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

年 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

April, 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

这是摘要

关键词 关键词1，关键词2，关键词3

**IMPROVEMENT AND IMPLEMENTATION OF**

**POWER MANAGEMENT**

**IN EMBEDDED LINUX SYSTEM**

**ABSTRACT**

This is abstract

**Keywords** keywords1, keywords2, keywords3

**目 录**

**1**绪论--------------------------------------------------------------------------------------------------1

1.1 研究背景------------------------------------------------------------1

1.1.1 绪论1.1.1节-------------------------------------------------------------1

----------------------------------------------- -------------------

1.2 研究现状----------------------------------------------------------------------------------1

**2** DME均质充量压燃着火的数值模拟方法-----------------------------------------------------2

2.1 二级标题----------------------------------------------------------------------------------2

2.1.1 三级标题------------------------------------------------------------------------2

-------------------------------------------------------------------

-------------------------------------------------------------------

**5**结论------------------------------------------------------------------------------------------------6

参考文献----------------------------------------------------------------------------------------------7

附录--------------------------------------------------------------------------------------------8

致谢---------------------------------------------------------------------------------------------------9

攻读学位期间发表的学术论文目录-----------------------------------------------------------10

**1 绪 论**

**1.1** 研究背景

正文内容

**1.2** 研究现状

正文内容

**1.3** 本文研究工作

正文内容

**1.4** 论文结构

正文内容

**2 嵌入式Linux系统电源管理技术分析**

**2.1** 文献综述

正文内容

**2.1.1** 三级标题

正文内容

**3 嵌入式Linux系统电源管理方法的改进**

**3.1** 一种有节能意识的调度器设计

从前面第二章的分析的分析可以得知，有节能意识的调度器(Energy-Aware Scheduling)(简称EAS)是目前嵌入式Linux系统电源管理方法可以改进的一个方向。本节就将从负载追踪、任务放置、负载均衡和可调整参数这几个方面介绍一种基于目前Linux内核CFS (Completely Fair Scheduler) 调度器改进的EAS调度器。

**3.1.1** 负载追踪的改进

从内核3.8版本开始，CFS的负载追踪(Load Tracking)(简称LT)是一种基于entity的机制(per-entity load tracking)(简称PELT)[]。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可以用公式表示为

*L = L0 + L1\*y + L2\*y2 + L3\*y3 + …* (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在设计时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。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%[]。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)。

在WinLT中，主要包含以下设计思想：

1. 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。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。
2. 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：
   1. 任务开始在CPU上运行；
   2. 任务在CPU上运行结束；
   3. 任务被从睡眠状态唤醒；
   4. 任务完成退出时；
   5. 任务从一个CPU转移到另外一个CPU运行；
   6. 调度器时钟中断发生时；
   7. 硬件中断处理完成时；
   8. CPU频率发生变化时。
3. 每一个任务的CPU的负载要求计算依据不同的策略执行。这些策略包括：
   1. 最大值(max)，即取N个窗口的负载值中的最大值；
   2. 平均值(avg),N个窗口的负载值求平均；
   3. 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。

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。

1. 时间窗口的起止时间在各个CPU上是同步的。
2. CPU负载的计算是归一化的。归一到CPU运行频率和多核CPU拓扑架构上。这可以用下面的公式表示：

Load = t \* (fcur / fmax) \* (1 / Smax) (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。

其中第3）项是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：

1. 判断当前时间对应的窗口是否是任务的一个新窗口；
2. 如果当前窗口是活跃窗口，更新当前窗口的负载值；
3. 如果不是新窗口，结束；
4. 用当前1个新窗口的负载值更新任务的历史负载纪录；
5. 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；
6. 返回步骤1）。

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：

1. 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；
2. 将负载历史数组中的前N－n个窗口内的值右移n个元素;
3. 将新纪录的n个窗口内的负载值加到数组头上；
4. 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；
5. 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。

**3.1.2** 任务放置的改进

任务放置(Task Placement)的主要目的是为所要调度的任务选择最合适的CPU。在我们的设计中，它在以下时刻进行：

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

任务放置的核心算法可表述为：

1. 找到一个中断负载低于高中断负载阈值的合适的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）；
2. 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；
3. 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。
4. 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。

算法的流程图如下：

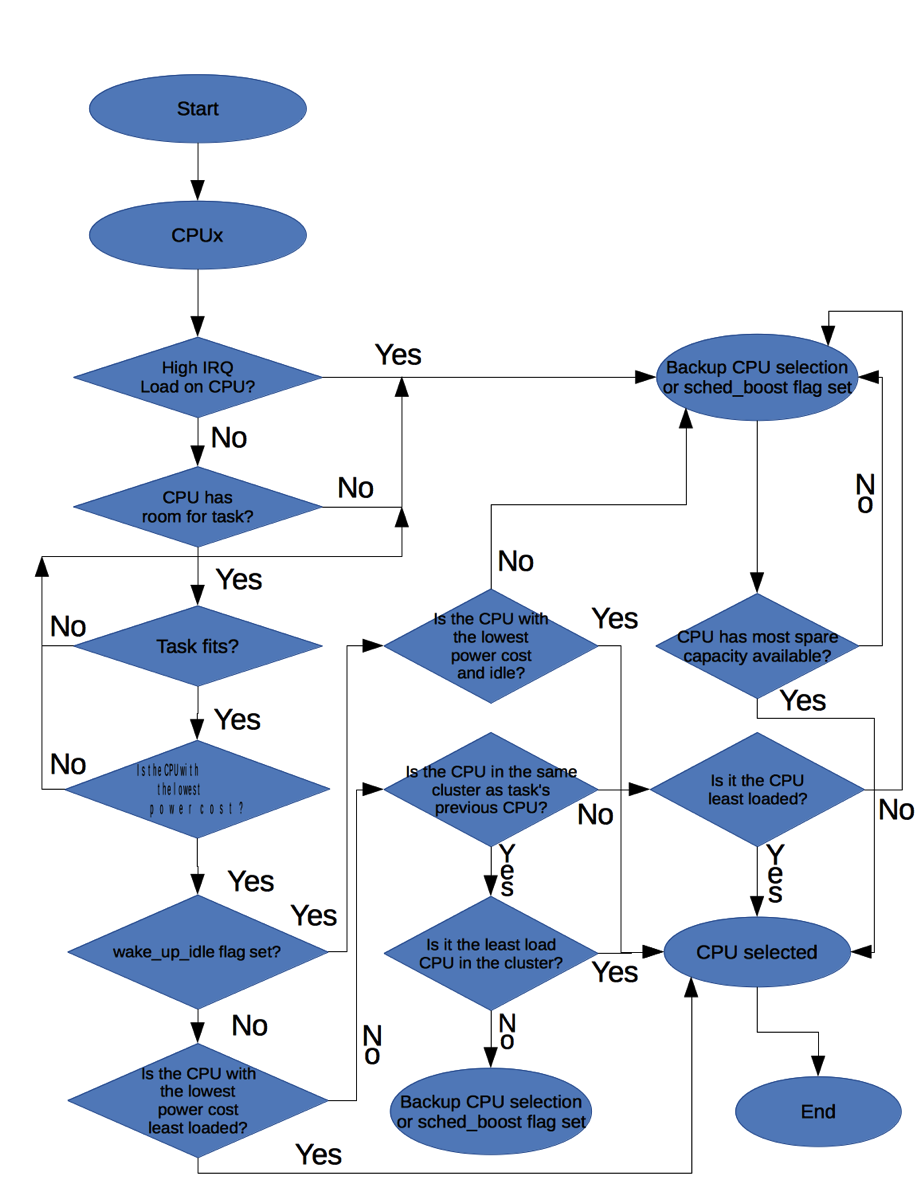


图 3-1 任务放置算法流程图

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：

1. 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。

1. 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(Load Balancing)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。

负载均衡算法可表述为：

1. 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。
2. 对于最忙碌的CPU簇，找出该簇内最忙碌的CPU核。
3. 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。

在现有的实时(Real Time,简称RT)任务调度中，RT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：

1. 那个CPU正在运行另外一个绑定在该CPU上的RT任务；
2. 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

如第二章所描述，具有节能意识的调度器相对现有的调度器实现的一个改进在于对CPU频率调整的反馈，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。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：

1. 唤醒一个任务；
2. 任务被升级调度到另外一个CPU；
3. 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

该调度器包含下列可调整参数：

表3-1 sched\_ravg\_window可调整参数

Table 3-1 sched\_ravg\_window tunable parameter

|  |  |
| --- | --- |
| 名称 | sched\_ravg\_window |
| 位置 | 内核命令行参数 |
| 默认值 | 10ms |
| 描述 | WinLT中每一个时间窗口的长度 |

表3-2 RAVG\_HIST\_SIZE\_MAX可调整参数

Table 3-2 RAVG\_HIST\_SIZE\_MAX tunable parameter

|  |  |
| --- | --- |
| 名称 | RAVG\_HIST\_SIZE\_MAX |
| 位置 | 预编译宏 |
| 默认值 | 5 |
| 描述 | 每一个任务的负载追踪窗口个数 |

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

Table 3-3 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 |
| 描述 | 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。取值范围如下：  WINDOW\_STATS\_RECENT - 最近一个窗口值  WINDOW\_STATS\_MAX - 所有窗口中的最大值  WINDOW\_STATS\_MAX\_RECENT\_AVG - 最近一个窗口和所有窗口平均值之间的大者  WINDOW\_STATS\_AVG - 所有窗口中的平均值  WINDOW\_STATS\_INVALID\_POLICY - 非法值 |

表3-4 sched\_init\_task\_load可调整参数

Table 3-4 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 |
| 描述 | 对于新创建没有历史纪录的任务的初始负载百分比。 |

表3-5 sched\_account\_wait\_time可调整参数

Table 3-5 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 |
| 描述 | 用于判断是否将任务的等待时间计入任务对CPU时间要求对标志。 |

表3-6 sched\_upmigrate可调整参数

Table 3-6 sched\_upmigrate tunable parameter

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

表3-7 sched\_spill\_nr\_run可调整参数

Table 3-7 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 |
| 描述 | 一个CPU能够同时容纳的可运行任务的最大个数。 |

表3-8 sched\_spill\_load可调整参数

Table 3-8 sched\_spill\_load tunable parameter

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

表3-9 cpu\_high\_irqload可调整参数

Table 3-9 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 |

**3.2** 设备驱动的动态电源管理设计

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。Runtime PM可以根据设备的使用量计数动态地将设备由active状态切换到suspended状态或者相反，从而在不影响设备正常使用的情况下达到动态降低设备功耗的目的。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。

**3.2.1** 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[]：

* 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下：



图 3-2 MXT1664接口原理图

Figure 3-2 MXT1664 interface schematics

MXT1664有三种工作模式: Active, Idle和Deep Sleep[]。其中功耗最低的工作模式是DeepSleep。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下表所示：

表3-10 MXT1664S三种模式下耗电情况（单位：mA）

Table 3-10 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3.2.2节中详细描述。

1. 驱动流程

基于以上目标，MXT1664驱动基于Runtime PM的动态电源管理流程可以表述为：

* 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acitve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。

整个流程如下图表示：

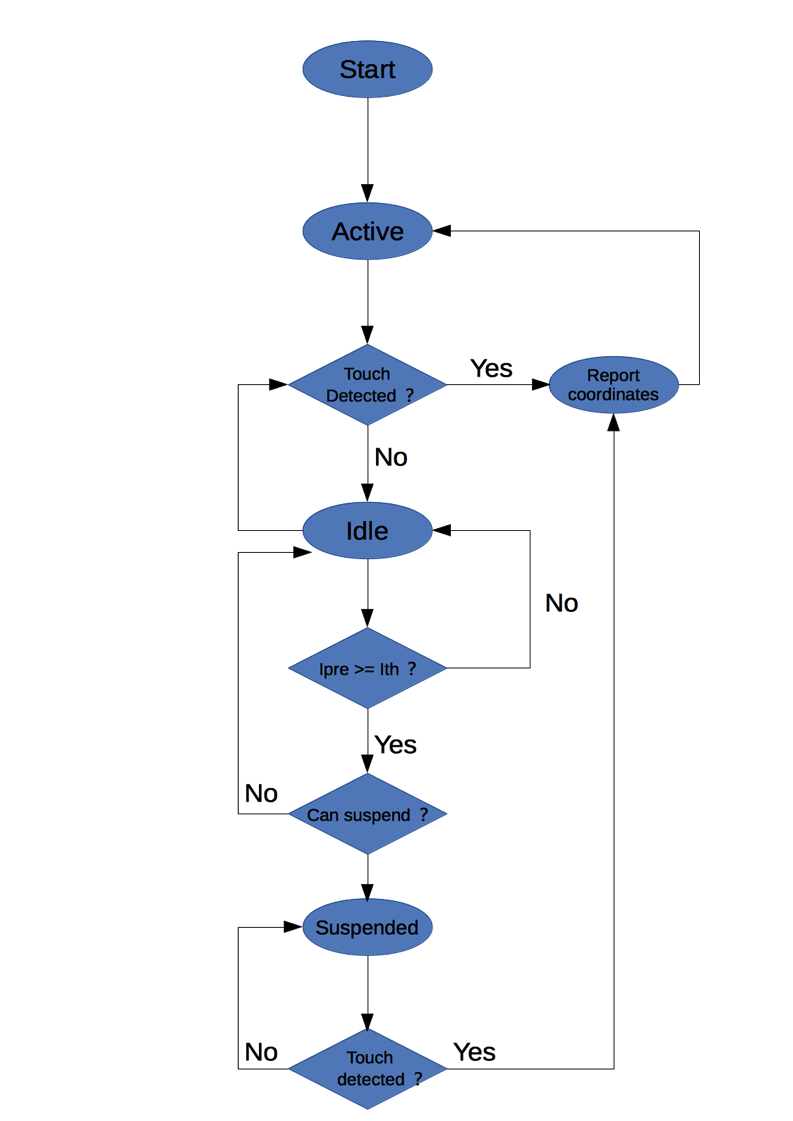


图 3-3 MXT1664驱动Runtime PM流程

Figure 3-3 MXT1664 driver Runtime PM 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，将在3.2.2节中具体描述。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。此外， mxt\_runtime\_suspend()需要在将设备切换到低功耗状态之前同步I/O请求的状态以确保所有的I/O请求都已经完成。否则，设备将不能进入Runtime PM suspended状态。它的流程可以如下图表示：

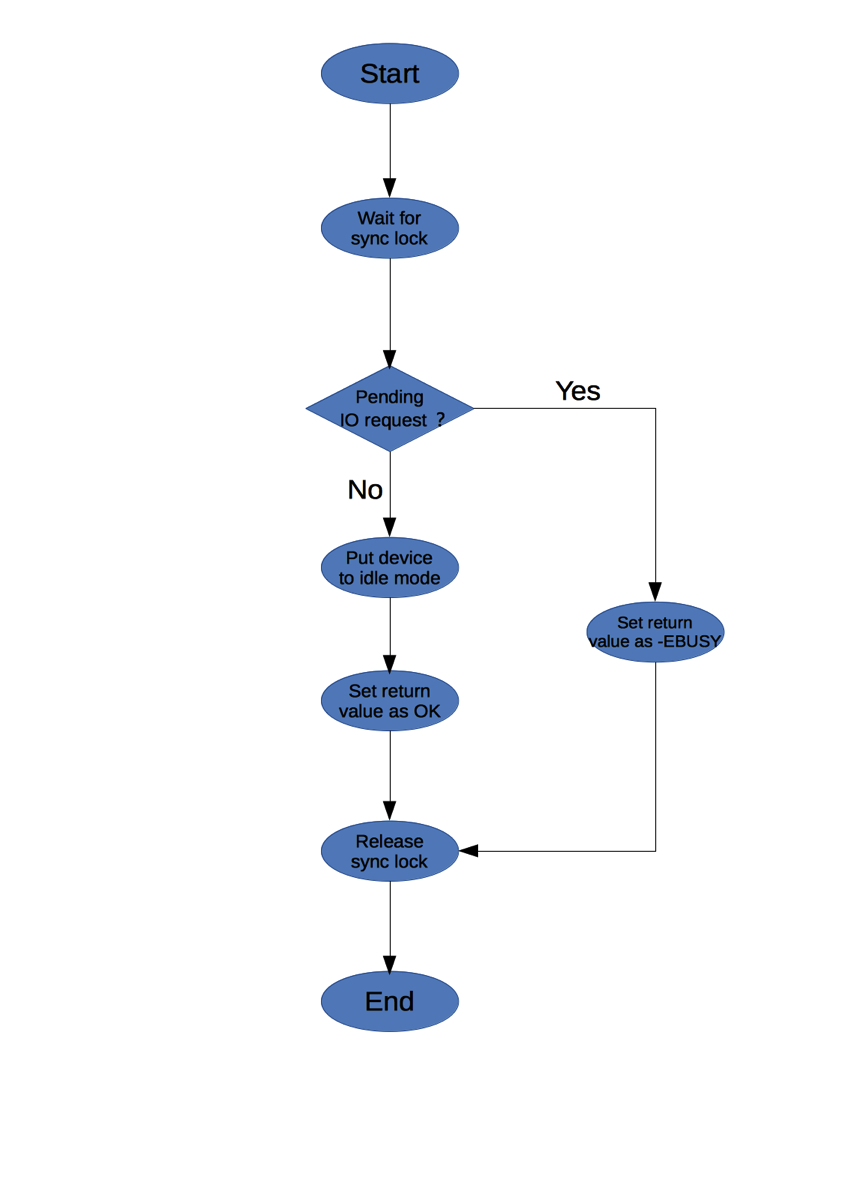


图 3-4 mxt\_runtime\_suspend（）流程

Figure 3-4 mxt\_runtime\_suspend() flow

mxt\_runtime\_resume()函数的作用与mxt\_runtime\_suspend()相反，用于将设备从低功耗状态恢复到正常运行状态。同时，如果在设备进入suspended之后有来自上层应用的I/O请求，在此时需要处理它们。它的流程如下图所示：

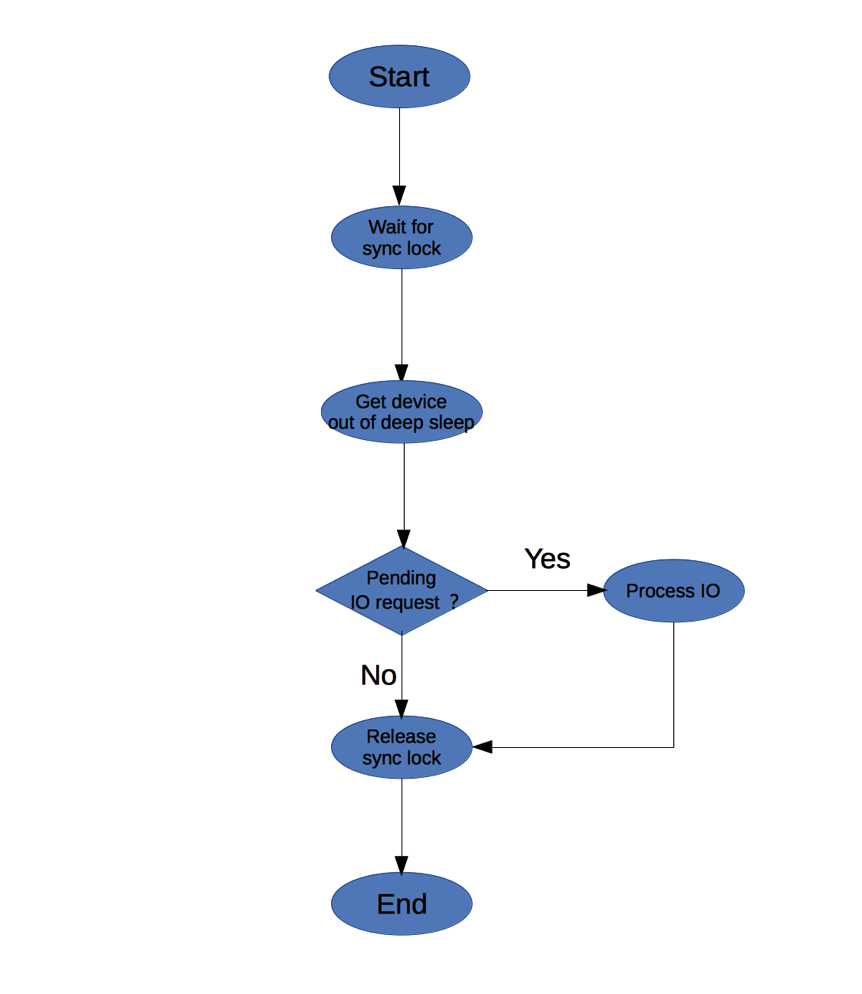


图 3-5 mxt\_runtime\_resume()流程

Figure 3-5 mxt\_runtime\_resume() flow

mxt\_io\_complete()函数用于同步异步I/O环境下的I/O请求同步以确保设备在进入Runtime PM suspended状态之前处理完所有等待的I/O请求。它的流程如下图所示：

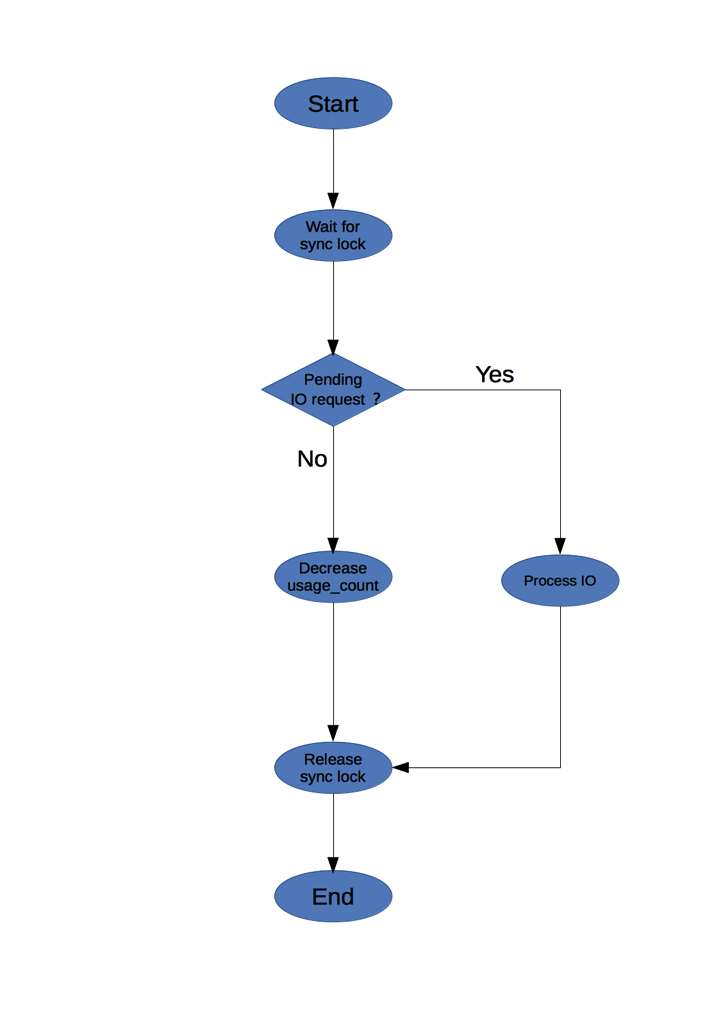


图 3-6 mxt\_io\_complete()流程

Figure 3-6 mxt\_io\_complete() flow

**3.2.2** 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[]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Ith的最小值可以用下面的公式表示：

(3-2)

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，我们用下面的方法预测设备下一次空闲时间In＋1：

In+1 = a\*in + (1 - a)\*In  (3-3)

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（3-3）式预测的设备空闲时间；a是预测因子，a的取值范围是大于等于0且小于等于1。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中断处理函数入口处纪录空闲状态的结束时间，两者相减作为上一次设备的实际空闲时间in。增加mxt\_runtime\_idle()函数用于预测下一次设备的空闲时间并当In＋1大于Ith时，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。它的流程可以用下图表示：

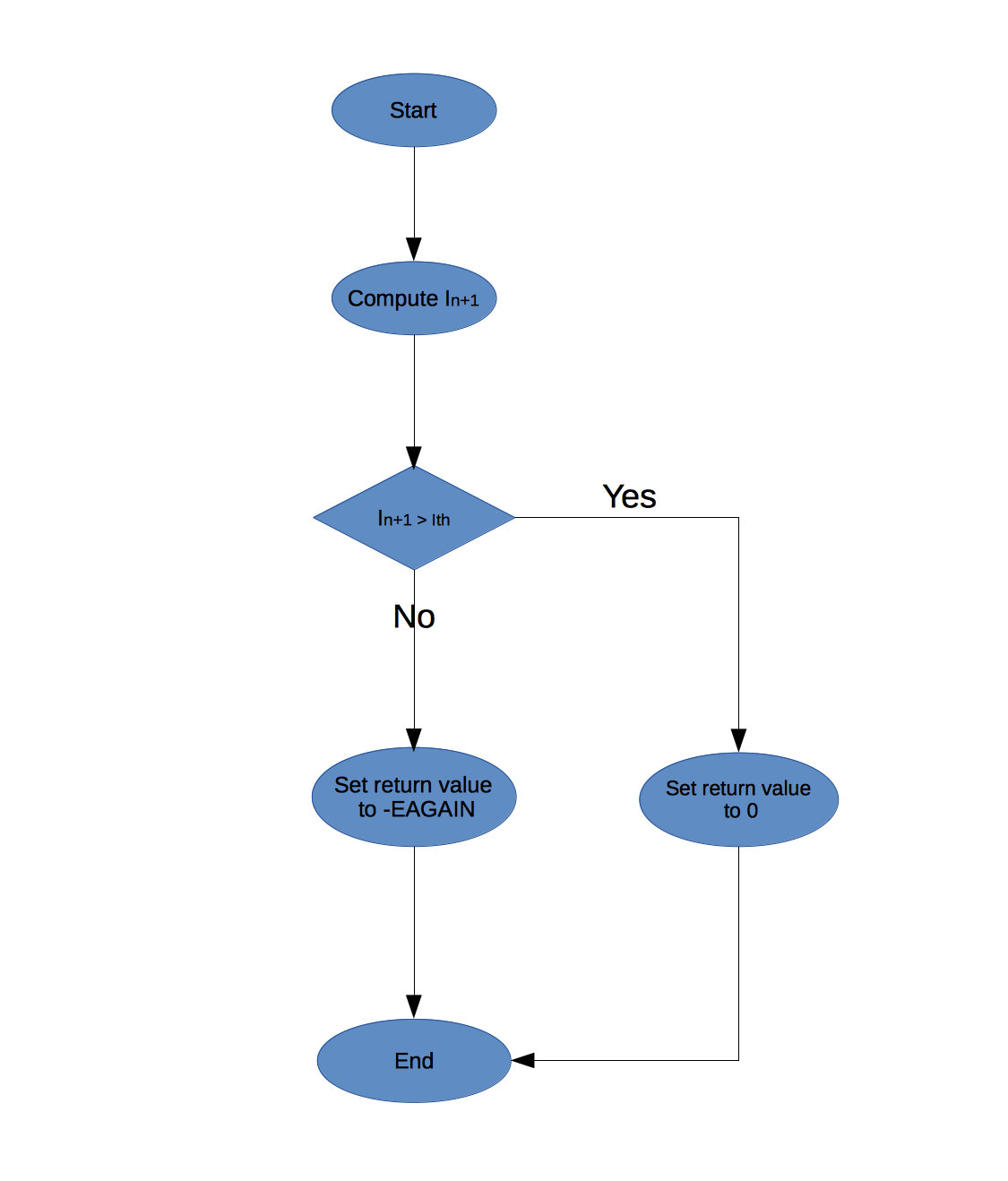


图 3-7 mxt\_runtime\_idle()流程

Figure 3-7 mxt\_runtime\_idle() flow

**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

我们在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。

**4.1.1** 主要数据结构和算法实现

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）。它的主要成员变量定义如下：

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。它的实现代码如下：

/\*

\* 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。

为实现与CPUFreq框架的互动，调度器实现sched\_get\_cpus\_busy()接口供CPUFreq governor查询CPU负载情况。这个接口的实现关键部分如下：

void sched\_get\_cpus\_busy(struct sched\_load \*busy, const struct cpumask \*query\_cpus)

{

unsigned long flags;

struct rq \*rq;

const int cpus = cpumask\_weight(query\_cpus);

u64 load[cpus], nload[cpus];

unsigned int cur\_freq[cpus], max\_freq[cpus];

int notifier\_sent[cpus];

int early\_detection[cpus];

int cpu, i = 0;

unsigned int window\_size;

…

window\_size = sched\_ravg\_window;

for\_each\_cpu(cpu, query\_cpus) {

rq = cpu\_rq(cpu);

update\_task\_ravg(rq->curr, rq, TASK\_UPDATE,

sched\_ktime\_clock(), 0);

load[i] = rq->old\_busy\_time = rq->prev\_runnable\_sum;

nload[i] = rq->nt\_prev\_runnable\_sum;

/\*

\* Scale load in reference to cluster max\_possible\_freq.

\*

\* Note that scale\_load\_to\_cpu() scales load in reference to

\* the cluster max\_freq.

\*/

load[i] = scale\_load\_to\_cpu(load[i], cpu);

nload[i] = scale\_load\_to\_cpu(nload[i], cpu);

notifier\_sent[i] = rq->notifier\_sent;

early\_detection[i] = (rq->ed\_task != NULL);

rq->notifier\_sent = 0;

cur\_freq[i] = cpu\_cur\_freq(cpu);

max\_freq[i] = cpu\_max\_freq(cpu);

i++;

}

…

for\_each\_cpu(cpu, query\_cpus) {

rq = cpu\_rq(cpu);

if (early\_detection[i]) {

busy[i].prev\_load = div64\_u64(sched\_ravg\_window,

NSEC\_PER\_USEC);

busy[i].new\_task\_load = 0;

goto exit\_early;

}

if (!notifier\_sent[i]) {

load[i] = scale\_load\_to\_freq(load[i], max\_freq[i],

cur\_freq[i]);

nload[i] = scale\_load\_to\_freq(nload[i], max\_freq[i],

cur\_freq[i]);

if (load[i] > window\_size)

load[i] = window\_size;

if (nload[i] > window\_size)

nload[i] = window\_size;

load[i] = scale\_load\_to\_freq(load[i], cur\_freq[i],

cpu\_max\_possible\_freq(cpu));

nload[i] = scale\_load\_to\_freq(nload[i], cur\_freq[i],

cpu\_max\_possible\_freq(cpu));

} else {

load[i] = scale\_load\_to\_freq(load[i], max\_freq[i],

cpu\_max\_possible\_freq(cpu));

nload[i] = scale\_load\_to\_freq(nload[i], max\_freq[i],

cpu\_max\_possible\_freq(cpu));

}

busy[i].prev\_load = div64\_u64(load[i], NSEC\_PER\_USEC);

busy[i].new\_task\_load = div64\_u64(nload[i], NSEC\_PER\_USEC);

exit\_early:

trace\_sched\_get\_busy(cpu, busy[i].prev\_load,

busy[i].new\_task\_load, early\_detection[i]);

i++;

}

}

**4.1.2** 性能评估测试结果

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[]。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：

1. CPU核处于各C-states的时间和百分比；
2. CPU核的P-states，即CPU核处于不同时钟频率的时间和百分比；
3. 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。

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平均用量如下表4-2和4-3所示：

表4-2 C-states平均用量对比（单位：百分比）

Table 4-2 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4-3 P-states平均用量对比（单位：百分比）

Table 4-3 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 |
| 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 |
| 941 | N/A | N/A | N/A | 1.06 | 0.34 | -0.72 |
| 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 总结与展望**

正文内容

**三号黑体粗体居中，上下各空一行**

**按论文中参考文献出现的次序，用中括号的数字连续编号，顶格书写，小四号宋体，1.25倍行距。最少20篇**

**人**

参考文献

普通图书

[1] 蒋有绪，郭泉水，马娟等，中国森林群落分类及其群落学特征[M]，北京，科学出版社，1998，11-12.

论文集、

会议录

# 附 录

**附录**

**主要列入正文内过分冗长的公式推导，供查读方便所需的辅助性数学工具或表格；重复性数据图表；论文使用的缩写、程序全文及说明等。**

**致 谢**

**三号黑体加粗居中，上下各空一行**

正文内容

**中文小四号宋体，英文用小四号Times New Roman，首行缩进二个字，1.25倍行距。**

# 攻读学位期间发表的学术论文目录

**三号黑体加粗居中，上下各空一行**