1930 | N/A | N/A | N/A | 0.08 | 0.10 | 0.02 |
| 2000 | N/A | N/A | N/A | 0.10 | 0.10 | 0.00 |
| 2080 | N/A | N/A | N/A | 0.04 | 0.10 | 0.06 |

其中，N/A表示该CPU簇不支持在该时钟频率下运行。从表中数据可以看出，在节能意识的调度器作用下，功耗更低的Silver簇工作在低频率的时间增加了，如在307MHz时间增加了9.5个百分点，而工作在高频率的时间减少了，如1600MHz的时间减少了3.92个百分点；而对于功耗更高的Golden簇来说，整体的工作时间得到了降低。综合C-states和P-states的结果，考虑到CPU的功耗与其工作频率成正比，因此可以证明带有节能意识的调度器在轻量级系统负荷情况下确实能够降低CPU的整体功耗。

**5.2.1** 设备动态电源管理测试

针对Atmel MXT1664的功耗测试，我们的方法是定义浏览网页，阅读电子书和游戏三种和触摸屏相关的测试场景，分别进行T=30分钟的固定步骤的人工测试，统计使用固定超时机制(超时时间为ttimeout)和动态预测空闲时间两种情况下MXT1664处于runtime active和runtime suspended状态的时间ts-f和ts-d (单位: 秒)。这两个时间可以通过读取MXT1664在sysfs文件系统中对应设备节点的power属性下的runtime\_suspended\_time[5]这个属性来获取。每个测试场景测试10次计算tsuspended的平均值。

表5-3 触摸屏30分钟测试runtime suspended时间（单位：秒）

Table 5-3 Touchscreen runtime suspended time in 30 minutes test (unit: second)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **测试场景** | **固定超时机制(ttimeout=0.5秒)** | | **预测空闲时间(a=0.5)** | | **ts-d/ts-f** |
| ts-f | ts-f/T | ts-d | ts-d/T |
| 浏览网页 | 327.81 | 18.21% | 376.92 | 20.94% | 114.98% |
| 阅读电子书 | 544.16 | 30.23% | 663.88 | 36.88% | 122.00% |
| 游戏 | 226.08 | 12.56% | 244.62 | 13.59% | 108.20% |

从表5-3中数据可以看出，在这三种使用场景下，针对MXT1664的动态电源管理采用Runtime PM机制可以获得设备约13%到37%的处于低功耗状态时间。而在Runtime PM中采用预测空闲时间的方法可以比固定超时机制获得约8%到22%的功耗降低。

图5-1给出了预测空闲时间与实际空闲时间的误差比值曲线图。纵坐标表示误差比值 单位为百分比, 横坐标是采样序列。三角形和正方形分别表示浏览网页和阅读电子书两种情况。从测试结果看，这个预测算法在空闲时间相对稳定的情况下，如阅读电子书，比在空闲时间波动较大的情况下，如浏览网页，误差更小。



图 5-1 预测空闲时间误差比值曲线图

Figure 5-1 Error ratio graphs of idle time prediction

综合以上的测试结果，预测空闲时间的Runtime PM机制是一种降低设备动态功耗的有效方法。在此基础之上，我们可以通过自适应的预测因子改进线性预测算法，提高预测的准确性。

**5.3** 本章小结

本章首先介绍了调度器设计验证的测试环境，在此基础之上对采用默认CFS和有节能意识的调度器进行对比测试，通过比较CPU在不同频率的运行时间验证调度器改进设计能够带来CPU整体功耗的降低。

**6 总结与展望**

**6.1** 本文总结

本文深入研究了嵌入式系统的各种电源管理技术。设计并实现了一种有节能意识的调度器和基于Linux Runtime PM框架的设备动态电源管理机制。取得的主要研究成果包括：

(1) 提出了一种有节能意识的Linux内核调度器设计。该设计从采用基于历史窗口的负载追踪机制，根据各个CPU核的功耗成本进行任务放置和负载均衡，与CPUFreq框架的协同工作根据实时CPU工作负荷调整工作电压和运行以及提供运行时可调整参数五个方面达到降低CPU整体功耗的目的，解决目前嵌入式Linux系统中任务调度和CPU工作频率管理没有统筹的问题。

(2) 提出了一种基于Linux Runtime PM框架的设备动态电源管理机制。通过让设备进入Runtime suspended状态使设备进入硬件的低功耗模式。为避免频繁进出低功耗模式带来的额外开销，更有效地降低设备功耗，还提出了用预测空闲时间方法取代现有的基于固定超时机制的设计。为减少预测误差，我们采用了一种自适应预测因子的方法。

(3) 实现并验证具有节能意识的调度器设计和基于Runtime PM的设备动态电源管理设计。在高通MSM8996四核处理器硬件平台+Android 6.0.1操作系统环境下，通过BBench和PowerTop工具测试各CPU核在不同功耗和运行频率状态下的时间，验证了具有节能意识的调度器设计分别在CPU空闲和高频率运行时间上3.98%和3.92%的提升。在高通APQ8074硬件平台上，通过浏览网页、阅读电子书和游戏三种触摸屏测试，记录MXT1664触摸屏控制器处于低功耗模式的时间，验证了基于Runtime PM的设备动态电源管理设计能够带来最多约37%的低功耗时间。此外，采用预测空闲时间的方法可以有效避免频繁进出低功耗模式带来的额外开销，相比固定超时机制可以有最多约22%的功耗降低。通过预测误差分析得出自适应预测因子的方法在相对稳定的空闲时间使用场景下误差更小的结论。

**6.2** 未来工作展望

本文对嵌入式Linux系统的电源管理技术进行了分析并在此基础上深入研究了Linux内核的调度器和动态电源管理机制。设计并实现了现有的CFS调度器基础上的一种有节能意识的调度器和Runtime PM框架基础上的动态电源管理机制。但由于电源管理技术本身的复杂性和与嵌入式系统内其它组件的相关性，依然还有很多工作需要研究，其中包括：

持续改进有节能意识的调度器，尤其是作为整个调度器设计和节能意识基础的负载追踪。目前提出的基于时间窗口的负载追踪方法只是众多负载追踪数据模型中一种，由于嵌入式系统使用场景的多样化和系统负载的不确定性，我们需要在实际应用中不断改进和完善负载追踪的数据模型和设计方法。

通用Runtime PM和电源管理域。目前的Runtime PM和电源管理域还只限于在外设驱动，我们需要将这些机制扩展到嵌入式系统核心的CPU处理器，达到用同一种框架来统一嵌入式Linux系统的动态电源管理。此外，随着多核技术的发展和HPM架构的广泛应用，我们也需要将电源管理域扩展到CPU核的簇的层面上。

参考文献

[1] 张立. 基于DPM和DVFS的嵌入式系统低功耗优化技术研究[硕士论文]. 北京交通大学. 2011.

[2] Tam D, Tsang W, Drula C. Dynamic voltage scaling in mobile devices. Csc2228 project final report, University of Toront (Dec. 2003). 2003 Dec 15.

[3] Venkatesh Pallipadi, Shaohua Li, Adam Belay, cpuidle: Do nothing, efficiently, Proceedings of the Linux Symposium, 2007.

[4] 卢怡杰. Linux环境下低CPU资源消耗的应用软件低功耗研究[硕士论文]. 西北大学. 2012.

[5] Wysocki RJ. Runtime Power Management Framework for I. O Devices in the Linux Kernel, LinuxCon 2010. 2010 Jul 31.

[6] Tsitsikas, Kimon. Batch Scheduling for Energy-Efficient Sensing in Smartphones. Diss. TU Delft, Delft University of Technology, 2013.

[7] 林政伟. 手机厂商攻坚电池技术 受限电池材料难创新. 通信信息报. 2015-06-03.

[8] 黄仁甫，钱金维. 中小尺寸LCD驱动IC的背光省电技术-LABC/CABC. 现代显示. Jan., 2009, 总第96期

[9] Kamdar, Shubham, and Neha Kamdar. big. LITTLE Architecture: Heterogeneous Multicore Processing. International Journal of Computer Applications 119.1 (2015).

[10] Linaro, PM Architecture Block Diagram, https://wiki.linaro.org/WorkingGroups/

PowerManagement/Doc/Architecture.

[11] Z. Mwaikambo, A. Raj, R. Russel, and J. Schopp, Linux kernel hotplug CPU support, in Proc. Ottawa Linux Symp., Jul. 2004.

[12] Brown, A. Leonard, and Rafael J. Wysocki. Suspend-to-RAM in LinuxR. Linux symposium. 2008.

[13] 谢亮. 嵌入式系统低功耗设计[硕士论文]. 上海交通大学. 2011.

[14] M.Tim Jones. Inside the Linux 2.6 Completely Fair Scheduler. IBM developerWorks. 2009.

[15] 王益涵，王凯林，孙宪坤，等. 基于CPUfreq的DVFS节能技术的研究与实现.计算机测量与控制，2016.24(2): 151-154.

[16] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor, ols 2006.

[17] Benini, Luca, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 8.3 (2000): 299-316.

[18] Steve Muckle. “A QuIC Take on Energy-Aware Scheduling.” Linaro Connect USA 2014.

[19] Qualcomm. “PELT vs Window tracking and EAS on SMP multi-cluster”

[20] maxTouchTM 1664-channel Touchscreen Controller, Revision 0.9, Atmel, March, 2012.

[21] Hwang CH, Wu AC. A predictive system shutdown method for energy saving of event-driven computation. ACM Transactions on Design Automation of Electronic Systems (TODAES). 2000 Apr 1;5(2):226-41.

[22] MSM8996 Linux Android APSS CPU Power Management Overview, Revision C, Qualcomm, December, 2015.

[23] Linux drivers for Atmel maxTouch touchscreen controllers, Nick Dyer, <https://github.com/atmel-maxtouch/linux>

[24] 邱文华, 邱珍珍. 基于扁平设备树的 Linux 内核启动方式[J]. 现代计算机: 下半月版, 2009 (3): 115-118.

[25] Das A, Walker MJ, Hansson A, Al-Hashimi BM, Merrett GV. Hardware-software interaction for run-time power optimization: A case study of embedded Linux on multicore smartphones. InLow Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on 2015 Jul 22 (pp. 165-170). IEEE.

[26] Nicolas Pitre. Teaching the scheduler about power management. <https://lwn.net/Articles/602479>

[27] Wysocki, Rafael J. "Device PM QoS and Generic PM Domains." (2012).

# 附 录

附录I. update\_task\_demand()函数代码

static void update\_task\_demand(struct task\_struct \*p, struct rq \*rq,

int event, u64 wallclock)

{

u64 mark\_start = p->ravg.mark\_start;

u64 delta, window\_start = rq->window\_start;

int new\_window, nr\_full\_windows;

u32 window\_size = sched\_ravg\_window;

new\_window = mark\_start < window\_start;

if (!account\_busy\_for\_task\_demand(p, event)) {

if (new\_window)

/\* If the time accounted isn't being accounted as

\* busy time, and a new window started, only the

\* previous window need be closed out with the

\* pre-existing demand. Multiple windows may have

\* elapsed, but since empty windows are dropped,

\* it is not necessary to account those. \*/

update\_history(rq, p, p->ravg.sum, 1, event);

return;

}

if (!new\_window) {

/\* The simple case - busy time contained within the existing

\* window. \*/

add\_to\_task\_demand(rq, p, wallclock - mark\_start);

return;

}

/\* Busy time spans at least two windows. Temporarily rewind

\* window\_start to first window boundary after mark\_start. \*/

delta = window\_start - mark\_start;

nr\_full\_windows = div64\_u64(delta, window\_size);

window\_start -= (u64)nr\_full\_windows \* (u64)window\_size;

/\* Process (window\_start - mark\_start) first \*/

add\_to\_task\_demand(rq, p, window\_start - mark\_start);

/\* Push new sample(s) into task's demand history \*/

update\_history(rq, p, p->ravg.sum, 1, event);

if (nr\_full\_windows)

update\_history(rq, p, scale\_exec\_time(window\_size, rq),

nr\_full\_windows, event);

/\* Roll window\_start back to current to process any remainder

\* in current window. \*/

window\_start += (u64)nr\_full\_windows \* (u64)window\_size;

/\* Process (wallclock - window\_start) next \*/

mark\_start = window\_start;

add\_to\_task\_demand(rq, p, wallclock - mark\_start);

}

附录II. update\_history()函数代码

/\*

\* Called when new window is starting for a task, to record cpu usage over

\* recently concluded window(s). Normally 'samples' should be 1. It can be > 1

\* when, say, a real-time task runs without preemption for several windows at a

\* stretch.

\*/

static void update\_history(struct rq \*rq, struct task\_struct \*p,

u32 runtime, int samples, int event)

{

u32 \*hist = &p->ravg.sum\_history[0];

int ridx, widx;

u32 max = 0, avg, demand;

u64 sum = 0;

/\* Ignore windows where task had no activity \*/

if (!runtime || is\_idle\_task(p) || exiting\_task(p) || !samples)

goto done;

/\* Push new 'runtime' value onto stack \*/

widx = sched\_ravg\_hist\_size - 1;

ridx = widx - samples;

for (; ridx >= 0; --widx, --ridx) {

hist[widx] = hist[ridx];

sum += hist[widx];

if (hist[widx] > max)

max = hist[widx];

}

for (widx = 0; widx < samples && widx < sched\_ravg\_hist\_size; widx++) {

hist[widx] = runtime;

sum += hist[widx];

if (hist[widx] > max)

max = hist[widx];

}

p->ravg.sum = 0;

if (sched\_window\_stats\_policy == WINDOW\_STATS\_RECENT) {

demand = runtime;

} else if (sched\_window\_stats\_policy == WINDOW\_STATS\_MAX) {

demand = max;

} else {

avg = div64\_u64(sum, sched\_ravg\_hist\_size);

if (sched\_window\_stats\_policy == WINDOW\_STATS\_AVG)

demand = avg;

else

demand = max(avg, runtime);

}

/\*

\* A throttled deadline sched class task gets dequeued without

\* changing p->on\_rq. Since the dequeue decrements hmp stats

\* avoid decrementing it here again.

\*/

if (task\_on\_rq\_queued(p) && (!task\_has\_dl\_policy(p) ||

!p->dl.dl\_throttled))

p->sched\_class->fixup\_hmp\_sched\_stats(rq, p, demand);

p->ravg.demand = demand;

done:

trace\_sched\_update\_history(rq, p, runtime, samples, event);

}

其中，p指向当前任务的task\_struct结构，samples表示需要更新的最近的窗口个数，runtime是每个这样的窗口中任务的运行时间。sched\_window\_stats\_policy是生成负载统计值的策略，在表3-3中描述。

附录III. update\_cpu\_busy\_time()函数代码

/\*

\* Account cpu activity in its busy time counters (rq->curr/prev\_runnable\_sum)

\*/

static void update\_cpu\_busy\_time(struct task\_struct \*p, struct rq \*rq,

int event, u64 wallclock, u64 irqtime)

{

int new\_window, nr\_full\_windows = 0;

int p\_is\_curr\_task = (p == rq->curr);

u64 mark\_start = p->ravg.mark\_start;

u64 window\_start = rq->window\_start;

u32 window\_size = walt\_ravg\_window;

u64 delta;

new\_window = mark\_start < window\_start;

if (new\_window) {

nr\_full\_windows = div64\_u64((window\_start - mark\_start),

window\_size);

if (p->ravg.active\_windows < USHRT\_MAX)

p->ravg.active\_windows++;

}

/\* Handle per-task window rollover. We don't care about the idle

\* task or exiting tasks. \*/

if (new\_window && !is\_idle\_task(p) && !exiting\_task(p)) {

u32 curr\_window = 0;

if (!nr\_full\_windows)

curr\_window = p->ravg.curr\_window;

p->ravg.prev\_window = curr\_window;

p->ravg.curr\_window = 0;

}

if (!account\_busy\_for\_cpu\_time(rq, p, irqtime, event)) {

/\* account\_busy\_for\_cpu\_time() = 0, so no update to the

\* task's current window needs to be made. This could be

\* for example

\*

\* - a wakeup event on a task within the current

\* window (!new\_window below, no action required),

\* - switching to a new task from idle (PICK\_NEXT\_TASK)

\* in a new window where irqtime is 0 and we aren't

\* waiting on IO \*/

if (!new\_window)

return;

/\* A new window has started. The RQ demand must be rolled

\* over if p is the current task. \*/

if (p\_is\_curr\_task) {

u64 prev\_sum = 0;

/\* p is either idle task or an exiting task \*/

if (!nr\_full\_windows) {

prev\_sum = rq->curr\_runnable\_sum;

}

rq->prev\_runnable\_sum = prev\_sum;

rq->curr\_runnable\_sum = 0;

}

return;

}

if (!new\_window) {

/\* account\_busy\_for\_cpu\_time() = 1 so busy time needs

\* to be accounted to the current window. No rollover

\* since we didn't start a new window. An example of this is

\* when a task starts execution and then sleeps within the

\* same window. \*/

if (!irqtime || !is\_idle\_task(p) || cpu\_is\_waiting\_on\_io(rq))

delta = wallclock - mark\_start;

else

delta = irqtime;

delta = scale\_exec\_time(delta, rq);

rq->curr\_runnable\_sum += delta;

if (!is\_idle\_task(p) && !exiting\_task(p))

p->ravg.curr\_window += delta;

return;

}

if (!p\_is\_curr\_task) {

/\* account\_busy\_for\_cpu\_time() = 1 so busy time needs

\* to be accounted to the current window. A new window

\* has also started, but p is not the current task, so the

\* window is not rolled over - just split up and account

\* as necessary into curr and prev. The window is only

\* rolled over when a new window is processed for the current

\* task.

\*

\* Irqtime can't be accounted by a task that isn't the

\* currently running task. \*/

if (!nr\_full\_windows) {

/\* A full window hasn't elapsed, account partial

\* contribution to previous completed window. \*/

delta = scale\_exec\_time(window\_start - mark\_start, rq);

if (!exiting\_task(p))

p->ravg.prev\_window += delta;

} else {

/\* Since at least one full window has elapsed,

\* the contribution to the previous window is the

\* full window (window\_size). \*/

delta = scale\_exec\_time(window\_size, rq);

if (!exiting\_task(p))

p->ravg.prev\_window = delta;

}

rq->prev\_runnable\_sum += delta;

/\* Account piece of busy time in the current window. \*/

delta = scale\_exec\_time(wallclock - window\_start, rq);

rq->curr\_runnable\_sum += delta;

if (!exiting\_task(p))

p->ravg.curr\_window = delta;

return;

}

if (!irqtime || !is\_idle\_task(p) || cpu\_is\_waiting\_on\_io(rq)) {

/\* account\_busy\_for\_cpu\_time() = 1 so busy time needs

\* to be accounted to the current window. A new window

\* has started and p is the current task so rollover is

\* needed. If any of these three above conditions are true

\* then this busy time can't be accounted as irqtime.

\*

\* Busy time for the idle task or exiting tasks need not

\* be accounted.

\*

\* An example of this would be a task that starts execution

\* and then sleeps once a new window has begun. \*/

if (!nr\_full\_windows) {

/\* A full window hasn't elapsed, account partial

\* contribution to previous completed window. \*/

delta = scale\_exec\_time(window\_start - mark\_start, rq);

if (!is\_idle\_task(p) && !exiting\_task(p))

p->ravg.prev\_window += delta;

delta += rq->curr\_runnable\_sum;

} else {

/\* Since at least one full window has elapsed,

\* the contribution to the previous window is the

\* full window (window\_size). \*/

delta = scale\_exec\_time(window\_size, rq);

if (!is\_idle\_task(p) && !exiting\_task(p))

p->ravg.prev\_window = delta;

}

/\*

\* Rollover for normal runnable sum is done here by overwriting

\* the values in prev\_runnable\_sum and curr\_runnable\_sum.

\* Rollover for new task runnable sum has completed by previous

\* if-else statement.

\*/

rq->prev\_runnable\_sum = delta;

/\* Account piece of busy time in the current window. \*/

delta = scale\_exec\_time(wallclock - window\_start, rq);

rq->curr\_runnable\_sum = delta;

if (!is\_idle\_task(p) && !exiting\_task(p))

p->ravg.curr\_window = delta;

return;

}

if (irqtime) {

/\* account\_busy\_for\_cpu\_time() = 1 so busy time needs

\* to be accounted to the current window. A new window

\* has started and p is the current task so rollover is

\* needed. The current task must be the idle task because

\* irqtime is not accounted for any other task.

\*

\* Irqtime will be accounted each time we process IRQ activity

\* after a period of idleness, so we know the IRQ busy time

\* started at wallclock - irqtime. \*/

BUG\_ON(!is\_idle\_task(p));

mark\_start = wallclock - irqtime;

/\* Roll window over. If IRQ busy time was just in the current

\* window then that is all that need be accounted. \*/

rq->prev\_runnable\_sum = rq->curr\_runnable\_sum;

if (mark\_start > window\_start) {

rq->curr\_runnable\_sum = scale\_exec\_time(irqtime, rq);

return;

}

/\* The IRQ busy time spanned multiple windows. Process the

\* busy time preceding the current window start first. \*/

delta = window\_start - mark\_start;

if (delta > window\_size)

delta = window\_size;

delta = scale\_exec\_time(delta, rq);

rq->prev\_runnable\_sum += delta;

/\* Process the remaining IRQ busy time in the current window. \*/

delta = wallclock - window\_start;

rq->curr\_runnable\_sum = scale\_exec\_time(delta, rq);

return;

}

BUG();

}

附录IV. mxt\_runtime\_suspend()函数代码

int mxt\_runtime\_suspend(struct device \*dev)

{

struct i2c\_client \*client = to\_i2c\_client(dev);

struct mxt\_data \*data = i2c\_get\_clientdata(client);

struct input\_dev \*input\_dev = data->input\_dev;

int ret = 0;

mutex\_lock(&input\_dev->mutex);

if (data->num\_pending\_requests > 0) {

ret = -EBUSY;

} else {

/\* put device to Idle mode \*/

mxt\_set\_state(client, MXT\_STATE\_IDLE);

data->is\_suspended = 1;

}

mutex\_unlock(&input\_dev->mutex);

return ret;

}

其中，mxt\_set\_state()函数是用来通过写I2C寄存器将MXT1664切换到指定状态的。

附录V. mxt\_runtime\_resume()函数代码

int mxt\_runtime\_resume(struct device \*dev)

{

struct i2c\_client \*client = to\_i2c\_client(dev);

struct mxt\_data \*data = i2c\_get\_clientdata(client);

struct input\_dev \*input\_dev = data->input\_dev;

mutex\_lock(&input\_dev->mutex);

mxt\_set\_state(client, MXT\_STATE\_ACTIVE);

data->is\_suspended = 0;

pm\_runtime\_mark\_last\_busy(&data->dev);

if (data->num\_pending\_requests > 0)

mxt\_process\_requests(data);

mutex\_unlock(&inputdev->mutex);

return 0;

}

附录VI. mxt\_io\_complete()函数代码

void mxt\_io\_completion(struct mxt\_data \*data)

{

struct input\_dev \*input\_dev = data->input\_dev;

mutex\_lock(&input\_dev->mutex);

if (--data->num\_pending\_requests == 0) {

pm\_runtime\_mark\_last\_busy(&data->dev);

pm\_runtime\_put (&data->dev);

} else {

mxt\_process\_next\_request(data);

}

data->idle\_start = sched\_clock();

mutex\_unlock(&inputdev->mutex);

/\* Send req result back to the user ... \*/

}

附录VII. mxt\_runtime\_idle()函数代码

int mxt\_runtime\_idle(struct device \*dev)

{

struct i2c\_client \*client = to\_i2c\_client(dev);

struct mxt\_data \*data = i2c\_get\_clientdata(client);

struct input\_dev \*input\_dev = data->input\_dev;

int ret = 0;

mutex\_lock(&input\_dev->mutex);

if (data->num\_pending\_requests > 0) {

ret = -EBUSY;

} else {

predict = calc\_predict\_idle\_time(data);

if (predict <= MXT\_IDLE\_THRES)

ret = -EAGAIN;

data->prev\_predict = predict;

}

mutex\_unlock(&inputdev->mutex);

return ret;

}

其中，MXT\_IDLE\_THRES是根据公式3-2计算出的空闲时间阈值Ith。calc\_predict\_idle\_time()函数是用来计算预测设备可处于空闲时间的。它的实现如下：

#define INIT\_PREDICT\_FACTOR 50

#define PREDICT\_SCALE (100)

#define MIN(a,b) (a < b? a : b)

#define MAX(a,b) (a > b? a : b)

static unsigned int calc\_predict\_factor(prev\_idle, prev\_predict)

{

unsigned int factor;

if (!prev\_predict)

factor = INIT\_PREDICT\_FACTOR;

else

factor = PREDICT\_SCALE \* (MAX(prev\_idle, prev\_predict) - MIN(prev\_idle, prev\_predict)) / MAX(prev\_idle, prev\_predict);

return factor;

}

static unsigned long calc\_predict\_idle\_time(struct mxt\_data \*data)

{

unsigned long prev\_idle, wall\_clock, predict;

unsigned int predict\_factor;

wall\_clock = sched\_clock();

data->prev\_idle = wall\_clock - data->idle\_start;

predict\_factor = calc\_predict\_factor(data->prev\_idle, data->prev\_predict);

predict = data->predict\_factor \* prev\_idle

+ (PREDICT\_SCALE - predict\_factor)

\* data->prev\_predict;

predict /= PREDICT\_SCALE;

return predict;

}

**致 谢**

在本文即将结束之际，我要对在这两年多来的硕士研究生学习、科研和生活中给予过我支持和鼓励的人们表示深深的感谢。

首先，我非常感谢我的导师——王东副教授。本论文自始至终都是在王老师的悉心指导下完成的，其中的每一步进展和每一个成果无不凝聚着导师的心血与汗水。同时他勤奋的工作作风、勤于思考的研究态度、积极进取的勇气和魄力，以及对我严格的要求等等，使我受益匪浅，值得我终身学习，这都是比知识的获得还要宝贵的财富。

其次我还应当感谢我的家人，是他们多年的养育和关爱才有了我今天能够取得的成绩。同时，不能忘记的还有在工作中一直给予我指导的主管王忠阳经理和各位同事，他们在我工作期间给与我许多的启发和帮助，这份经历及这份友谊将是我终身值得怀念的。

我应该感谢的人太多太多，很难这里一一列出他们的姓名，但是我会把所有美好的回忆都永远留在心中，大家给予我的支持、帮助和鼓励将会永远伴随我将来的道路。

# 攻读学位期间发表的学术论文目录

[1] 凌云,王东. 嵌入式Linux系统电源管理技术的研究，上海交通大学院内公示