

# 抢占阈值调度的功耗优化

贺小川 贾 焰

(国防科学技术大学计算机学院网络与信息安全研究所 长沙 410073)

**摘 要** DVS(Dynamic Voltage Scaling)技术的应用使得任务执行时间延长进而使得处理器的静态功耗(由 CMOS 电路的泄露电流引起)迅速增加. 延迟调度(Procrastination Scheduling)算法是近年提出用于减少静态功耗的有效方法,它通过推迟任务的正常执行来尽可能长时间地让处理器处于睡眠或关闭状态,从而避免过多的静态功耗泄露. 文中针对可变电压处理器上运用抢占阈值调度策略的周期性任务集合,将节能调度和延迟调度结合起来,提出一种两阶段节能调度算法,先使用离线算法来计算每个任务的最优处理器执行速度,而后使用在线模拟调度算法来计算每个任务的延迟时间,从而动态判定处理器开启/关闭时刻. 实例研究和仿真实验表明,作者的方法能够进一步降低抢占阈值任务调度算法的功耗.

**关键词** 动态电压调整;抢占阈值调度;延迟调度;阈值分配;实时系统

**中图法分类号** TP312 **DOI号**: 10.3724/SP.J.1016.2008.02060

## Leakage-Aware Energy-Efficient Scheduling for Fixed-Priority Tasks with Preemption Thresholds

HE Xiao-Chuan JIA Yan

(Institute of Network Technology and Information Security, School of Computer,  
National University of Defense Technology, Changsha 410073)

**Abstract** Dynamic Voltage Scaling (DVS), which adjusts the clock speed and supply voltage dynamically, is an effective technique in reducing the energy consumption of embedded real-time systems. However, the longer a job is executed, the more energy in the leakage current the device/processor consumes for the job. Procrastination scheduling, where task execution can be delayed to maximize the duration of idle intervals by keeping the processor in a sleep/shutdown state even if there are pending tasks within the timing constraints imposed by performance requirements, has been proposed to minimize leakage energy drain. This paper targets energy-efficient fixed-priority with preemption threshold scheduling for periodic real-time tasks on a uni-processor DVS system with non-negligible leakage power consumption. A two-phase algorithm is proposed. In the first phase, the execution speed, i. e., the supply voltage of each task are determined by applying off-line algorithms, and in the second phase, the procrastination length of each task is derived by applying on-line simulated work-demand time analysis, and thus the time moment to turn on/off the system is determined on the fly. A series of simulation experiments was evaluated for the performance of our algorithms. The results show that the proposed algorithms can derive energy-efficient schedules.

**Keywords** Dynamic Voltage Scaling (DVS); Fixed Priority with Preemption Threshold (FPPT); procrastination scheduling; preemption threshold assignment; real-time system

## 1 引言

随着集成电路规模的飞速发展和设计工艺的日趋复杂,能量消耗问题越来越受到关注,尤其在嵌入式移动计算技术日益普及的今天,嵌入式移动设备往往依靠电池供电,电池的供电时间成为系统性能的重要指标,而且与半导体技术的发展相比较,电池技术的发展要缓慢得多,因此,如何在保持系统实时性能的前提下尽可能降低系统能量损耗成为热点问题. 动态电压调整(Dynamic Voltage Scaling, DVS)是一种新型硬件节能机制. 动态电压调整的核心思想是,在满足任务完成时限要求的前提下,在程序运行时动态调节处理器电压和频率,使得处理器不总以最高电压工作,从而起到降低能耗的目的.

随着移动嵌入式市场的日趋成熟,许多商用可变电电压嵌入式微处理器大量涌现. 目前绝大多数嵌入式处理器都用 CMOS 技术制造,而且支持多种处理器频率和电压设置. CMOS 电路的功耗正比于时钟频率和电压的平方<sup>[1]</sup>,即每个时钟周期的能量消耗正比于电压的平方. 对于一个任务来说,完成它所需要的时钟周期是固定的,所消耗的能量与电压的平方成正比,通过降低电压就可以降低能耗. 但由于时钟频率与电压之间的线性关系<sup>[1]</sup>,降低电压会调低时钟频率,增加任务的完成时间,许多针对强实时系统的节能调度(Energy-Efficient Task Scheduling)算法就动态调整处理器电压,在恰好满足任务实时需求的前提下尽可能降低系统能耗<sup>[2-10]</sup>.

几乎每一种使用 DVS 技术的节能调度算法<sup>[2-10]</sup>在降低系统能耗/功耗方面都很有效,但是现有节能调度算法将注意力主要集中在处理器的动态能耗上,却忽略了由处理器 CMOS 泄露电流造成的静态能耗. DVS 技术在调低处理器电压/频率、使得任务执行时间延长从而降低动态能耗时,处理器的静态能耗却随之增加. 而在嵌入式实时系统中,嵌入式微处理器的静态能耗很难忽略不计. 延迟调度(Procrastination Scheduling)算法是近年提出用于减少静态能耗的有效方法,它通过推迟任务的正常执行来尽可能长时间地让处理器处于睡眠或关闭模式(dormant mode),从而避免过多的静态能耗泄露<sup>[11-16]</sup>.

在各种传统的实时系统中,为了尽可能提高系统利用率,抢占式调度得到普遍重视和广泛应用. 但是随着计算环境逐渐向普适计算发展,实时系统所

应用的平台往往不再是大型复杂的系统,而是微型简单的嵌入式系统. 而随着嵌入式系统资源的限制愈加严格,抢占式调度所带来的任务切换次数过多、内存消耗过大等不足之处,已经难以忽略. 如何在降低系统可调度性的前提下尽可能减少不必要的任务切换,从而降低内存需求,在嵌入式实时计算环境下逐渐成为新的研究热点. 为此,文献[17-18]提出抢占阈值调度算法(Fixed-Priority tasks with Preemption Thresholds scheduling, FPPT),每个任务  $\tau_i$  不仅分配任务优先级  $\pi_i$ , 还分配抢占阈值  $\gamma_i$ ,  $\gamma_i \geq \pi_i$ ,  $\pi_i$  用于竞争处理器的使用权,而  $\gamma_i$  是实际执行时的任务优先级. 此时,抢占式调度和非抢占式调度就是抢占阈值调度的两个特例,而抢占阈值调度通过调节任务的抢占阈值,减少不必要的现场切换次数,同时保持任务间的部分可抢占性来确保任务实时性得到满足,从而提高整个任务集合的可调度性<sup>[17-18]</sup>.

本文针对在支持睡眠模式的可变电电压处理器上实施 FPPT 调度策略的周期性实时任务集合,首先给出一种基于响应时间分析(Response Time Analysis)的任务可调度性判定方法,而后证明系统在最大阈值分配的情况下功耗最低,然后将节能调度算法和延迟调度算法结合起来,提出一种两阶段节能调度算法,第一阶段是基于任务最长执行时间离线计算每个任务可能使用的最优处理器速度,第二阶段是在线计算每个任务的最大可能延迟距离(procrastination interval),而后决定在什么时刻让处理器进入睡眠模式,从而同时降低系统动态功耗和静态功耗. 理论分析表明,该方法所带来的系统整体能耗最多是最优调度所带来能耗的  $\max\{1/(U_{bd})^2, 2\}$  倍,其中  $U_{bd}$  是使用 FPPT 调度策略的任务集合的临界过载利用率(breakdown utilization)<sup>[19]</sup>. 一系列仿真实验也表明我们的方法能够进一步降低系统功耗.

## 2 系统模型和问题定义

### 2.1 任务模型

本文的研究对象是强实时系统中使用 DVS 技术的静态优先级任务集合,使用周期性任务模型,任务集合  $T = \{\tau_1, \tau_2, \dots, \tau_n\}$ ,  $J_{i,j}$  表示任务  $\tau_i$  的第  $j$  个实例. 周期性任务  $\tau_i$  使用 3 元组  $(C_i, T_i, D_i)$  表示,其中  $C_i$  表示任务执行所需要的时钟周期个数,  $D_i$  表示任务的截止时限,  $T_i$  表示两个任务  $\tau_i$  实例到来的最

小间隔. 任务集合  $T$  的超周期(hyper-period)的长度用  $L$  表示,  $L$  是  $T$  内所有任务周期的最小公倍数,  $L/T_i$  表示任务  $\tau_i$  在超周期  $L$  内能够执行的任务实例个数. 每个任务  $\tau_i$  都分配了任务优先级  $\pi_i$  和抢占阈值  $\gamma_i$ ,  $\pi_i, \gamma_i \in [1 \cdots n]$ ,  $\gamma_i \geq \pi_i$ . 任务优先级的大小决定了任务竞争处理器或其他资源能力的强弱, 任务抢占阈值的大小决定了它被更高优先级任务抢占的可能性的.  $T$  的任务优先级分配  $\Pi$  和抢占阈值分配  $\gamma$  实质上是映射  $T \rightarrow [1, 2, \dots, n]$ , 实际的映射结果由具体的分配策略决定.

这里我们假定  $T$  内所有任务都是独立的, 任务间没有共享资源. 任务除非执行完毕, 在任务执行过程中不会自行中止.

## 2.2 处理器功耗和执行模型

本文的工作是在可变电压处理器上考察实时任务调度问题, 处理器的功耗主要来自 CMOS 门电路频繁充放电所导致的动态功耗和由 CMOS 泄露电流导致的静态功耗组成. 可变电压处理器的动态功耗  $P_d$  可表示为处理器频率的函数:

$$P_d = C_{\text{eff}} V_{\text{DD}}^2 f \quad (1)$$

$$f = \alpha k' \frac{(V_{\text{DD}} - V_{\text{TH}})^2}{V_{\text{DD}}} \quad (2)$$

嵌入式实时系统的静态功耗  $P_s$  主要来自于处理器、I/O 设备和 RAM 内存的泄露电流, 这里主要考虑处理器的静态功耗. 处理器的整体功耗表示为  $P$ ,  $P = P_d + P_s$ , 本文考虑  $P(f)$  为非递减凸函数的情况, 此时,  $P(f)/f$  也是凸函数<sup>[11, 15]</sup>. 当前可变电压处理器往往提供多个可设置的电压/频率级别, 这里假定处理器可用的频率集合为  $\{FLK_1, FLK_2, \dots, FLK_s\}$ , 相应的可用电压集合为  $\{v_1, v_2, \dots, v_s\}$ . 我们假定任务  $\tau_i$  的处理器频率  $f_i$  可在最低频率  $FLK_1$  和最高频率  $FLK_s$  之间变化, 我们提出的算法只是在任务切换时间内完成处理器电压/频率的调整动作; 如果该时间与任务执行时间相比实在无法忽略, 就将该段时间加入任务负载内作为任务最长执行时间考虑.

处理器在必要的时候可进入睡眠模式, 在该工作模式下处理器的功耗可近似为 0. 我们假定处理器可在任何时刻关闭(进入睡眠模式)和重新启动(进入活跃模式), 而且将处理器关闭能够进一步降低功耗. 这里假定关闭处理器的功耗是可忽略的, 但是将处理器重新开启的功耗是不可忽略的<sup>[20]</sup>. 处理器从睡眠模式进入活跃模式所需能耗表示为  $E_{\text{sw}}$ , 在本文中, 如果处理器在时刻  $t$  不执行任何任务, 我

们就称系统在时刻  $t$  处于“空闲状态”. 当系统处于空闲状态而处理器处于活跃模式时, 处理器执行空指令而且处理器频率设定为  $FLK_1$ , 此时的处理器功耗表示为  $P_l$ ,  $P_l = P(FLK_1)$ .

这里我们假定嵌入式可变电压处理器的物理实现使用单核心处理单元. 尽管使用多个处理核心是处理器制造的发展趋势, 但是目前在嵌入式应用中使用多核处理器尚不多见.

## 2.3 问题定义

本文的研究问题是: 使用抢占阈值调度策略的静态优先级任务集合, 在可变电压处理器上如何实现泄漏敏感的节能调度 (Leakage-Aware Energy-Efficient Scheduling for FPPT, LAEES-FPPT). 问题定义如下.

**问题 1** (泄漏敏感的 FPPT 节能调度, 简称 LAEES-FPPT). 由  $n$  个在时刻 0 释放的任务组成的任务集合  $T$ ,  $T$  内每个任务  $\tau_i$  的执行需要  $C_i$  个时钟周期, 每个  $\tau_i$  实例的最小释放间隔时间为  $T_i$ ,  $\tau_i$  实例自释放时刻起必须在  $D_i$  之前完成. 同时  $T$  内每个任务  $\tau_i$  都分配了任务优先级  $\pi_i$  和抢占阈值  $\gamma_i$ ,  $\pi_i$  用于竞争处理器而  $\gamma_i$  用于  $\tau_i$  开始执行后保护  $\tau_i$  避免过多不必要的任务抢占. 处理器功耗  $P(f)$  是非递减的凸函数, 而  $P(f)/f$  只是凸函数. 处理器具有一系列离散的频率/电压设置,  $\{FLK_1, FLK_2, \dots, FLK_s\}$  和  $\{v_1, v_2, \dots, v_s\}$ . 处理器从睡眠模式进入活跃模式的额外能耗为  $E_{\text{sw}}$ , 而处理器处于活跃模式和空闲状态时的功耗为  $P_l$ . 问题在于如何在保证  $T$  可调度性的前提下使得超周期  $L$  内的系统功耗最小.

任务集合  $T$  的节能调度就是在保证  $T$  内每个任务  $\tau_i$  的实例在  $D_i$  之前完成的前提下为每个任务  $\tau_i$  分配处理器执行频率  $f_i$ , 如果这种分配没有影响  $T$  内每个任务的可调度性, 我们就称这种节能调度是可行的. LASSE-FPPT 的最优调度就是它的调度结果不仅是可行的, 而且它的调度结果使得系统整体功耗最小. 本文的后续部分用符号  $S^*$  表示 LASSE-FPPT 的最优调度.

对于任务集合  $T$  的某个调度结果而言, “空闲时段”是指系统处于空闲状态的最长时间间隔, 而“执行时段”则是指处理器正在执行任务的最长时间间隔. 处理器在空闲时段很可能会进入睡眠模式, 而处理器在执行时段肯定处于活跃模式, 如果某个调度结果的空闲时段长度大于  $E_{\text{sw}}/P_l$ , 那么在该时段开始时刻将处理器关闭在节能上就很有意义. 这里

用符号  $t_0$  表示该空闲时段长度, 即  $t_0 = E_{sw}/P_I$ ,  $t_0$  也被称为“空闲时段阈值”。

LASSE-FPPT 某个调度结果  $S$  在超周期内的能耗表示为  $E(S)$ .  $E(S)$  由两部分能耗组成: 执行能耗  $\phi(S)$  和空闲能耗  $\epsilon(S)$ . 其中执行能耗  $\phi(S)$  是调度  $S$  内的任务实例在超周期  $L$  的执行时段内能耗总和, 而空闲能耗  $\epsilon(S)$  是在超周期  $L$  的空闲时段内系统能耗. 用符号  $v(t, S)$  表示调度  $S$  在时刻  $t$  的处理器频率, 那么  $\phi(S)$  可表示为  $\int_0^L P(v(t, S))dt$ , 而  $\epsilon(S)$  则是  $E_{sw}$  乘以超周期  $L$  内的处理器重新开启的次数和  $P_I$  乘以超周期  $L$  内系统处于空闲状态而处理器处于活跃模式的时段长度。

### 3 可调度性分析

本节给出 LASSE-FPPT 功耗最小化问题的某些初步分析结果, 为下一节提出两阶段算法准备理论基础和分析方法。

#### 3.1 关键速度

所谓“关键速度”就是使得处理器功耗最低的处理器频率, 记为  $\hat{f}$ . 由于处理器功耗函数  $P(f)$  是凸函数, 那么在一个处理器时钟周期内使用一个恒定的处理器频率将使得该时钟周期内系统能耗最小. 因此, 处理器频率被设置为  $f$  时它的能耗可表示为  $P(f)/f$ .  $P(f)/f$  是凸函数, 那么不妨假定处理器频率为  $f^*$  时  $P(f)/f$  取得最小值, 此时  $d(P(f^*)/f^*)/df^* = 0$ . 所以, 要使得任务集合  $T$  的功耗最小, 各个任务不一定非得要在最低处理器频率下工作, 各个任务在执行时只需要将处理器频率调整为  $f^*$  即可。

如果  $f^* \in \{FLK_1, FLK_2, \dots, FLK_s\}$ ,  $\hat{f}$  就等于  $f^*$ ; 如果  $f^* \leq FLK_1$ , 那么出于平台的限制,  $\hat{f}$  只能设置为  $FLK_1$ ; 如果  $f^* \geq FLK_s$ , 那么  $\hat{f}$  就是  $FLK_s$ . 各个任务只需要使用最高处理器频率即可, 此时节能调度就没有太多意义了. 总之,  $\hat{f}$  可设置为  $\min\{\max\{f^*, FLK_1\}, FLK_s\}$ . 这里我们假定  $f^*$  是预先可知的。

#### 3.2 节能 FPPT 算法的可调度性分析

当前针对静态优先级任务的抢占阈值调度 (FPPT) 分析<sup>[17-18, 21-24]</sup> 都基于单个任务的最长响应时间分析, 这种响应时间分析方法是 level- $i$  忙周期分析方法<sup>[23]</sup> 的扩展, 该方法计算自 FPPT 关键时刻<sup>[23]</sup> 开始的忙周期内所有任务  $\tau_i$  实例的响应时间,

如果这些任务实例都在截止时限之前执行完毕, 则任务  $\tau_i$  就是可调度的. FPPT 关键时刻对于任务集合中每个任务都不同, 它是所有更高优先级任务实例和可能形成最长阻塞时间的更低优先级任务同时到来的时刻. 任务  $\tau_i$  的最长响应时间可使用如下公式推导:

$$B_i = \max_{\tau_j \in T} \{C_j/f_j \mid \pi_i > \pi_j \wedge \pi_i \leq \gamma_j\} \quad (3)$$

$$L_i = B_i + \sum_{\forall j, \pi_j \geq \pi_i} \left\lceil \frac{L_j}{T_j} \right\rceil \cdot (C_j/f_j) \quad (4)$$

$$S_i^k = B_i + k \cdot (C_i/f_i) + \sum_{\forall j, \pi_j \geq \pi_i} \left(1 + \left\lfloor \frac{S_j^k}{T_j} \right\rfloor\right) \cdot (C_j/f_j) \quad (5)$$

$$F_i^k = S_i^k + C_i/f_i + \sum_{\forall j, \pi_j \geq \pi_i} \left( \left\lceil \frac{F_j^k}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S_j^k}{T_j} \right\rfloor\right) \right) \cdot (C_j/f_j) \quad (6)$$

$$R_i = \max_{k=0,1,\dots,\lfloor L_i/T_i \rfloor} (F_i^k - k \cdot T_i) \quad (7)$$

任务  $\tau_i$  的响应时间包括 3 部分: (1)  $\tau_i$  自身的计算时间 (最差情况下)  $C_i/f_i$ ; (2) 来自其他更高优先级或相同优先级任务实例的干预时间, 用  $I_i$  表示; (3) 来自低优先级任务的阻塞时间, 用  $B_i$  表示. 方程(3)用于计算任务的最大阻塞时间, 方程(4)计算 level- $i$  忙周期的长度, 方程(5)计算任务实例  $J_{i,k}$  的开始执行时间, 方程(6)计算任务实例  $J_{i,k}$  的结束执行时间, 方程(7)计算  $\tau_i$  的最长响应时间。

基于可调度性分析方法, 文献[17]提出一种为各个任务分配抢占阈值的算法, 并声称该算法是最优的. 文献[21, 23]提出如果某个静态优先级任务集合在抢占阈值策略下是可调度的, 那么该任务集合应该存在多个可行抢占阈值分配, 而且所有这些阈值分配之间可定义某种偏序关系, 这种偏序关系使得所有可行阈值分配组成一个解空间, 这个解空间存在最大下界和最小上界, 即最小阈值分配和最大阈值分配. 文献[17]中的算法所得到的阈值分配实际上是最小阈值分配, 文献[21, 23]还给出了从某个可行阈值分配得到最大阈值分配的算法。

抢占阈值调度策略能够有效提高任务集合的可调度性, 为了比较任务集合在不同调度策略下的可调度性, 文献[17]提出“临界过载利用率” (break-down utilization) 的概念作为任务集合可调度性的定量衡量. 任务集合  $T$  的临界过载利用率  $U_{bd}$  是指当  $T$  内所有任务的执行时间按比例增加以至于使得某个任务无法在截止期限之前完成时, 任务集合  $T$  的处理器利用率. 在本文中, 计算  $U_{bd}$  过程中所用

的任务执行时间增加系数(用  $\eta$  表示)是任务集合  $T$  内最小归一化执行速度(用  $f_i/FLK_i$  表示)的下界, 即  $\min\{f_i/FLK_i, i=1..n\} \geq \eta$ .

### 3.3 功耗最优的抢占阈值分配

在本文 3.2 节给出的任务集合可调度性判定过程中, 任务  $\tau_i$  的自身最长执行时间  $C_i$  与处理器频率  $f_i$  相关,  $C_i = C_i/f_i$ ,  $f_i$  不同,  $C_i$  就不同, 当然系统  $(T, \Pi, \Gamma')$  的功耗就不同, 此时, 实时任务集合  $T$  是带参数  $F$  的静态优先级系统  $(T, \Pi, \Gamma')$ , 其中  $F = [f_1, f_2, \dots, f_n]$ ,  $f_i \in \{FLK^1, FLK^2, \dots, FLK^m\}$ , 而本节的任务就是发现使得任务集合  $T$  功耗最小的抢占阈值分配.

对于任务集合  $T$  的各个任务而言, 在确定任务执行时所用的处理器频率之后, 任务的各种时间特征也就最终确定, 而任务集合  $T$  就转换为不带参数的普通静态优先级任务集合  $\hat{T}$ , 此时, 使用第 2 节给出的可调度性判定方法, 我们可以使用现有的各种算法<sup>[17, 21, 23]</sup>来为各个任务分配抢占阈值, 从而减少不必要的任务切换次数. 显然, 不同的  $F, \hat{T}$  就不同, 那么集合内各个任务所分配的抢占阈值也就不同. 实际上, 由于参数  $F$  的变化而生成的各个任务集合  $\hat{T}$  的抢占阈值分配都各不相同, 这些可行的抢占阈值分配就构成了任务集合  $T$  功耗最小的抢占阈值分配的解空间. 而且各个任务集合  $\hat{T}$  的功耗由于  $F$  不同也不相同, 那么哪个任务集合  $\hat{T}$  的功耗最低呢?

本节用  $\gamma(T, \Pi)$  表示已采用  $\Pi$  分配任务优先级的任务集合  $T$  所有可行抢占阈值分配的集合.

**定义 1**(最小阈值分配, Identity Preemption Threshold Assignment, IPTA). 在 IPTA 内, 任务的抢占阈值等于任务优先级, 即对于  $i \in [1, 2, \dots, n]$ ,  $\pi_i = \gamma_i$ . IPTA 记为  $\Gamma^I$ ,  $\Gamma^I \in \gamma(T, \Pi)$ .

由于任务集合  $T$  的任务优先级是预先分配的, 所以  $\Gamma^I$  总是存在的, 与就是说,  $\gamma(T, \Pi)$  必定不是空集. 给定任意两个阈值分配  $\Gamma, \Gamma' \in \gamma(T, \Pi)$ , 如果  $\forall i, \gamma_i \geq \gamma'_i$ , 就认为  $\Gamma$  比  $\Gamma'$  “更大”, 记为  $\Gamma > \Gamma'$ , 显然  $>$  是偏序关系,  $>$  使得  $\gamma(T, \Pi)$  成为具有最小上界和最大下界的集合. 显然最小分配  $\Gamma^I$  是  $\gamma(T, \Pi)$  的最大下界. 下面定义  $\gamma(T, \Pi)$  的最小上界  $\Gamma^{\max}$ .

**定义 2**(最大阈值分配, Maximal Preemption Threshold Assignment, MPTA). MPTA 记为  $\Gamma^{\max}$ ,  $\Gamma^{\max} = (\gamma_1^{\max}, \gamma_2^{\max}, \dots, \gamma_n^{\max}) \in \gamma(T, \Pi)$ , 而且对于所有  $\Gamma \in \gamma(T, \Pi)$ ,  $\Gamma^{\max} > \Gamma$ .

文献[17]首先提出 MPTA, 而后文献[21, 23]分析了 MPTA, 指出只要任务集合存在一个可行阈

值分配, 那么 MPTA 总是存在而且可以计算出的. 因为我们总可以以 IPTA 为起点, 使用第 3.2 节给出的可调度性判定方法, 逐步提升每个任务的抢占阈值, 最终计算得出 MPTA.

下面我们给出本节的主要定理, 在该定理的证明过程中, 任务集合  $T$  在使用优先级分配  $\Pi$  和抢占阈值分配  $\Gamma$  的情况下功耗记为  $P_{\text{total}}(T, \Gamma, \Pi)$ .

**定理 1.** 给定两个实时系统  $(T, \Gamma, \Pi)$  和  $(T, \Gamma', \Pi)$ , 其中  $\Gamma, \Gamma' \in \gamma(T, \Pi)$ , 而且  $\Gamma > \Gamma'$ , 功耗  $P_{\text{total}}(T, \Gamma', \Pi)$  不可能比  $P_{\text{total}}(T, \Gamma, \Pi)$  更小.

证明. 为不失一般性, 假定  $\gamma_k = \gamma'_k$  对于  $k=1, 2, \dots, i-1, i+1, \dots, n$ , 对于  $i \in [1, 2, \dots, n]$  令  $\gamma'_i < \gamma_i$ , 显然  $\Gamma > \Gamma'$ . 当任务  $\tau_i$  的抢占阈值从  $\gamma_i$  变化为  $\gamma'_i$  时, 根据文献[17]的定理 4.1, 对于  $\pi_k > \gamma'_i$  的任务  $\tau_k$  而言,  $\tau_k$  的最长响应时间不变化; 对于  $\gamma'_i > \pi_k > \pi_i$  的任务  $\tau_k$  而言,  $\tau_k$  的最长响应时间也保持不变; 对于  $\gamma'_i < \pi_k < \gamma_i$  的任务  $\tau_k$  而言,  $\tau_k$  在阈值分配  $\Gamma'$  下的最长响应时间并不比  $\Gamma$  更差, 而且  $\tau_k$  会对任务  $\tau_i$  的正常执行产生更多的干扰, 从而进一步压缩任务  $\tau_i$  自身的计算时间. 在第 3.2 节任务  $\tau_i$  的可调度性判定过程中,  $\tau_i$  自身计算时间与处理器频率  $f_i$  相关,  $f_i$  决定了  $\tau_i$  执行时的速度, 提升  $f_i$  可以减少  $\tau_i$  的执行需求并减少  $\tau_i$  的功耗. 如果  $\tau_i$  的抢占阈值的变化 ( $\gamma_i \rightarrow \gamma'_i$ ) 使得  $\tau_i$  本身不可调度, 那么提升  $f_i$  至  $f'_i$ ,  $f'_i > f_i$ , 将是使得  $\tau_i$  重新可调度的唯一途径. 显然, 当任务集合  $T$  的阈值分配“减少”(在偏序关系  $>$  的意义上)时, 即  $\Gamma \rightarrow \Gamma'$ , 使得  $T$  的功耗增加, 即  $P_{\text{total}}(T, \Gamma', \Pi) > P_{\text{total}}(T, \Gamma, \Pi)$ .

如果任务  $\tau_i$  执行时的处理器频率从  $f_i$  被调低为  $f''_i$ , 其中  $f''_i < f_i$ ,  $\tau_i$  自身计算时间就被延长了, 那么  $\tau_i$  的最长响应时间也随之增加, 增量为  $C_i(1/f''_i - 1/f_i)$ , 如果  $R_i + C_i(1/f''_i - 1/f_i) \leq D_i$ , 即调低  $f_i$  依然可以保持  $\tau_i$  的可调度性, 那么调低  $f_i$  至  $f''_i$  就是可行的并降低系统功耗. 对于  $\pi_k < \pi_i$  的任务  $\tau_k$  而言,  $\tau_i$  计算时间的增加会对  $\tau_k$  的正常执行造成更多的干扰, 会增大  $\tau_k$  的最长响应时间, 影响  $\tau_k$  的可调度性. 假设任务  $\tau_j$  是任务子集  $\{\tau_k | \pi_k < \pi_i\}$  中最长响应时间最接近  $D_j$  的, 如果  $\tau_i$  的增量  $C_i(1/f''_i - 1/f_i)$  使得  $\tau_j$  不可调度, 那么, 提升  $\tau_j$  的抢占阈值  $\gamma_j$  至  $\gamma''_j$ ,  $\gamma''_j > \gamma_j$ , 则是使得  $\tau_j$  恢复可调度性的唯一途径. 对于  $\pi_i < \pi_k < \gamma_i$  的任务  $\tau_k$  而言,  $\tau_i$  的增量  $C_i(1/f''_i - 1/f_i)$  可能会增加  $\tau_k$  的阻塞时间. 不过, 根据方程(3)可知,  $\tau_k$  的阻塞时间  $B_k$  是几乎很难达到的上限, 而且根据文献[23]的定理 2,  $\tau_k$  在执行过程中最多被  $\tau_i$  阻塞一

次,因此, $\tau_i$ 自身计算时间的增加对任务 $\tau_k$ 的最长响应时间几乎没有影响,即使有影响,使得 $\tau_k$ 不可调度,也可以通过提升 $\tau_k$ 的抢占阈值重新恢复其可调度性.只是,如果 $\tau_k$ 的抢占阈值已经等于系统最高任务优先级,那么根据文献[17]的定理4.2,任务集合 $T$ 都是不可调度的,此时,调低 $\tau_i$ 的处理器频率就是不可行的,就只有保持 $f_i$ 不变.

总之,当任务集合 $T$ 的阈值分配“增大”时,即 $\Gamma \rightarrow \Gamma'', \Gamma'' > \Gamma, P_{\text{total}}(T, \Gamma'', \Pi) < P_{\text{total}}(T, \Gamma, \Pi)$ .

证毕.

**推论 1.** 任务集合 $T$ 在 MPTA 下可以获得最小功耗.

证明. 根据定理 1,可以以 IPTA 为起点,逐步提高各个任务的抢占阈值,从而可以调低各个任务执行时的处理器频率,降低系统功耗,最终发现 MPTA 时,系统功耗也达到最低. 证毕.

## 4 两阶段节能调度算法

本节给出用于任务集合 $T$ 节能调度的两阶段算法.第一阶段先离线计算使得任务集合 $T$ 功耗最小时各个任务所需的处理器频率 $[f_1^{\text{opt}}, f_2^{\text{opt}}, \dots, f_n^{\text{opt}}]$ ,第二阶段在线决定何时关闭处理器来进一步降低功耗.

### 4.1 最小化执行功耗的离线算法 EE-FPPT

根据推论 1 可知,对于使用 DVS 技术的静态优先级系统 $(T, \Pi, \Gamma^I)$ 而言,可在阈值分配为 MPTA 时取得最小功耗并大幅减少任务切换次数,那么从 $(T, \Pi, \Gamma^I)$ 出发,如何得出使得系统取得最小功耗时各个任务所需的处理器频率,并且在此过程中保持系统的可调度性呢?

显然,求取静态优先级系统 $(T, \Pi, \Gamma^I)$ 的最小功耗的过程,就是求取静态优先级系统 $(T, \Pi, \Gamma^I)$ 的最大抢占阈值 MPTA 的过程,因为任务集合 $T$ 在 MPTA 下功耗最小(推论 1).那么,如何计算 $(T, \Pi, \Gamma^I)$ 的最大抢占阈值呢? 文献[17, 21, 23]也给出了用于计算普通静态任务优先级系统的各个抢占阈值分配的算法,不过这些算法的使用前提是实时任务各种时间特征(比如任务周期、任务最长执行时间等)已经确定不变.而本文中的实时任务集合 $T$ 实际上是带参数 $F$ 的静态优先级系统 $(T, \Pi, \Gamma^I)$ ,其中 $F = [f_1, f_2, \dots, f_n]$ ,而本节的任务就是合理调整参数 $F$ ,求取系统 $(T, \Pi, \Gamma^I)$ 的最大抢占阈值 MPTA,同时也使得系统 $(T, \Pi, \Gamma^I)$ 的功耗达到最

低.接下来给出的算法以 IPTA 为起点,在保证任务可调度性的前提下逐渐提高任务的抢占阈值,降低任务执行时的处理器频率,在发现 MPTA 的同时,也求出了系统的最小功耗.

#### 算法 1. FindMaximalPTA.

输入:  $T, \Pi, \Gamma^I$

输出:  $f_1^{\text{opt}}, f_2^{\text{opt}}, \dots, f_n^{\text{opt}}$

```

1.  $\Gamma \leftarrow \Gamma^I$ 
2. 将任务集合  $T$  按照优先级升序排列
   //从最高优先级任务开始
3. for ( $i \leftarrow 1$  to  $n$ ) do
4.   While ( $(\text{schedulable} = \text{True})$  and  $(f_i \neq \text{FLK}_i)$ ) do
5.     If  $(f_i = \text{FLK}_k)$  then
6.        $f_i \leftarrow \text{FLK}_{k-1}$ 
7.     Endif
8.   for ( $j = 1; j \leq \pi_i - 1; j++$ ) do
9.      $\text{schedulable} \leftarrow \text{False}$ 
10.    While ( $(\text{schedulable} = \text{False})$  and  $(\gamma_j \leq n)$ ) do
11.       $\text{schedulable} \leftarrow \text{schedulabilityTest}(\tau_j)$ 
12.      If  $(\text{schedulable} = \text{False})$  then
13.         $\gamma_j \leftarrow \gamma_j + 1$ 
14.      Endif
15.      If  $(\gamma_j \leq n)$  then
16.         $\text{schedulable} \leftarrow \text{True}$ 
17.      Else
18.         $f_i \leftarrow \text{FLK}_k$ 
19.      Endif
20.    Endwhile
21.  Endfor
22.  for ( $k = \pi_i + 1; k \leq \gamma_i; k++$ ) do
23.     $\text{schedulable} \leftarrow \text{schedulabilityTest}(\tau_k)$ 
24.    If  $(\text{schedulable} = \text{False})$  then
25.       $f_i \leftarrow \text{FLK}_k$ 
26.    Endif
27.  Endfor
28. Endwhile
29. Endfor

```

算法能从任务集合的最低优先级任务开始,尝试调低任务的执行速度(5~7 行),而后检查受其影响的任务的可调度性,对于更低优先级的任务,如果由于当前任务执行速度的调低而使其不可调度,则提升这些低优先级任务的抢占阈值,以保持可调度性(8~21 行);如果这些低优先级任务的抢占阈值无法再提高,就说明当前任务的执行速度无法调低,只好恢复其原有速度(17~19 行);对于优先级高于当前任务,但是可能被当前任务所阻塞的其他任务而言,如果当前任务的速度调低使其不可调度,就只

好以其原有速度(22~27 行),因为提高这些任务的抢占阈值没有效果.

算法运行时的计算复杂度主要取决于各个任务可调度性分析过程的计算复杂度,其中式(3)~(7)需要迭代计算,求解这些方程的计算复杂度是不确定的,因此任务可调度性分析过程的计算复杂度同样也是不确定的.所以,该离线算法的计算复杂度是伪多项式时间.

#### 4.2 最小化空闲功耗的在线算法 LA-FPPT

假定使用 4.1 节的 EE-FPPT 算法所获得的调度结果用符号  $S_e$  表示,同时令  $C_i = C_i / f_{opt}^i$ . 第一阶段的 EE-FPPT 算法给出了在保证任务集合可调度性的前提下使得系统执行功耗  $\phi(S_e)$  最小时各个任务需要的最优处理器频率  $[f_1^{opt}, f_2^{opt}, \dots, f_n^{opt}]$ . 第二阶段则通过在线判断何时关闭处理器来降低系统空闲功耗  $\epsilon(S_e)$ , 其主要思想是通过整合超周期  $L$  内多个零散空闲时段,让处理器尽可能长时间地处于空闲状态(空闲时段长度至少大于  $t_0$ ),从而使得处理器尽可能多地进入睡眠模式,降低系统空闲功耗. 延迟调度算法通过推迟任务的释放时间来计算任务在保证可调度性的前提下的最大可延迟时间(Procrastination Interval)来使得处理器尽可能长地处于睡眠模式,文献[11,16]给出了 EDF 调度策略下任务可延迟时间的计算方法,而文献[14-15]给出了静态优先级调度策略下任务可延迟时间的计算方法. 其中文献[11,14]基于系统利用率来计算任务可延迟时间,而文献[15-16]通过将就绪任务实例尽可能推迟执行来计算任务可延迟时间.

本节我们给出一种基于时间需求分析的在线模拟调度算法来计算超周期  $L$  内空闲时段长度. 如果在时刻  $t$  某任务实例执行完毕,而此时系统任务队列为空,我们必须决定是否有必要关闭处理器. 令  $r_i(t)$  表示任务  $\tau_i$  在时刻  $t$  之后第一个任务实例(记为  $J_{i,t}$ )的释放时间,即  $r_i(t) = \lceil t/T_i \rceil \cdot T_i$ , 令  $d_i(t)$  表示  $J_{i,t}$  的截止时刻,即  $d_i(t) = r_i(t) + D_i$ .

我们的方法首先考虑在时间间隔  $[t, t + d_i(t))$  内如何计算优先级为  $\pi_i$  的最长可用空闲时段长度,记为  $S_{\pi_i}^{\max}(t)$ . 为了确保任务  $\tau_i$  实例可在  $d_i(t)$  之前完成,我们必须分析自  $t$  时刻起的最坏调度情况. 可以看出,时间间隔  $[t, t + d_i(t))$  是由一系列 level- $i$  忙周期和 level- $i$  空闲周期交错组成,自  $\tau_i$  实例完成时刻之后到  $d_i(t)$  之前间的任何 level- $i$  空闲周期都通过与临近的 level- $i$  忙周期相互交换而用于计算  $S_{\pi_i}^{\max}(t)$ .

我们使用文献[25]的技巧推导出方程(8),尽管在  $t$  时刻系统就绪队列为空,但由于延迟调度的影响,level- $i$  忙周期长度受到以下两个因素影响:(1)对于  $\pi_k < \pi_i < \gamma_k$  的任务  $\tau_k$  而言,  $\tau_k$  的某个任务实例恰好在 level- $i$  忙周期之前释放;(2)对于  $\pi_j > \gamma_i$  的任务  $\tau_j$  而言,  $\tau_j$  的任务实例在 level- $i$  忙周期内释放. 其中第 2 个因素隐含着某种递归定义,level- $i$  忙周期内需要处理的负载随着 level- $i$  忙周期长度的增加而单调增长,因此,可使用方程(8)来计算 level- $i$  忙周期内需要处理的负载总量  $w_{\pi_i}(t)$ . 其中  $S_{\pi_i}(t)$  表示自  $t$  时刻起 level- $i$  空闲周期的起点.

$$w_{\pi_i}^{m+1}(t) = S_{\pi_i}(t) + \max_{\forall k, \pi_k < \pi_i \leq \gamma_k} C_k + \sum_{\forall j, \pi_j > \pi_i} \left( \left\lceil \frac{w_{\pi_i}^m(t) - r_j(t)}{T_j} \right\rceil \cdot C_j \right) \quad (8)$$

递归方程(8)的计算可自  $w_{\pi_i}^0(t) = 0$  开始,不断迭代,直到  $w_{\pi_i}^{m+1}(t) = w_{\pi_i}^m(t)$  或者  $w_{\pi_i}^{m+1}(t) \geq d_i(t)$  为止. 递归方程(8)可收敛性的证明与文献[25]中的方法类似. 递归方程(8)的计算结果  $w_{\pi_i}(t)$  定义了该 level- $i$  忙周期的长度,那么也可以将  $t + w_{\pi_i}(t)$  视为 level- $i$  空闲周期的起点. 如果  $[t, t + d_i(t))$  内某个 level- $i$  空闲周期的起点在时间间隔  $[t, t + d_i(t))$  内,那么该 level- $i$  空闲周期的终点要么是时间间隔  $[t, t + d_i(t))$  的终点,要么是某个更高优先级任务  $\tau_j (\pi_j > \pi_i)$  实例的释放时刻. 所以,方程(9)给出了 level- $i$  空闲周期的长度  $l_i(t, w_{\pi_i}(t))$ :

$$l_i(t, w_{\pi_i}(t)) = \min \left[ d_i(t) - w_{\pi_i}(t), \min_{\forall j, \pi_j \geq \gamma_i} \left( \left\lceil \frac{w_{\pi_i}(t) - r_j(t)}{T_j} \right\rceil \cdot T_j + r_j(t) - w_{\pi_i}(t) \right) \right] \quad (9)$$

其中  $d_i(t) - w_{\pi_i}(t)$  表示 level- $i$  空闲周期在时间间隔  $[t, t + d_i(t))$  的终点时刻结束,而  $\lceil (w_{\pi_i}(t) - r_j(t)) / T_j \rceil \cdot T_j + r_j(t)$  表示任务  $\tau_j$  在 level- $i$  忙周期内的执行需求. 根据方程(8)和方程(9),可以给出  $S_{\pi_i}^{\max}(t)$  的计算步骤如算法 2 的 7~14 行所示. 首先设置  $S_{\pi_i}(t)$  为 0,而后使用方程(8)计算时间间隔  $[t, t + d_i(t))$  level- $i$  忙周期的结束时刻  $w_{\pi_i}^{m+1}(t)$ , level- $i$  忙周期的结束时刻可以认为是 level- $i$  空闲周期的起点时刻,方程(9)给出了 level- $i$  空闲周期的长度  $l_i(t, w_{\pi_i}^m(t))$ ; 此时  $S_{\pi_i}(t)$  递加  $l_i(t, w_{\pi_i}^m(t))$ ,  $w_{\pi_i}^{m+1}(t)$  也递加  $l_i(t, w_{\pi_i}^m(t))$ , 如果此时  $w_{\pi_i}^{m+1}(t)$  到达了  $J_{i,t}$  的截止时刻,就停止计算,否则就重复上述步骤.

**算法 2.** Dynamic Procrastination.

某任务实例在时刻  $t$  完成,此时系统任务队列为空.

1.  $\Gamma \leftarrow \Gamma^{\max}$  //任务集合  $T$  的阈值分配为 MPTA
2. 将任务集合  $T$  按照任务优先级升序排列  
//从最低优先级任务开始
3. for ( $i \leftarrow 1$  to  $n$ ) do
4.  $r_i(t) \leftarrow \lceil t/T_i \rceil \cdot T_i$
5.  $d_i(t) \leftarrow r_i(t) + D_i$
6.  $S_{\pi_i}(t) \leftarrow 0$
7.  $w_{\pi_i}^{m+1}(t) \leftarrow 0$
8. While ( $w_{\pi_i}^{m+1}(t) \leq d_i(t)$ ) do
9.  $w_{\pi_i}^m(t) \leftarrow w_{\pi_i}^{m+1}(t)$
10.  $w_{\pi_i}^{m+1}(t) = S_{\pi_i}(t) + \max_{\forall k, \pi_k < \pi_i \leq \gamma_k} C_k +$   

$$\sum_{\forall j, \pi_j > \pi_i} \left( \left\lceil \frac{w_{\pi_i}^m(t) - r_j(t)}{T_j} \right\rceil \cdot C_j \right)$$
11. If ( $w_{\pi_i}^{m+1}(t) = w_{\pi_i}^m(t)$ ) then
12.  $S_{\pi_i}(t) \leftarrow S_{\pi_i}(t) + l_i(t, w_{\pi_i}^m(t))$
13.  $w_{\pi_i}^{m+1}(t) \leftarrow w_{\pi_i}^{m+1}(t) + l_i(t, w_{\pi_i}^m(t))$
14. Endif
15. Endwhile
16. Endfor
17.  $S_{\pi_i}^{\max}(t) \leftarrow S_{\pi_i}(t)$
18. 修改  $J_{i,t}$  的释放时间为  $r'_i(t)$ ,  
 $r'_i(t) \leftarrow r_i(t) + S_{\pi_i}^{\max}(t)$
19. If ( $\min_{\forall \tau_i \in T} r'_i(t) - t > t_\theta$ ) then
20. 在  $t$  时刻关闭处理器而  
 在  $\min_{\forall \tau_i \in T} r'_i(t)$  时刻重新开启处理器
21. Else
22. 保持处理器处于活跃模式
23. Endif

在计算出每个任务  $\tau_i$  的最长可用空闲时段长度  $S_{\pi_i}^{\max}(t)$  之后,  $J_{i,t}$  的释放时刻就可以推迟  $S_{\pi_i}^{\max}(t)$ , 这种任务释放时刻的推迟不会影响  $J_{i,t}$  的实时性. 在计算出任务集合  $T$  内所有任务实例  $J_{i,t} (i=1, 2, \dots, n)$  的新释放时刻之后, 最早释放时刻 ( $\min_{\forall \tau_i \in T} r'_i(t)$ ) 与  $t$  时刻的距离如果没有大于“空闲时段阈值”  $t_\theta$ , 就让处理器处于活跃模式, 否则就在  $t$  时刻关闭处理器.

计算  $S_{\pi_i}^{\max}(t)$  的时间复杂度为  $O(mn)$ , 其中  $m$  表示迭代次数, 而  $n$  表示任务集合  $T$  的任务个数, 因此, 算法 2 的时间复杂度为  $O(mn^2)$ . 由于迭代次数  $m$  依赖于任务集合  $T$  内各个任务的周期和执行时间, 所以算法 2 的时间复杂度是伪多项式时间的.

**4.3 两阶段调度算法的分析**

为了叙述的简洁, 将上述两阶段调度算法在超周期  $(0, L]$  内的实际调度结果用符号  $S_\phi$  表示.  $S_\phi$  的

可行性可由引理 1 给出.

**引理 1**(两阶段算法的可调度性). 在任意实际调度结果  $S_\phi$  中, 所有任务实例都在各自截止时间之前完成.

证明. 算法第一阶段所得到的调度结果  $S_e$  是可行的, 其中所有任务实例均在各自截止时间之前执行完毕. 算法第二阶段将  $S_e$  内  $t$  时刻之后释放的每个任务实例的零散空闲时间收集起来, 在保证各个任务实例在  $d_i(t) (i=1, 2, \dots, n)$  之前完成的前提下推迟各个任务实例的释放时间. 与此同时,  $S_e$  内每个任务实例的执行需求(时钟周期个数)与  $S_\phi$  相同, 所以  $S_\phi$  内所有任务实例的实时性也都得到满足.

证毕.

接下来我们分析本文提出的两阶段调度算法所产生的功耗与最优调度所产生的功耗的近似比率. 后续的理论分析表明, 本文给出的算法所得到的调度结果尽管不是功耗最小的, 但只是最小功耗的  $\max\{1/(U_{bd})^2, 2\}$  倍, 其中  $U_{bd}$  是  $S_e$  的临界过载利用率.

**引理 2.** 对于在最优调度结果  $S^*$  中的任意空闲时段  $I$ ,  $I$  最多与实际调度结果  $S_\phi$  中两个不连续的空闲时段重叠.

证明. 运用反证法. 假设空闲时段  $I$  至少与实际调度结果  $S_\phi$  中 3 个不连续的空闲时段重叠, 换句话说,  $I$  至少与  $S_\phi$  中两个不连续的执行时段重叠, 假定  $e_1^\phi$  和  $e_2^\phi$  分别是  $S_\phi$  中最早与  $I$  重叠的执行时段, 其中在时间线上,  $e_1^\phi$  在  $e_2^\phi$  之前. 显然, 自  $e_1^\phi$  的结束时刻系统开始使用第 3.2 节中的在线模拟延迟调度算法, 来判断是否需要关闭处理器. 显然, 在  $S_\phi$  中, 时间间隔 ( $\min_{\forall \tau_i \in T} r'_i(t), \min_{\forall \tau_i \in T} d_i(t)$ ) 内不存在 level- $i$  空闲周期, 同时在  $S_\phi$  中, 在时间间隔 ( $\min_{\forall \tau_i \in T} r'_i(t), \min_{\forall \tau_i \in T} d_i(t)$ ) 至少存在一个任务实例(记为  $J_j^*$ ),  $J_j^*$  在  $t$  时刻之后释放而截止时刻为  $\min_{\forall \tau_i \in T} d_i(t)$ . 必然存在某个任务实例(记为  $J_j$ ), 它的释放时间在  $e_1^\phi$  的结束时刻之后, 而  $J_j$  的截止时刻等于  $e_2^\phi$  的结束时刻. 根据  $S^*$  的定义,  $S^*$  是任务集合的最优可行调度, 那么在此情况下, 空闲时段  $I$  内至少还需要执行任务实例  $J_j$ . 这显然与空闲时段的定义矛盾. 证毕.

接下来的引理给出当调度结果  $S_e$  的处理器利用率过高时、两阶段调度算法所产生的功耗与最优调度所产生的功耗的近似比率.

**引理 3.**  $E(S_\phi) \leq \frac{1}{(U_{bd})^2} \cdot E(S^*)$ , 其中



$$\sum_{i=1}^n \frac{C_i}{T_i \hat{f}} \geq U_{bd}.$$

证明. 文献[24]指出对于使用 EDF 调度策略的动态优先级任务集合而言, 如果所有任务实例都使用  $\sum_{i=1}^n C_i/T_i$  速度执行, 那么系统功耗最低. 根据关键速度的定义, 可知如果所有任务实例均以  $\max\left\{\sum_{i=1}^n C_i/T_i, \hat{f}\right\}$  的速度执行, 系统功耗最小. 令  $f^b = \max\left\{\sum_{i=1}^n C_i/T_i, \hat{f}\right\}$ , 可得到如下结果:

$$E(S^*) \geq P(f^b) \cdot (L/f^b) \cdot \left(\sum_{i=1}^n C_i/T_i\right) \quad (10)$$

令  $f^u = \min\left\{FLK_s, \frac{1}{(U_{bd})} \sum_{i=1}^n C_i/T_i\right\}$ , 根据  $S_e$  的

定义, 可知  $\phi(S_\psi) \leq P(f^u) \cdot (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right)$ . 根据功耗函数的定义,  $P(f) = P_s(f) + P_d(f)$ , 其中  $P_d(f)/f$  是处理器频率  $f$  的非递减凸函数(见方程(1)), 而  $P_s(f)/f$  是处理器频率  $f$  的非递增函数. 因此, 在超周期  $L$  内由于泄露电流引发的空闲能耗  $\epsilon(S_\psi)$  最多是  $P_s(FLK_1)(L - (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right))$ , 那么调度  $S_\psi$  的整体能耗为

$$E(S_\psi) \leq (P_d(f^u) + P_s(f^u)) \cdot (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right) + P_s(FLK_1)(L - (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right)) \quad (11)$$

其中  $P_s(FLK_1)(L - (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right))$  是系统任务队列为空、处理器处于空闲状态时的能耗. 此时方程(10)可重新表示为

$$E(S^*) \geq (P_d(f^b) + P_s(f^b)) \cdot (L/f^b) \cdot \left(\sum_{i=1}^n C_i/T_i\right) \quad (12)$$

根据方程(11)和方程(12), 可得到如下结果:

$$\begin{aligned} \frac{E(S^*)}{E(S_\psi)} &\leq \max\left\{\frac{P_d(f^u)f^b}{P_d(f^b)f^u}, \right. \\ &\quad \left.(P_s(f^u) \cdot (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right) + \right. \\ &\quad \left. P_s(FLK_1)(L - (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right)) / \right. \\ &\quad \left. (P_s(f^b) \cdot (L/f^b) \cdot \left(\sum_{i=1}^n C_i/T_i\right))\right\} \quad (13) \end{aligned}$$

由于  $\frac{f^u}{f^b} \leq U_{bd}$ , 那么  $\frac{P_d(f^u)f^b}{P_d(f^b)f^u} \leq \frac{1}{(U_{bd})^2}$ . 因为

$P_s(f)/f$  是处理器频率  $f$  的非递增函数, 所以  $P_s(f^u)/f^u \leq P_s(f^b)/f^b$ . 其中  $f^b \geq FLK_1$ , 那么  $P_s(f^b) \geq P_s(FLK_1)$ , 而且  $\sum_{i=1}^n \frac{C_i}{T_i f^u} \geq U_{bd}$ , 那么  $\sum_{i=1}^n \frac{C_i}{T_i f^b} \geq U_{bd}$ , 由此可知

$$\begin{aligned} P_s(FLK_1)(L - (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right)) &\leq \\ &\left(\frac{1 - U_{bd}}{U_{bd}}\right) \cdot P_s(f^b) \cdot (L/f^b) \cdot \left(\sum_{i=1}^n C_i/T_i\right) \end{aligned} \quad (14)$$

因此, 可得到

$$\begin{aligned} &(P_s(f^u) \cdot (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right) + \\ &P_s(FLK_1)(L - (L/f^u) \cdot \left(\sum_{i=1}^n C_i/T_i\right))) / \\ &P_s(f^b) \cdot (L/f^b) \cdot \left(\sum_{i=1}^n C_i/T_i\right) \leq \frac{1}{(U_{bd})} \end{aligned} \quad (15)$$

那么根据方程(13), 可得到  $\frac{E(S^*)}{E(S_\psi)} \leq \frac{1}{(U_{bd})^2}$ . 证毕.

接下来的引理给出当调度结果  $S_e$  的处理器利用率较低时、两阶段调度算法所产生的功耗与最优调度所产生的功耗的近似比率.

**引理 4.**  $E(S_\psi) \leq 2E(S^*)$ , 其中  $\sum_{i=1}^n \frac{C_i}{T_i \hat{f}_i} \leq U_{bd}$ .

证明. 如果处理器在某个调度结果中的空闲时段进入睡眠模式, 该空闲时段就称为“睡眠时段”. 调度结果  $S_\psi$  中所有空闲时段组成的集合用符号  $I_{S_\psi}$  表示, 而最优调度结果  $S^*$  中所有空闲时段组成的集合用符号  $I_{S^*}$  表示. 其中调度结果  $S_\psi$  中、与  $I_{S^*}$  中的某些睡眠时段重叠的空闲时段组成的集合用符号  $I_{S_\psi}^l$  表示. 用  $\epsilon_1(S)$  表示调度结果  $S$  中处理器处于空闲状态和活跃模式时的能耗,  $\epsilon_2(S)$  表示调度结果  $S$  中处理器从睡眠模式切换到活跃模式的能耗.

由于调度结果  $S_\psi$  中空闲时段的能耗最多为  $E_{sw}$ , 那么  $I_{S_\psi}^l$  的空闲能耗为  $|I_{S_\psi}^l| \cdot E_{sw}$ . 根据引理 2, 可知  $\epsilon_2(S^*) \geq \frac{1}{2} |I_{S_\psi}^l| \cdot E_{sw}$ . 因为集合  $I_{S_\psi} \setminus I_{S_\psi}^l$  内所有的空闲时段都与集合  $I_{S^*}$  内非睡眠时段相重叠, 所以可知  $\epsilon_1(S^*) + \phi(S^*) \geq \epsilon(S_\psi) - |I_{S_\psi}^l| \cdot E_{sw}$ . 结合上述不等式, 可得到如下结果:

$$\begin{aligned} \epsilon(S^\psi) &\leq 2\epsilon_2(S^*) + \epsilon_1(S^*) + \phi(S^*) \\ &\leq 2\epsilon(S^*) + \phi(S^*) \end{aligned} \quad (16)$$

当  $\sum_{i=1}^n \frac{C_i}{T_i \hat{f}_i} \leq U_{bd}$  时, 可知调度结果  $S_e$  中所有任

务实例都可以处理器频率  $f$  运行. 根据关键速度  $f$  的定义, 调度结果  $S_e$  的能耗在所有可行调度中最小. 因此,  $\phi(S_\psi) = \phi(S_e) \leq \phi(S^*)$ , 最后可得到  $E(S_\psi) \leq 2E(S^*)$ . 证毕.

根据引理 3 和引理 4, 可以给出本文提出的两阶段算法在解决 LAEES-FPPT 问题时所得调度结果的能耗与最优调度能耗的近似比率.

**定理 2.** 对于任何存在可行 FPPT 调度结果的任务集合而言,  $E(S_\psi) \leq \max\{2, (1/(U_{bd})^2)\}E(S^*)$ .

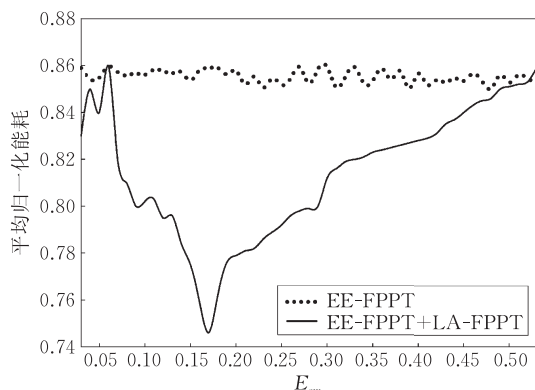
证明. 可根据引理 3 和引理 4 直接证明. 证毕.

## 5 仿真实验

第 4 节的工作指出两阶段算法能够减少处理器 CMOS 电路中的能耗泄露并降低任务集合的执行能耗, 同时保持任务集合的可调度性. 本节将使用随机生成的大量任务集合来考察算法各种特性的变化趋势.

本文的实验使用的处理器功耗函数为  $P(f) = f^3 + 3$ . 目前, 在支持睡眠模式的可变电电压处理器上使用 FPPT 调度模型的任务调度算法还没有类似的研究工作, 所以, 为了检验它的节能效果, 仿真实验采用“归一化能耗”(normalized total energy)作为节能效果评价标准. 所谓归一化能耗是指在时间间隔  $(0, L]$  内两阶段算法给出的调度结果的能耗除以采用普通 FPPT 调度策略时调度结果的能耗所得到的数值.

本节我们使用两组不同的实验设计, 其中第一组生成的随机任务集合用于考察处理器模式切换的



额外能耗  $E_{sw}$ 、系统利用率的变化对算法节能效果的影响. 为了保证仿真测试的覆盖程度, 我们随机生成了 20000 个任务集合, 每个任务集合都包括 10 个任务. 这 20000 个任务集合是这样创建的: 其中 1000 个任务集合具有 50% 的处理器利用率, 而后创建的 1000 个任务集合具有 52% 的处理器利用率, ..., 直到最后创建的 1000 个任务集合具有 90% 的处理器利用率. 对于每 1 组具有同样处理器利用率的任务集合而言, 这 1000 个任务集合又是这样创建的: 20 个任务集合内  $E_{sw}$  为 0.03, 而后 20 个任务集合内  $E_{sw}$  为 0.04, ..., 最后 20 个任务集合内的  $E_{sw}$  为 0.53. 第 2 组生成的随机任务集合用于考察任务集合内任务个数对算法节能效果的影响, 此时系统利用率为 67%,  $E_{sw}$  为 0.17. 同样为了保证仿真覆盖程度, 再随机生成 20000 个任务集合, 其中 1000 个任务集合具有 5 个独立任务, 而后创建的 1000 个任务集合具有 6 个独立任务, ..., 直到最后创建的 1000 个任务集合具有 25 个独立任务. 任务周期是在  $[1, 100]$  遵循均匀分布的随机变量, 出于实现的简单考虑, 任务截止时间等于任务周期, 任务的 WCEC 被分别设置以满足任务集合的处理器利用率. 所有这 40000 个任务集合在完全抢占式调度策略下都是可调度的.

图 1 的左图显示, 处理器模式切换能耗  $E_{sw}$  越高, 算法的整体功耗也随之增加. 只是当  $E_{sw}$  相对较小的时候,  $E_{sw} \leq 0.18$ , 由于 CMOS 泄露电路引起的系统静态能耗对于整体能耗的影响还非常微弱, 所以,  $E_{sw}$  越高, 算法的归一化功耗反而降低.

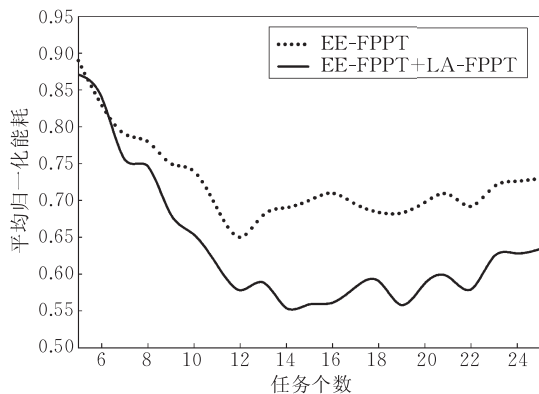


图 1 考虑处理器模式切换能耗变化以及任务个数变化时, 两阶段算法实验结果

图 2 中的左图显示, 在系统利用率相对较高的时候,  $U \geq 0.87$ , 两阶段算法 (EE-FPPT + LA-FPPT) 的节能效果反而不如 EE-FPPT 算法本身, 即不使用在线模拟调度来推迟任务执行、暂时关闭处理器对节能而言还更好些. 这是由于在系统利用率相对

较高时, LA-FPPT 算法可能会创建两个或者多个空闲时段来暂时关闭处理器, 同时原有的 EE-FPPT 算法不推迟任务执行, 反而可能会在比较短的空闲时段保持处理器处于活跃模式而在比较长的空闲时段关闭处理器, 这就造成两阶段算法此时节能效果

的下滑. 而且, 当系统利用率相对较高时, 空闲能耗在系统整体能耗中被边缘化, 执行能耗占据了系统

整体能耗中的绝大部分, 所以, 此时 LA-FPPT 算法的节能效果并不明显.

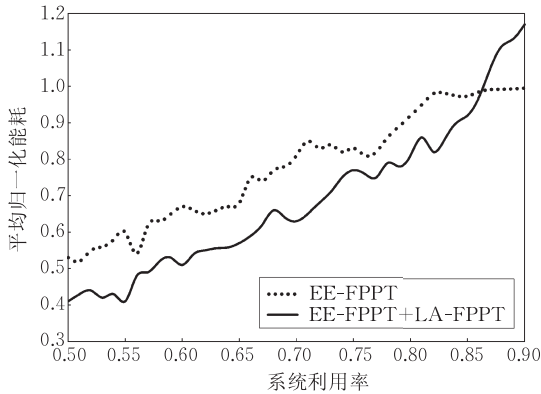


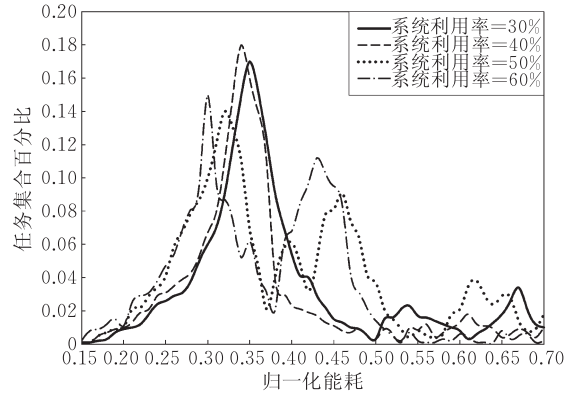
图 2 考虑系统利用率变化时两阶段算法的实验结果

图 1 中的右图显示, 两阶段算法的节能效果在任务个数比较少的时候 ( $\leq 12$ ), 随着任务个数的增加而下降; 而在任务个数比较多时候 ( $> 12$ ), 两阶段算法的节能效果保持平稳. 这是由于实验设计中第 2 组任务集合都使用同样的系统利用率, 那么在任务个数较少的时候, 每个任务本身的利用率 ( $C_i/(T_i f_i)$ ) 都比较高, 这就导致 LA-FPPT 算法很少有发生作用的空间; 而在任务个数较多的时候, 任务周期较小的任务个数增加, 而小周期任务的最高可推迟时间比较短, LA-FPPT 算法的作用也就比较微弱, 所以, LA-FPPT 算法对节能效果的提升就不甚明显.

另外一个有趣的现象是, 第 1 组 20000 个任务集合在不同的系统归一化功耗上的分布, 如图 2 中的右图所示. 该图给出了系统利用率在 30%、40%、50%、60% 时的分布情况, 显然, EE-FPPT 算法在静态优先级任务集合上的节能效果在某种程度上依赖于系统利用率的变化.

## 6 结束语

本文讨论了在支持睡眠模式的可变电电压处理器上对周期性实时任务集合进行节能调度的问题. 为了达到更好的节能效果, 我们提出的算法分为两个阶段, 第 1 阶段先离线计算使得任务集合功耗最小时各个任务所需的处理器频率, 第 2 阶段在线决定何时关闭处理器来进一步降低功耗. 理论分析表明, 该方法所带来的系统整体能耗最多是最优调度所带来的能耗的  $\max\{1/(U_{bd})^2, 2\}$  倍, 其中  $U_{bd}$  是使用 FPPT 调度策略的任务集合的临界过载利用率. 通



过实例研究和大量仿真实验表明, 我们提出的算法在保证任务集合可调度性的同时, 能够大幅降低系统功耗.

## 参 考 文 献

- [1] Sakurai Takayasu, Newton A Richard. Alpha-power law mosfet model and its applications to cmos inverter delay and other formulas. IEEE Journal of Solid-State Circuits, 1990, 25(2): 584-594
- [2] Kim Woonseok, Kim Jihong, Min Sang Lyul. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis//Paris F ed. Proceedings of the Design, Automation and Test in Europe Conference and Exposition. San Francisco, CA, USA, 2002: 788-794
- [3] Kim Woonseok, Kim Jihong, Sang Lyul Min. Dynamic voltage scaling algorithm for fixed-priority real-time systems using work-demand analysis//Roh I V, Hyung eds. Proceedings of the ISLPED. Seoul, Korea, 2003: 396-401
- [4] Hakan Aydi, Pedro Mejia-Alvarez, Daniel Mossé, Rami Melhem. Dynamic and aggressive scheduling techniques for power-aware real-time systems//Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01). London, UK, 2001: 95-105
- [5] Yun Han-Saem, Kim Jihong. On energy-optimal voltage scheduling for fixed-priority hard real-time systems. ACM Transactions on Embedded Computing Systems, 2003, 2(3): 393-430
- [6] Pedro Mejia-Alvarez, Eugene Levner, Daniel Mossé. Adaptive scheduling server for power-aware real-time tasks. ACM Transactions on Embedded Computing Systems, 2004, 3(2): 284-306
- [7] Tohru Ishihara, Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors//Proceedings of the 1998 International Symposium on Low Power Elec-

- tronics and Design. Monterey, California, United States, 1998; 197-202
- [8] Padmanabhan Pillai, Kang G Shin. Real-time dynamic voltage scaling for low-power embedded operating systems//Proceedings of the 18th ACM Symposium on Operating System Principles. Chateau Lake Louise, Banff, Alberta. 2001; 35: 89-102
- [9] Zhu Yifan, Mueller Frank. Feedback edf scheduling exploiting dynamic voltage scaling//Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium. Toronto, Canada, 2004; 84-93
- [10] Yifan Zhu, Frank Mueller. Feedback edf scheduling exploiting hardware-assisted asynchronous dynamic voltage scaling//Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. Chicago, Illinois, USA, 2005; 203-212
- [11] Ravindra Jejurikar, Cristiano Pereira, Rajesh K Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems//Kahng S M, Fix L eds. Proceedings of the 41th Design Automation Conference. San Diego, CA, USA, 2004; 275-280
- [12] Lee Yann-Hang, Reddy Krishna P, Krishna C Mani. Scheduling techniques for reducing leakage power in hard real-time systems//Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003). Porto, Portugal, 2003; 105-112
- [13] Jejurikar Ravindra, Gupta Rajesh K. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems//W H J J Kahng, Martin G eds. Proceedings of the 42nd Design Automation Conference. San Diego, CA, USA, 2005; 111-116
- [14] Jejurikar Ravindra, Gupta Rajesh K. Procrastination scheduling in fixed priority real-time systems//Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. Washington, DC, USA, 2004; 57-66
- [15] Quan Gang, Niu Linwei, Xiaobo Sharon Hu, Bren Mochoki. Fixed priority scheduling for reducing overall energy on variable voltage processors//Proceedings of the 25th IEEE Real-Time Systems Symposium, Lisbon, Portugal, 2004; 309-318
- [16] Niu Linwei, Quan Gang. Reducing both dynamic and leakage energy consumption for hard real-time systems//Mahlke M J I, Zhao W, Lavagno L, S A, eds. Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. Washington DC, USA, 2004; 140-148
- [17] Wang Yun, Saksena Manas. Scheduling fixed-priority tasks with preemption threshold//Proceedings of the 6th International Workshop on Real-Time Computing and Applications Symposium. Hong Kong, China, 1999; 328-335
- [18] Wang Yun, Saksena Manas. Scalable real-time system design using preemption thresholds//Proceedings of the 21st IEEE Real-Time Systems Symposium. Hong Kong, China, 2000; 25-34
- [19] Lehoczky John P, Sha Lui, Ding Y. The rate monotonic scheduling algorithm: Exact characterization and average case behavior//Proceedings of the IEEE Real-Time Systems Symposium 1989. Orlando, Florida, USA, 1989; 166-171
- [20] Irani Sandy, Shukla Sandeep K, Gupta Rajesh K. Algorithms for power savings//Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms. Santa Monica, California, USA, 2003; 37-46
- [21] Chen Jiong-Xiong, Harji Ashif, Buhr Peter. Solution space for fixed-priority with preemption threshold//Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS' 05). San Francisco, CA, USA, 2005; 385-394
- [22] Regehr John. Scheduling tasks with mixed preemption relations for robustness to timing faults//Proceedings of the IEEE Real-Time Systems Symposium. Austin, Texas, USA, 2002; 315-326
- [23] Chen Jiong-Xiong. Extensions to fixed priority with preemption-threshold and reservation-based scheduling[Ph. D. dissertation]. University of Waterloo, Waterloo, Ontario, Canada, 2005; 42-65
- [24] Lehoczky John P. Fixed priority scheduling of periodic task sets with arbitrary deadlines//Proceedings of the IEEE Real-Time Systems Symposium. Lake Buena Vista, Florida, USA, 1990; 201-213
- [25] Audsley Neil C, Burns Neil C, Richardson M F, Wellings Andy J. Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal, 1993, 8(5); 284-292



**HE Xiao-Chuan**, born in 1977, Ph. D. candidate. His current research interests include embedded systems, real-time scheduling algorithms and ubiquitous computing.

**JIA Yan**, born in 1964, professor, Ph. D. supervisor. Her main research interests are database system, software component techniques and distributed computing.