交互式系统下基于任务重要性的 DVFS 技术①

刘腾福[1,2], 朱宗卫[1,2], 吕 良[2], 孙贝磊[1,2], 周学海[1,2], 李 曦[1,2]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230026)
[2](中国科学技术大学 苏州研究院, 苏州 215028)

摘 要: 动态电压频率调节技术(DVFS)是从软件层面进行系统功耗管理的重要技术. 本论文针对交互式系统的特点, 首先分析了当前使用的 DVFS 策略的不足之处, 然后提出并实现了一种适用于交互式系统的 DVFS 策略. 该策略在保障用户体验的前提下对移动设备进行功耗优化. 实验结果证明, 对于大多数应用能达到10%以上的功耗优化效果, 部分应用最高有超过 30%的功耗降低.

关键词: 交互式系统; DVFS; 功耗优化; 行为分析; Android

Load Importance-Based DVFS Scheme for Interactive Systems

LIU Teng-Fu[1,2], ZHU Zong-Wei[1,2], LV Liang[2], SUN Bei-Lei[1,2], ZHOU Xue-Hai[1,2], LI Xi[1,2]

[1](Institute of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)
[2](Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215028, China)

Abstract: Dynamic voltage and frequency scaling technique (DVFS) has been introduced to manage system power at software level. In this paper, we firstly discussed the disadvantages of commercial CPU frequency governors on interactive systems. Then, we proposed and implemented a new DVFS scheme aimed at Interactive Systems. The scheme is designed to save system power based on the premise that user experience must not be declined. Experimental results demonstrated that most application can save more than ten percent power compared with the original Ondemand DVFS governor, and even more than thirty percent for some other applications.

Key words: interactive system; DVFS; power; task behavior; Android

1 引言

随着现在计算机、智能手机等计算设备 CPU 核数的不断增加以及 CPU 频率不断上升, CPU 的计算能力得到了快速的提升. 伴随着计算能力提升的是 CPU 功耗的增加. 动态电压频率调节技术(DVFS)正是为了降低 CPU 的功耗提出的. 在支持 DVFS 技术的平台上, 操作系统可以根据当前计算机的状态适当调节 CPU 的运行频率, 在不需要高计算能力的情况下降低 CPU 频率以达到降低 CPU 功耗的目的.

1.1 常用 DVFS 策略分析

当前商用的 DVFS 策略主要分为两种, 分别是静态和动态调频. 静态策略即将 CPU 的频率静态地设定为某一预定的频率; 动态调频策略有 Ondemand 策略和 Conservative 策略[1]. 动态调频策略都使用 CPU 负载(CPUload, 即一段时间内 CPU 非 idle 时间和总时间的比值)指导调频, 其目标是尽量将 *CPU load* 维持在某一预定范围, 当 *CPUload* 超出其上限时提升 CPU 频率, 反之降低频率. 以 Linux 主线内核中使用的 Ondemand 策略为例[2](图 1), 算法的目的是将 *CPUload* 维持在 70%-80%之间. Ondemand 的特点是当 *CPUload* 超出上限时立刻将频率提升到最大值, 小于下限时逐步降低频率, 即“急速升频, 缓慢降频”策略. Conservative 策略和 Ondemand 类似, 但是其特点是“急速降频, 缓慢升频策略”.

这种基于 CPU 负载的 CPU 频率调节算法符合通用计算机工作的正常逻辑: 当 CPU 忙碌时频率上升,

① 基金项目:国家自然基金(61272131,61202053);江苏省自然科学基金(SBK201240198);江苏省产学研前瞻性联合研究项目(BY2009128)
收稿时间:2014-03-18;收到修改稿时间:2014-05-16

当 CPU 空闲时频率下降. 然而 Ondemand 算法也被广泛使用于 Android 等交互式操作系统. 交互式系统的设计目标和通用计算机系统有很大的不同之处, 交互式系统设计主要追求的目标是向用户提供一流的用户体验, 准确、快速地响应用户的操作. 因此, 在交互式系统中, 对于与用户体验关系密切的任务必须尽可能优先快速处理, 而对于用户不是非常关心响应时间的任务, 系统可以延迟处理或者延长处理时间.

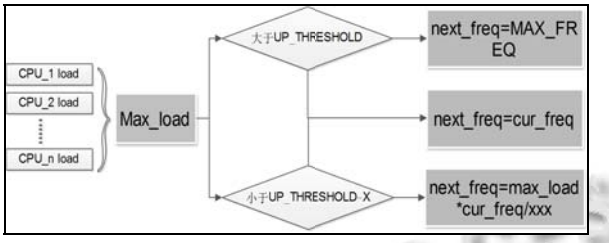

图 1 Ondemand 调频算法示意图

例如后台周期性邮件同步、天气数据的更新操作, 这种任务处理时间延长若干秒或者延迟若干秒处理并不会对用户体验带来严重的损害, 但是当前基于 CPU 负载的 Ondemand 策略可能会在执行该类操作时由于监测到 CPU 负载较高而将 CPU 频率调至最高. 这种“误调频”发生的原因在于 Ondemand 策略没有根据当前处理任务的特点进行调频.

1.2 国内外相关研究工作

为了提高系统交互时的性能, 最新的 Android 系统引入了 interactive governor[3]. 这种 CPU 调频策略假设用户的交互会导致 CPU 从 idle 状态切换到非 idle 状态. 基于这种假设, 系统在 CPU 从 idle 状态切换到非 idle 状态时对系统的负载进行检测以便在用户交互时迅速提升 CPU 频率. 这种策略能够在用户点击屏幕时及时提升 CPU 频率, 但是, 首先, 当用户在 CPU 非 idle 状态进行交互时, interactive 策略不能立刻提升 CPU 频率; 而且, interactive 策略仍然使用基于负载的策略进行 DVFS, 无法避免与用户交互无关的负载导致的 CPU 频率无谓提升.

国内外有不少研究者针对交互式系统 DVFS 进行了研究. 在文献[4]中, 作者通过监测用户输入时产生的事件以及 UI 画面是否正在变化来识别用户交互的开始和结束阶段, 从而识别当前系统是否正在处理用户交互相关的任务. 根据识别结果决定是否提升 CPU 频率. 该方法能够在一部分场景下适用, 但是使用画面不再变化作为一次交互任务的技术标志在很多情况下并不合理. 论文[5]通过分析用户的历史输入信息对下一次用户的交互进行预测, 从而根据预测结果对 CPU 频率进行调节. 这种方法需要较为复杂的算法支持, 而且预测的准确性也随着用户的不同会有差异, 不适合推广到商业系统中. 在文献[6]中, Mallik 等提出了“用户驱动”的 DVFS 策略. 该策略基于的假设是当用户感觉到系统性能不足时会不断地点击屏幕. 这样的策略对于一部分用户有较好的效果, 但是对于其他没有类似使用习惯的用户效果并不理想.

1.3 基于任务重要性的 DVFS

由于交互式系统的特殊性, 系统中任务的重要性随着它和用户体验的关系程度的不同而不同. 与响应用户操作相关的任务应当优先快速处理, 用户关注程度低的后台任务可以使 CPU 在较低的频率下运行. 基于交互式系统的特殊性以及当前基于 CPU 负载的调频技术的不足之处, 我们首次提出了一种基于任务重要性的 CPU 动态频率调节策略. 任务的重要性定义为与用户体验的关系密切程度, 与用户体验关系越密切的任务重要性越高. 这种 CPU 频率调节技术的目的是当 CPU 在处理与用户体验关系密切的任务时提高 CPU 的运行频率, 反之降低 CPU 的运行频率.

基于任务重要性的 DVFS 策略在当前基于 CPU 负载的调频策略的基础上引入任务重要性指标对任务的特点进行刻画, 充分结合了交互式系统提升用户体验的设计宗旨, 当且仅当 CPU 在处理会影响用户交互响应时间的任务时提升 CPU 的处理频率, 从最大程度上降低了交互式系统的功耗开销.

本论文将在第二节中提出 Android 系统中任务重要性指标的数学模型; 第三节描述使用任务重要性指标改进当前基于 CPU 负载的 DVFS 策略的算法; 我们将在第四节讲述该技术在真实手机中的实验方法及其结果; 第五节将对本论文进行总结.

2 任务重要性评价指标

使用任务重要性指导 DVFS, 首先需要量化任务重要性的评价标准. 本文提出了一种简单有效的、归一化的任务重要性评价数学模型. 该模型充分利用 Android 的消息处理与用户交互的关系对任务重要性进行评价.

2.1 Android 消息处理机制

Android 系统的应用程序是消息驱动的[7]. 当系统中有需要应用程序处理的消息时, Android 相应应用程序的相关线程便被唤醒进行消息处理; 消息处理结束以后, 该线程就进入睡眠等待状态, 直到产生新的消息出现为止.

根据对 Android 系统的消息处理机制的分析以及对 Android 应用程序编程模型的研究可知, Android 应用程序的线程分为主线程和子线程. Android 系统规定与界面相关的操作只可以在主线程中执行[2]. 主线程主要负责处理与用户交互界面相关的事务. 主线程在启动完成之后就会创建一个消息队列并且主动进入到一个无限循环当中等待、处理消息.

2.2 任务重要性衡量指标

刻画任务的重要性即使用一种数学模型量化任务与用户体验的相关程度. 在此, 我们引入了“消息处理时间占空比 ρ”作为衡量任务重要性的指标, ρ 定义如下:

$$\rho = \frac{msg\_time}{cpu\_time} \tag{1}$$

式中 *cpu_time* 和 *msg_time* 分别表示系统每两次调用调频算法对 CPU 进行频率调节的时间间隔内 CPU 用于处理任务的时间(非 Idle 时间)和 CPU 用于主线程处理消息的时间.

消息处理时间占空比 ρ 不仅可以表示 CPU 在一段时间内用于处理消息的时间的多少, 而且提供了一种便于比较的归一化的参考量. ρ 等于 0 表示在过去的一段时间内没有处理主线程消息, ρ 等于 1 表示过去的一段时间内都是在处理与主线程消息有关的任务.

通过 2.1 节对 Android 消息处理机制的分析可以看出, 用户的交互产生的消息都必须由应用程序的主线程进行处理. 所以, Android 应用程序主线程消息处理时间的多少可以反应出用户与系统的交互程度: 当 Android 应用程序的主线程消息处理时间长时, 说明用户与系统交互频繁, 当前系统中的任务很可能关系到用户交互的响应时间; 反之, 则说明用户与系统交互较少, 当前 CPU 可能在处理和用户交互无关的后台任务. 因此, 主线程的消息处理时间占空比是衡量任务重要性最好的指标.

2.3 消息处理时间占空比的计算

本节将介绍如何在 Android 系统中计算任务重要性评价指标“消息处理时间占空比”.

2.3.1 cpu_time 计算

Linux 系统中有一个 Per-cpu 的全局变量 *kernel_cpustat*, 该变量用于记录系统中每个 CPU 自从系统启动以来运行于用户态、内核态、中断、Idle 等状态的时间, get_cpu_idle_time 函数即根据该变量计算出自从系统启动以来 CPU 的 Idle 时间(*idle_time*). 要计算公式一中 *cpu_time* 的值, 只需要在每次调用调频算法时获取当前的 *idle_time* 和墙上时钟(*wall time*)即可:

$$cpu\_time=(cur\_wall\_time-prev\_wall\_time)-(cur\_idle\_time-prev\_idle\_time)$$

其中 *cur_wall_time* 和 *cur_idle_time* 分别表示当前墙上时钟和当前 Idle 时间, *prev_wall_time* 和 *prev_idle_time* 分别表示上一次调用 CPU 调频算法时的墙上时钟和当前 Idle 时间. 计算结束以后将 *cur_wall_time* 和 *cur_idle_time* 分别记录到 *prev_wall_time* 和 *prev_idle_time* 即可.

2.3.2 *msg_time* 计算

计算一段时间内用于处理主线程消息的时间总和需要正确处理如下两个问题: 第一, 如何准确地获取消息处理的开始和结束时间点; 第二, 如何正确处理当正在处理消息的线程被调度器切换出 CPU 的情况.

在 Android 系统中, 消息处理的开始和结束时间点可以从 Framework 层中进行截获. 在 Looper 类的 loop 方法中, 当线程的消息队列中有未处理的消息时, 线程从其消息队列中取出队头的消息, 然后调用 Message.target.dispatchMessage(msg) 函数对新消息 msg 进行处理. 因此, 只要在调用该函数的前面和后面加入采样点即可对消息处理的开始和结束进行追踪.

AndroidMsgNotify 函数通过在内核中新添加的专用系统调用将用户态消息处理开始和结束的信息传递到内核. 在 Framework 的 Native 层也有相应的消息处理函数, 使用类似的方法即可在 C++层将消息处理开始和结束的信息传递到内核.

在多任务的系统中, CPU 通过调度器由多个进程分时共享, 这样就使得每一个消息实际处理的时间长度并不一定等于消息处理结束时刻同开始时刻的时间差. 由于 CPU 共享而导致的情况包括三种. 第一: 在进程被切换出 CPU 时没有正在处理的消息; 第二: 进程在被调度出 CPU 时正在处理消息, 下一次被调度执行时还在原来的 CPU 上运行; 第三种情况同第二种类

似, 差别在于下一次被调度到另一个 CPU 上运行. 如图 2 所示, 图中阴影部分代表处理消息, 深黑色和灰色背景的方框分别代表进程在不同的 CPU 上运行.

考虑到上述各种情况, 我们为每个 CPU 设置了一个全局变量 *last_schedule_time* 用于记录每个 CPU 上一次发生调度的时间. 另外, 在每个进程对应的 “进程描述符表”即 Linux 内核中的 task_struct 结构体中增加变量 *is_processing_msg* 和 *msg_start_time*, 分别表示该线程是否正在处理消息和开始处理消息的时间.

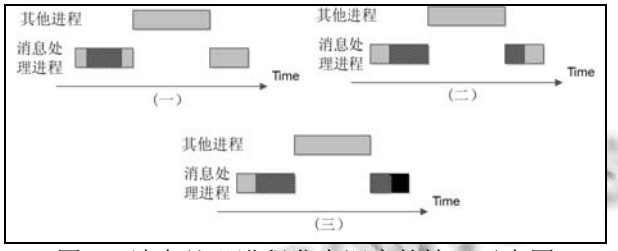

图 2 消息处理进程发生调度的情况示意图

统计消息处理时间总和要在消息处理开始、结束和上下文切换时对这些变量进行计算, 算法示意图如图 3 所示.

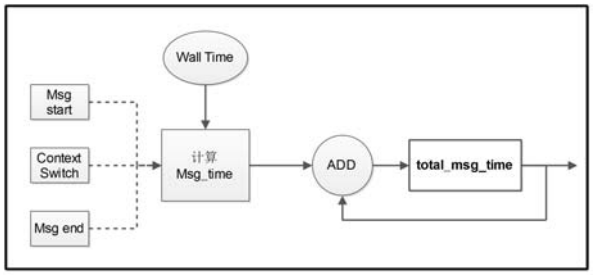

图 3 消息处理时间统计算法示意图

当 Framework 层通过系统调用通知内核当前线程开始处理一个新的消息时, 如果当前线程是主线程, 则将 *is_processing_msg* 赋值为 1, 同时将当前时间记录到 *msg_start_time*.

当内核被通知当前线程的消息处理结束或者内核发生调度时, 如果当前线程是主线程, 则比较 task_struct 中的 *msg_start_time* 值和本 CPU 的 *last_schedule_time* 值, 如果前者较大, 则说明属于情况一, 本次消息处理时间 *this_msg_time* = *now* – *msg_start_time*; 否则, 属于情况二或者三, *this_msg_time* = *now* – *last_schedule_time*. 最后再将 *this_msg_time* 累加到当前 CPU 对应的 *msg_time* 中, 如果是消息处理结束, 则将 *is_processing_msg* 清零, 如果是发生了调度, 则将 *last_schedule_time* 更新为当前时间.

这样, 在多任务多 CPU 的系统上就可以正确地统计任意一段时间内各个 CPU 用于处理主线程消息的时间总和了.

3 基于任务重要性指标的DVFS算法

当前 Android 中使用的动态 DVFS 调频策略是根据上一时段 CPU 负载的值进行决策的, 调频算法得出的下一时段 CPU 的频率由 *max_load* 决定(如图 1 所示). 基于负载的 CPU 调频算法基于的假设是: 负载值越高, CPU 越忙碌, 对 CPU 计算能力的 “期望值”越大.

由前文分析可知, “消息处理时间占空比 ρ”是一种衡量一段时间内 CPU 处理的任务的重要性的指标. 结合基于负载的 CPU 调频算法的基本假设, 我们在基于负载的 CPU 调频算法计算 CPU 负载时使用缩放因子 k 对 CPU 负载进行缩放. 对于重要性越高的负载, k 的值越大, 反之, k 的值越小. 因此, k 是 ρ 的函数, k=F(ρ). 函数 F(x)可以根据平台的不同制定不同的实现方式, 且需要保证程序运行时不会出现 “卡顿”现象. 当ρ较大时, 对原始计算得到的 cpu_load 乘以一个较大的数进行放大, 反之乘以一个较小的数进行缩小. 由于在基于负载的调频算法中, *cpu_load* 的值越大, 算法计算得出的 CPU 频率越高, 因此, 我们通过修改 *cpu_load* 的计算方法 “间接”改变了系统对下一时段 CPU 频率的计算, 达到了基于任务重要性指标对 CPU 进行 DVFS 的目的, 使得任务越重要(ρ 值越大), DVFS 算法将 CPU 频率设置得越高.

使用这种方法, 我们只要对系统作非常少的修改即可实现基于任务重要性指标的 DVFS 算法.

在实验中, 我们根据平台的特点, 使用如图 4 所示的算法实现 ρ 和 k 之间的映射. 当 ρ 大于 *threshold*%时说明消息处理较多, k 取最大值 *top*%; 当 ρ 在 0%-*threshold*%之间时 k 与 ρ 呈线性关系, 即:

$$k = \alpha * \rho + bottom\%$$

bottom%等于最小缩放因子, 用于保证性能, *top*%等于最大缩放因子, 用于控制功耗开销.

对于不同的平台, *threshold*、*bottom* 和 *top* 三个参数的选择各不相同. 表 1 显示了本论文用于实验的两个平台的参数.

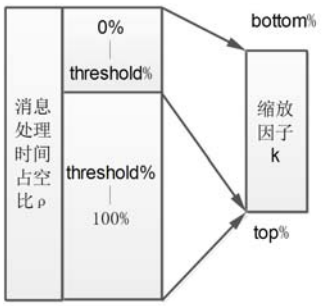

图 4 ρ 和缩放因子 k 的映射函数示意图

表 1　实验参数列表

平台	Threshold	Bottom	Top
Odroid-PC	30	70	120
I9500	30	70	100

4　实验

4.1 实验平台和设备

为了验证基于任务重要性的 DVFS 技术, 我们首先在开发板 Odroid-PC 上进行了实验. 该开发板与三星手机 I9100 具有相同的配置, 是 I9100 的实验平台. 其主要参数如表 2 所示.

表 2　Odroid-PC 开发板主要参数

CPU	Exynos4210 Cortex-A9 Dualcore 1.2Ghz
Cache	1MB L2 cache
内存	1024MB LP DDR2

Samsung GALAXY S4(I9500)是三星公司的旗舰产品, 搭载的是 Exynos 5410 双四核处理器, 包括四颗主频 1.6GHz 的 Cortex-A15 核和四颗主频 1.2GHz 的 Cortex-A7 核. 其主要参数如表 3 所示.

表 3　Exynos 5410 主要参数列表

	big core	LITTLE core
体系结构	ARM Cortex-A7	ARM Cortex-A15
制造工艺	Samsung 28nm HKMG	Samsung 28nm HKMG
频率	200MHz~1.8GHz+	200MHz~1.2GHz+
面积	19 mm^2	3.8 mm^2
能耗系数	1	0.17
L1 Cache	32KB I/D cache	32KB I/D cache
L2 Cache	2MB Data cache	512KB Data cache

I9500 作为八核手机的代表, 具有强大的计算能力, 能够带来流畅的用户体验, 但是八核 CPU 同时也带来强大的功耗开销, 玩游戏、看视频时普遍大于 500mA 的电流很快就能将电池消耗殆尽. 因此, 为了在实际平台上验证基于任务重要性的 DVFS 技术的功耗优化效果, 我们还在 I9500 上进行了实验.

在实验中, 我们使用工业界广泛使用的能耗测量工具 Power Monitor[8]对手机的能耗进行测量. Power Monitor 可以实时显示监控设备的当前功率, 并可以计算出一段时间内的总功耗及平均功率.

4.2 实验用例

在实验中, 我们选取了一组热门的游戏作为我们的实验用例. 因为游戏既能很好的反应系统的响应速度, 同时又是功耗较大的应用. 我们使用的测试用例如表 4 所示.

表 4　实验用例列表

实验用例名称	实验用例类型
AngryBots	游戏
NBA 2K14	游戏
Tank 3D	游戏

4.3 实验结果

为了保证实验场景的一致性, 每个实验用例的每一次测试都是重新启动手机并且待系统稳定后进行的. 这样可以最大限度减小系统中其他进程对本实验造成的不确定性干扰. 另外, 我们对每个游戏分别测试了多次以消除随机误差带来的影响.

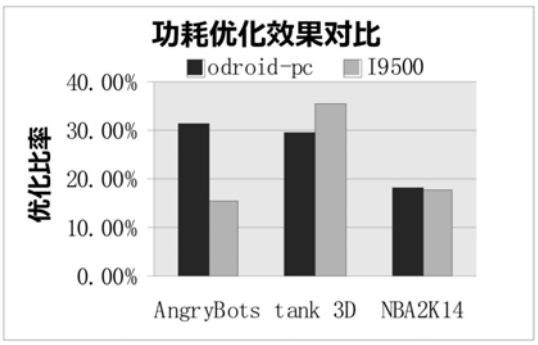

图 5　各应用程序在不同平台下功耗优化对比图

图 5 表示了分在 Odroid-PC 和 I9500 平台下, 基于任务重要性的 DVFS 策略和 Android 原始调频策略 Ondemand 算法相比各个应用的功耗优化效果. 从图中可以看出, 在引入任务重要性的评价指标以后, 系统的功耗得到了明显的下降. 其中在 I9500 上效果最明显的 tank 3D 游戏在原始 Ondemand 调频策略下的平均电流为 744mA, 在基于任务重要性的调频策略下平均电流为 478mA, 下降了约 36%, 下降幅度最小的 AngryBots 游戏也达到了 15%的功耗优化效果. 在 Odroid-PC 平台上效果也很明显, 其中 AngrBots 的功

耗下降幅度最大, 为 31.7%; 而最小的 NBA2K14 功耗优化比例为 18.98%.

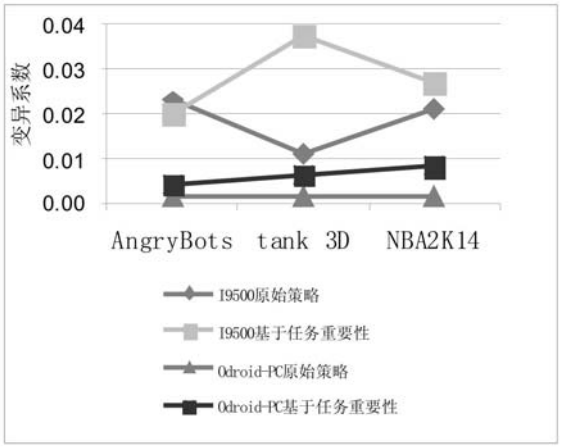

图 6 各组实验结果的变异系数

图 6 显示了实验结果中各组实验数据的变异系数值[9]. 从图中可以看出, 在 I9500 平台上每一组实验的三次实验结果相差幅度非常小, 最大的标准差也不超过期望的 4%; 而在 Odroid-PC 上由于受到的干扰比实际手机中小很多, 因此变异系数也比 I9500 小一个数量级. 这一数据说明上述功耗对比实验数据的可信度较高, 随机因素的干扰较小.

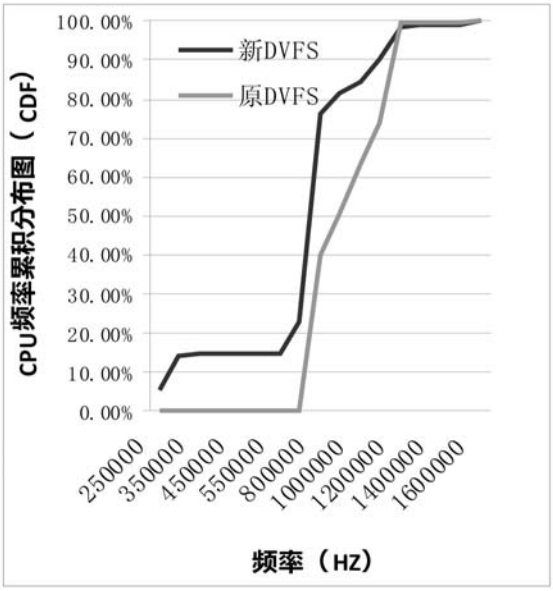

图 7 AngryBots 应用 CPU 频率累积分布图

通过对各个应用程序运行时 CPU 频率分布的跟踪统计可以发现, 基于任务重要性的 CPU 调频技术使得 CPU 频率分布向低频端移动了一段. 图 7 表示在运行 AngryBots 时 CPU 频率的累积分布函数(CDF)曲线图[10]. 从图中可以看出, 在新 DVFS 策略下运行 AngryBots 的 CPU 频率带比在原始 DVFS 策略下运行时低.

这种 CPU 频率带的左移正说明了新的 DVFS 策略使用了更低的 CPU 频率, 进而取得了较好的功耗优化效果. 证明了基于任务重要性的 DVFS 技术在 Android 平台上功耗优化效果显著.

5 结语

本论文针对交互式系统的特点, 提出了一种基于任务重要性的 CPU 频率调节技术. 论文以与用户体验的相关程度定义了交互式系统上任务的重要性, 并且使用 UI 线程的消息处理时间占空比对 Android 系统上的任务重要性进行了量化描述. 根据我们目前的调研结果来看, 这方面的工作是我们首先提出并在实际平台中进行验证有效的.

下一阶段我们将延伸任务重要性的概念到计算机其他资源的分配领域中, 例如通过评价网络请求同用户体验的关联程度指导网络设备的状态控制和响应请求的算法优化, 以期提高用户操作的响应速度, 同时对网络设备的功耗进行优化.

参考文献

1 https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
2 Pallipadi V, Starikovskiy A. The ondemand governor. Proc. of the Linux Symposium. 2006, 2.
3 https://android.googlesource.com/kernel/common/+/android-3.4%5E/Documentation/cpu-freq/governors.txt
4 Kim S, Kim H, Hwang J, et al. An event-driven power management scheme for mobile consumer electronics. IEEE Trans. on Consumer Electronics, 2013, 59(1): 259–266.
5 Zhong L, Jha NK. Dynamic power optimization targeting user delays in interactive systems. IEEE Trans. on Mobile Computing, 2006, 5: 1473–1488.
6 Mallik A, Lin B, Memik G, et al. User-driven frequency scaling. Computer Architecture Letters, 2006, 5(2): 16.
7 罗升阳.Android 系统源代码情景分析.北京:电子工业出版社,2012.
8 http://www.msoon.com/LabEquipment/PowerMonitor/
9 http://zh.wikipedia.org/wiki/变异系数
10 http://en.wikipedia.org/wiki/Cumulative_distribution_function