ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.7, July 2008, pp.1837−1846
DOI: 10.3724/SP.J.1001.2008.01837

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于裁剪的弱硬实时调度算法*

吴 彤[1], 金士尧[1], 刘华锋[1], 陈积明[2+]

[1](国防科学技术大学 并行与分布处理国家重点实验室,湖南 长沙 410073)

[2](浙江大学 工业控制技术国家重点实验室,浙江 杭州 310037)

A Weakly Hard Real-Time Schedule Algorithm Based on Cutting Down

WU Tong[1], JIN Shi-Yao[1], LIU Hua-Feng[1], CHEN Ji-Ming[2+]

[1](National Laboratory of Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China)

[2](National Laboratory of Industrial Control Technology, Zhejiang University, Hangzhou 310027, China)

+ Corresponding author: E-mail: jmchen@ieee.org

Wu T, Jin SY, Liu HF, Chen JM. A weakly hard real-time schedule algorithm based on cutting down. *Journal of Software*, 2008,19(7):1837−1846. http://www.jos.org.cn/1000-9825/19/1837.htm

Abstract: Existing weakly hard real-time scheduling algorithms can not guarantee the meeting ratio of executing sequence of which the length islarger than fixed window-size. Therefore, this paper, based on the $(\overline{m},p)$ constraint, proposes an algorithm which is named as CDBS (cut-down based scheduling). Since the discrimination of the satisfiability of $(\overline{m},p)$ constraintneeds to go over the whole executing sequence of the task, it is very difficult and infeasible. For this reason, this paper introduces an efficient algorithm of cutting down the sequence, proves the correctness of the algorithm. This paper uses proper data structures so that the complexity of judgment is irrelevant to the length of sequence. Experimental results show the efficiency. Furthermore, this paper compares CDBS with other classical algorithms, such as EDF (earliest deadline first), DBP (distance-based priority), DWCS (dynamic window constraint schedule), and the results show its competence.

Key words: weakly hard real-time; dynamic failure; executing sequence; CDBS (cut-down based scheduling); turnpoint

摘 要: 针对当前弱硬实时调度算法无法保证超过窗口长度的执行序列的满足率达到一定比例的问题,基于$(\overline{m},p)$弱硬实时约束,提出了一种基于裁剪的调度算法(cut-down based scheduling,简称CDBS).由于判断$(\overline{m},p)$约束是否满足需要遍历任务的整个执行序列,因此判断复杂度很大.为此,提出一种高效的裁剪执行序列的算法,同时证明其正确性,并利用适当的数据结构,使得计算复杂度与序列长度无关,通过实验说明其降低计算复杂度的有效性.进一步与其他经典实时调度算法(EDF(earliest deadline first),DBP(distance-based priority),DWCS(dynamic window constraint schedule))进行比较,验证该算法与其他算法具有相当的性能.

关键词: 弱硬实时;动态失效;执行序列;CDBS(cut-down based scheduling);转折点

中图法分类号: TP316 **文献标识码**: A

* Supported by the National Natural Science Foundation of China under Grant Nos.60603032, 60604029 (国家自然科学基金); the NSFC-Guangdong Province of China under Grant No.U0735003 (国家自然科学基金委-广东省联合基金重点项目)

Received 2006-11-17; Accepted 2007-03-08

随着信息技术的发展和网络技术的普及,弱硬实时的应用越来越广泛,诸如实时网络传输、多媒体处理、无线传感器网络等系统都需要弱硬实时调度算法的支持.因此,研究弱硬实时调度算法具有重要意义.

1995 年,Hamdaoui和Ramanathan首先引入了(m,k)-firm约束模型[1],从有限时间内允许部分丢失截止期的角度来描述网络多媒体的服务质量问题,并给出了基于距离优先级(distance-based priority,简称DBP)的算法.同年,Koren和Shasha提出了Skip-Over算法[2],算法基于跳跃因子s,确保在每次错过截止期后都有s–1 个任务连续满足截止期.

1997 年,Wedde和Lind提出了一种分布式操作系统Melody[3],在系统中提出了关键度和敏感度的概念,系统根据任务的关键程度进行调度,实际上是确保任务连续错过的最大次数,但是并没有保证任务满足截止期的比例.

Bernat将具有窗口约束的实时系统定义为弱硬实时(weakly hard real-time)系统[4,5],定义了 4 类经典的时间约束(n,m)(在 m 个任务中完成任意 n 个)、$(\overline{n,m})$(在 m 个任务中至多有 n 个任务错过截止期)、$\langle n,m\rangle$(在 m 个任务中连续完成 n 个)和$\langle \overline{n,m}\rangle$ (在 m 个任务中至多有 n 个任务连续错过截止期),并在文献[6]中给出了BMS(bi-modal schedule)算法,算法基于μ-*pattern* 序列将任务的状态分为紧急状态和一般状态,根据约束与μ-*pattern* 计算关键度函数,其约束具有固定窗口大小.

Richard West等人基于(x,y)约束(等价于Bernat提出的 $(\overline{n,m})$ 约束)提出DWCS算法[7−10],用于保证动态实时窗口内任务的丢失数,算法用丢失率x/y来描述系统对任务错过截止期的容忍程度,使用表格驱动决定其优先级,他们进一步证明了服务的最大延时.Zhang等人在DWCS(dynamic window constraint schedule)的基础上,提出了一种松弛的(m,k)模型,从任务截止期和任务的窗口约束中导出虚拟截止期,进而提出虚拟截止期调度(virtual deadline schedule,简称VDS)[11].

尽管在弱硬实时调度算法的研究中已经取得了巨大的成果,但是,上述算法都基于定长窗口,从截止期满足和任务丢失的角度进行调度,或者是仅根据任务的连续丢失情况进行调度,都没有综合考虑任务连续丢失和整个时间区域上的任务丢失率.由此,陈积明提出了一种新的弱硬实时约束 $(\overline{m},p)$ [12],即任务在其任意连续窗口中,连续超过截止期的次数不超过 m 次,且满足截止期的比例不小于 p.

这种约束面向这样一种应用:给定一组任务集,对于每个任务实例有如下要求:允许错过截止期,但是连续错过截止期的次数不能超过 m 次,同时,为了使应用成功完成,还要保证任务实例满足截止期的次数达到一定的比例.其实,所有弱硬实时约束都是为了达到这种目的,例如(m,k)-firm 模型在任意长度为 k 的序列内满足比例为m/k,而 DWCS 算法则在任意长度为 y 的序列内满足比例为$(y–x)/y$.但是,上述模型在序列长度大于窗口长度时都无法保证这种比例仍然存在.例如,序列 00111111110011111111 满足 DWCS 的约束(2,10),即其执行序列中任意长度为 10 的序列的满足比例不低于 80%,但是我们可以看到,上述序列中存在一个长度为 12 的子序列001111111100,其满足截止期的比例仅为 66.7%.

上面的实例说明,如果单从固定时间窗口的角度进行调度,可能会导致在某个窗口内实时性能突然下降,有时甚至会影响系统的运行.为了解决这个问题,有必要研究基于 $(\overline{m},p)$ 约束的调度算法.

但是,判断这种约束需要检查整个执行序列,即执行一次调度的计算复杂度与已经调度的任务实例总数成正比.而实际上,实时调度中的任务实例数通常很大,因此,在算法中完全检查是不可行的.本文提出了一种高效的剪裁执行序列的方法,并进一步利用适当的数据结构,使得复杂度与序列长度无关.实验结果表明,算法有效地降低了检查的复杂度.

本文第 1 节介绍预备知识和符号说明.第 2 节具体描述算法,证明裁剪算法的正确性,并分析算法的复杂度.第 3 节通过实验分析算法的复杂性,并与其他实时算法进行比较.第 4 节对全文作简单总结.

1 预备知识

定义 1. 任务τ_i的执行情况可以用一个二进制字符串$\Pi_i=\{\alpha_{i1}\alpha_{i2}\ldots\alpha_{ij}\ldots\}$来表示.其中,$\alpha_{ij}$=1 表示任务实例$\tau_{ij}$满足截止期;$\alpha_{ij}$=0 表示任务实例$\tau_{ij}$超时.称字符串$\Pi_i$为任务$\tau_i$的执行序列.以$\alpha_i$表示$\Pi_i$的一个子序列.在不引起混

淆的情况下,在执行序列Π_i中,α_{ij}表示任务实例τ_{ij}的执行情况;而在执行子序列α_i中,α_{ij}表示子序列α_i中的第$j(j\geq 1)$个请求的执行情况.

定义 2. 指定Σ为集合{0,1}中的元素,Σ^+为长度大于 0 的二进制序列集.Σ^n则是长度为n的二进制序列集.对于任一序列$\omega\in\Sigma^+$,该序列的长度$l(\omega)=|\omega|$,也就是该序列所组成的二进制位数.如果$\omega\in\Sigma^n$,则$l(\omega)=n$,称这种序列为n序列.序列ω的长度为k的子序列集,表示为$S^k(\omega)$.

定义 3. 二进制序列ω中 1 的个数表示为$l^1(\omega)$,0 的个数表示为$l^0(\omega)$.

根据文献[12]中的定义,如果ω的任意子k序列中不包含长度大于m的全 0 序列($k\geq\left\lceil\frac{m}{1-p}\right\rceil$),且满足截止期的比例不小于$p$,那么,$n$序列$\omega$满足约束$(\overline{m},p)$.即需要证明如下条件成立:

当$p\neq 1$时,$\omega\models(\overline{m},p)\Leftrightarrow\forall\omega'\in S^k(\omega),0^{m+1}\notin S^{m+1}(\omega'),\text{且}\frac{l^1(\omega')}{l(\omega')}\geq p$;

当$p=1$时,$\omega\models(\overline{m},p)\Leftrightarrow l^1(\omega)=n$.

定义 4. 任务τ_i的基准窗口大小定义如下:

$$w_i=\begin{cases}1, & p_i=1\\ \dfrac{m_i}{1-p_i}, & p_i\neq 1\end{cases}.$$

定义 5. 在二进制序列中,由 1 变为 0 时 0 的位置称为转折点.

定义 6. 在本文中,任务τ_i的时间参数定义如下:T_i表示任务的周期,D_i表示任务的相对截止期,C_i表示任务的计算时间,m_i和p_i组合在一起表示CDBS算法的$(\overline{m}_i,p_i)$约束,m_i和k_i组合在一起表示DBP算法的(m_i,k_i)约束,x_i和y_i组合在一起表示DWCS算法的(x_i,y_i)约束.

2 基于裁剪的调度算法

2.1 序列裁剪

判断$(\overline{m},p)$约束是否满足,需要检查任务τ_i的所有长度不小于基准窗口w_i的执行子序列α_i是否满足约束,在实际调度过程中,判断的复杂度随着任务实例的数量递增,显然这是不可行的,必须对任务的执行序列进行剪裁,并且要确保剪裁后的序列不会对违背约束的子序列漏检.

在每次任务实例的截止期到达时,根据其执行成功与否更新测试序列,采用如下方法来判断测试序列是否满足约束,序列测试算法的基本思路如下:

输入:任务τ_i的测试序列α_i;

1. 如果α_i的长度小于w_i,算法返回;
2. 裁剪测试序列;
3. 如果α_i的长度不小于w_i:
   1) 取α_i的后w_i位组成子序列α_i',如果$\frac{l^1(\alpha_i')}{l(\alpha')}\geq p_i$,则满足约束,否则违背约束;
   2) 计算剩余序列中每个转折点以后的长度不小于w_i的子序列的满足比例,如果存在比例小于p_i的子序列,则违背约束,否则满足约束.

裁剪测试序列α_i算法的基本思路如下:

1. 如果$l(\alpha_i)\leq w_i$,则返回;
2. 由前向后逐一选取序列中的转折点,如果转折点后面的子序列的长度不小于w_i,并且前面部分的满足比例不小于p_i,也不低于后面部分的满足比例,则删除前面的子序列;
3. 在α_i中从后向前取长度为w_i的子序列α_i',如果剩余序列的满足比例不小于p_i,而且不低于α_i'的满足比例,则删除α_i'前面的子序列.

为了证明算法不会漏检,首先证明如下两个引理:

引理 1. 对于二进制序列ω,将其进行划分,使得前面 k 位组成序列ω',其余位组成序列ω'',且

$$\frac{l^1(\omega')}{l(\omega')} \geq \frac{l^1(\omega'')}{l(\omega'')} \geq p,$$

那么,对于任意序列 ω''',如果 $\frac{l^1(\omega''\omega''')}{l(\omega''\omega''')} \geq p$,则必然有 $\frac{l^1(\omega\omega''')}{l(\omega\omega''')} \geq p$;如果 $\frac{l^1(\omega\omega''')}{l(\omega\omega''')} < p$,则必然有 $\frac{l^1(\omega''\omega''')}{l(\omega''\omega''')} < p$.

证明:

如果 $\frac{l^1(\omega''\omega''')}{l(\omega''\omega''')} \geq p$,那么 $\frac{l^1(\omega\omega''')}{l(\omega\omega''')} = \frac{l^1(\omega')+l^1(\omega''\omega''')}{l(\omega')+l(\omega''\omega''')} \geq \frac{p\cdot l(\omega')+p\cdot l(\omega''\omega''')}{l(\omega')+l(\omega''\omega''')} = p$, 显然成立.

如果 $\frac{l^1(\omega\omega''')}{l(\omega\omega''')} < p$,则 $\frac{l^1(\omega\omega''')}{l(\omega\omega''')} = \frac{l^1(\omega')+l^1(\omega'')+l^1(\omega''')}{l(\omega')+l(\omega'')+l(\omega''')}$,而 $\frac{l^1(\omega''\omega''')}{l(\omega''\omega''')} = \frac{l^1(\omega'')+l^1(\omega''')}{l(\omega'')+l(\omega''')}$,

两式相减并合并,可得到下式:

$$\frac{l^1(\omega\omega''')}{l(\omega\omega''')} - \frac{l^1(\omega''\omega''')}{(\omega''\omega''')} = \frac{(l^1(\omega')l(\omega'')+l^1(\omega')l(\omega'''))-(l(\omega')l^1(\omega'')+l^1(\omega''')l(\omega'))}{(l(\omega')+l(\omega'')+l(\omega'''))(l(\omega'')+l(\omega'''))}.$$

因为

$$\frac{l^1(\omega')l(\omega'')}{l(\omega')l^1(\omega'')} \geq \frac{l^1(\omega'')}{l(\omega'')}\cdot\frac{l(\omega'')}{l^1(\omega'')} = 1 \tag{1}$$

显然有

$$\frac{l^1(\omega''')}{l(\omega''')} < p \leq \frac{l^1(\omega')}{l(\omega')},$$

则有

$$\frac{l^1(\omega')l(\omega''')}{l(\omega')l^1(\omega''')} \geq \frac{l^1(\omega')}{l(\omega')}\cdot\frac{l(\omega')}{l^1(\omega')} = 1 \tag{2}$$

由式(1)、式(2)可得

$$\frac{l^1(\omega\omega''')}{l(\omega\omega''')} - \frac{l^1(\omega''\omega''')}{l(\omega''\omega''')} \geq 0,$$

所以, $\frac{l^1(\omega''\omega''')}{l(\omega''\omega''')} \leq \frac{l^1(\omega\omega''')}{l(\omega\omega''')} < p$ 成立. □

引理 2. 如果有违背约束的子序列,那么在该子序列中,必然存在一个首位为 0 的违背约束的子序列,或者存在一个长度等于基准窗口长度 w 的违背约束的子序列.

证明:

如果序列中存在一个首位为 1 的违背约束的子序列ω,则有 $l(\omega)\geq w$,且 $\frac{l^1(\omega)}{l(\omega)} < p$.

在ω中必然可以找到第 1 个 0 的位置,将这个 0 前面的部分删除,得到一个首位为 0 的新子序列ω'.

1) 如果 $l(\omega')\geq w$,而且由于上述操作不会减少ω'中的 0 的数量,那么显然有 $p > \frac{l^1(\omega)}{l(\omega)} \geq \frac{l^1(\omega')}{l(\omega')}$,因此, ω' 是一个首位为 0 的违背约束的子序列;

2) 如果 $l(\omega')<w$,那么可以取ω的后 w 位组成新子序列ω'',显然,ω''是从ω中删去若干个 1 而得到的, 则有

$$p > \frac{l^1(\omega)}{l(\omega)} \geq \frac{l^1(\omega'')}{l(\omega'')},$$

因此,ω''也是一个长度等于基准窗口长度 w 的违背约束的子序列.

由 1)、2),命题得证. □

定理 1. 根据序列测试算法和裁剪算法,可以确保:如果任务τ_i的执行序列存在违背约束 $(\overline{m}_i, p_i)$ 的子序列,

那么在裁剪后的执行序列中,也必然存在违背约束 $(\overline{m}_i, p_i)$ 的子序列.

证明:执行序列随着任务实例的到达逐渐增长,在长度达到w_i之前不存在违背约束的子序列,因此不存在漏检.

在执行序列长度达到w_i以后,根据裁剪算法的基本思路,算法不会裁掉满足比例小于p_i的子序列,而且会保证测试序列的长度不小于w_i.因此,裁剪算法不会裁掉测试序列中违背约束的子序列.根据引理 2,如果测试序列包含违背约束的子序列,那么一定包含一个首位为 0 的违背约束的子序列或者长度等于w_i的违背约束的子序列,因此,序列测试算法不会漏检.

假设在任务的上一实例调度后,序列测试算法没有检测到违背约束的子序列.那么在已经产生的序列中不会有违背约束的子序列,因此,唯一有可能出现漏检的原因就是违背约束的子序列只有一部分进入测试序列,即出现图 1 中的两种情况,下面分别证明这两种情况都不可能出现漏检.

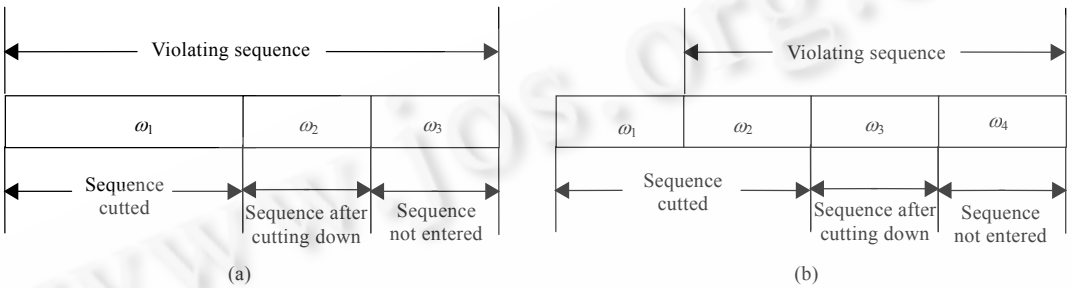

Fig.1 Partial violating sub-sequence enters into test sequence

图 1 部分违背约束的子序列进入测试序列的情况

对于图 1(a)中的情况,由引理 1 和裁剪过程可知,如果 $\frac{l^1(\omega_1\omega_2\omega_3)}{l(\omega_1\omega_2\omega_3)} < p$,那么必然有 $\frac{l^1(\omega_2\omega_3)}{l(\omega_2\omega_3)} < p$, 因为$\omega_2\omega_3$的长度大于$w_i$,因此,$\omega_2\omega_3$也是一个违背约束的子序列,即这种情况可以通过序列测试发现.

对于图 1(b)中的情况,根据裁剪过程和序列测试算法,存在以下不等式:

$$\frac{l^1(\omega_1\omega_2)}{l(\omega_1\omega_2)} \geq \frac{l^1(\omega_3)}{l(\omega_3)}, \frac{l^1(\omega_1\omega_2)}{l(\omega_1\omega_2)} \geq p, \frac{l^1(\omega_2\omega_3\omega_4)}{l(\omega_2\omega_3\omega_4)} < p.$$

因为 $\frac{l^1(\omega_3\omega_4)}{l(\omega_3\omega_4)} \geq p$,而且 $\frac{l^1(\omega_3)}{l(\omega_3)} \geq p$ (ω_3的长度大于w_i,而且ω_3不是违背约束的子序列),那么显然有

$$\frac{l^1(\omega_2)}{l(\omega_2)} < p,$$

因此有

$$\frac{l^1(\omega_1)}{l(\omega_1)} \geq \frac{l^1(\omega_1\omega_2)}{l(\omega_1\omega_2)} \geq \frac{l^1(\omega_3)}{l(\omega_3)} \geq \frac{l^1(\omega_2\omega_3)}{l(\omega_2\omega_3)}$$

成立,那么根据裁剪过程可知,被裁剪的序列为ω_1,而不是$\omega_1\omega_2$,即不可能出现该情况.

综上可知,在裁剪算法和序列测试算法的配合下,不会对违背约束的子序列漏检. □

2.2 优先级分配

对于约束 $(\overline{m}, p)$,需要同时考虑不满足截止期的次数和满足截止期的比例,为了使算法能够刻画不满足截止期的实例对任务优先级的影响,算法根据距离连续 m 次错过截止期的距离来定义优先级,距离越远,优先级越低,图 2 为 m=2 时的状态转换图.

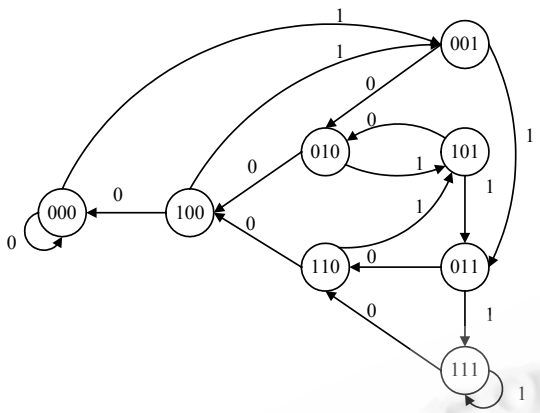

Fig.2 State transition diagram of m=2

图 2 m=2 时的状态转换图

任务τ_i的最近的长度为m_i+1 的执行子序列α_i,$s_i(\alpha_i)$是子序列α_i中最右边的 1 的位置,如果α_i中没有 1,那么$s_i(\alpha_i)=m_i+1$.例如,对于约束 $(\overline{2},0.8)$,$s_i(001)=1$,而$s_i(000)=3$.显然,$m_i+1-s_i(\alpha_i)$为此时距离连续m_i次错过截止期的距离,因此定义任务τ_i的优先级r_i如下:

$$r_i=\begin{cases} m_i+1-s_i(\alpha_i), & l(\alpha_i)\geq m_i+1 \\ m_i+1-l(\alpha_i), & l(\alpha_i)<m_i+1 \end{cases} \tag{3}$$

任务可能处于 4 种状态:

(1) 连续错过次数满足,成功率满足;

(2) 连续错过次数满足,成功率不满足;

(3) 连续错过次数不满足,成功率满足;

(4) 连续错过次数不满足,成功率不满足.

在所有任务中,按照 4,3,2,1 的优先顺序进行调度,处于同一种状态的任务,根据表 1 进行调度.

Table 1 Precedence rules are applied top-to-bottom for the tasks in the same state

表 1 同一状态下任务按照自顶向下的顺序调度

Pariwise job ordering
Highest priority first
Equal priority, earliest deadline first
Equal priority and equal deadline, the smallest the difference between the current success ratio and the target success ratio first
All other cases: First-Come-First-Serve

2.3 算法描述

我们定义如下数据结构:

```
typedef enum {
    MISS=0;                          //错过截止期
    MEET=1;                          //满足截止期
} STATUS;                            //执行情况
struct TestSequence{                 //测试序列
    int nTotalTask;                  //测试序列中任务实例的总数
    STATUS arrayStatus[BASE_SIZE];   //保存最后 w 次执行情况的数组
    TurnPoint*pTurnPoint;            //测试序列中的转折点链表
}
```

```
struct TurnPoint{
    int nTotalTask;                 //本转折点后面的子序列中包含的任务总数
    int nSuccessTask;               //本转折点后面的子序列中包含的成功任务数
    TurnPoint*pNext;                //后续转折点
    TurnPoint*pPre;                 //前驱转折点
}
```

CDBS 算法包含 3 个函数:CreateTurnPoint,CalculateProbability 和 CutDown.

函数 CreateTurnPoint:构建转折点.

输入:当前任务实例是否成功,任务编号 i;

1. τ_i的测试序列中的任务实例总数加 1,更新测试序列中的*arrayStatus*;
2. 如果任务实例成功,那么,τ_i测试序列中所有转折点任务总数加 1,成功任务数加 1;
3. 如果任务实例不成功,那么:
   1) 如果τ_i测试序列中的第 1 个转折点的任务成功数等于 0,那么,τ_i测试序列中所有转折点的任务总数加 1;
   2) 否则,为τ_i测试序列新建一个转折点(任务总数为 1,成功任务数为 0),放在转折点链表的链首,τ_i测试序列中其他转折点任务总数加 1.

函数*CalculateProbability*:返回执行序列中长度不小于w_i的子序列中最小的满足比例.

输入:任务编号 i;

输出:序列中所包含的长度不小于w_i的子序列中最小的满足比例.

1. 按照τ_i测试序列中转折点链表的逆序计算成功执行率,当转折点的任务总数小于w_i时停止;
2. 计算τ_i测试序列中后w_i位组成的子序列的成功执行率;
3. 返回 1 步、2 步计算的成功执行率的最小值,如果序列总长度小于w_i,则返回 100%.

函数 *CutDown*:序列裁剪.

输入:任务编号 i;

1. 如果τ_i当前测试序列的长度不大于w_i,则算法停止;
2. 如果转折点链表为空,则转到 6;
3. 按照τ_i测试序列中转折点链表的逆序依次取出转折点,如果取空,则转到 6;
4. 如果所取的转折点的任务总数小于w_i,则转到 6;
5. 所取的转折点前的执行子序列的成功执行率为 p',后面子序列的成功执行率为 p'':
   1) 如果$p'\geq p''$,而且$p'\geq p_i$,那么删除所取转折点后面的所有转折点,转到 3;
   2) 如果 $p'<p''$,则转到 3;
6. 测试序列中最后w_i个任务实例的成功执行率为p',前面部分的成功执行率为p'',如果$p'\geq p''$,而且$p'\geq p_i$,那么删除前面部分所包含的转折点;
7. 算法停止.

调度算法.

1. 根据第 2.2 节调度任务;
2. 根据任务 i 的实例的执行成功或失败,更新任务状态:
   1) 构建转折点,*CreateTurnPoint*(i,*status*);
   2) 裁剪执行序列,*CutDown*(i);
   3) 计算执行成功率,*CalculateProbability*(i);
   4) 根据公式(3)计算任务的优先级;
3. 转到 1.

2.4 复杂度分析

设基准窗口长度为 w,转折点数量为 k,m 与约束中的 m 同义,那么,

时间复杂度:创建转折点的时间复杂度为$O(k)$,计算成功执行率的时间复杂度为$O(k)+O(w)$,序列裁剪的时间复杂度为$O(k)+O(w)$,计算任务优先级的时间复杂度为$O(m+1)$.因此,任务τ_i每一步调度的时间复杂度为

$$O(k_i)+(O(k_i)+O(w_i))+(O(k_i)+O(w_i))+O(m_i+1)=O(k_i)+O(w_i)+O(m_i).$$

空间复杂度:任务τ_i所占用的空间为$O(w_i)+O(k_i)$.

显然,w_i和m_i对于一个任务来说是常数,因此,影响复杂度的主要因素是k_i,即算法的复杂度与测试序列中的转折点的数量呈线性关系.下一节将对转折点的数量进行模拟分析.

3 算法比较和实验结果

3.1 算法复杂度实验

(1) 随机选取 100 组任务,每组包含 20 个任务,固定任务的计算时间 C 为 1,相对截止期 $D=T$,随机分配任务的周期 T,使其在[1,100]区间内变化,同时使得每组任务的利用率 u 在[0.7,1.5]区间内变化,固定 m=3,目标成功执行率 p=70%.图 3、图 4 给出了基于裁剪(CDBS)和不进行裁剪(NO_CUT)情况下的转折点数量变化图.实验结果表明:当利用率 $u<1$ 时,所有任务实例调度成功,所以转折点个数为 0,基于裁剪与不进行裁剪的复杂度相同;当 $1<u<1.4$ 时,执行裁剪时转折点个数变化缓慢,小于基准窗口长度,而不执行裁剪时转折点个数变化较快,远远大于基准窗口长度;当 $u>1.4$ 时,由于这时 $u\cdot p>1$,即系统满足约束所必需的 CPU 利用率已经超过 1,因此,这时各类任务频繁地违背约束,导致转折点数量激增,但基于裁剪的情况仍然优于不执行裁剪的情况.实验结果表明:当 $u\cdot p<1$ 时,CDBS 算法的复杂度由基准窗口长度决定,即 $O(1)$;而当 $u\cdot p>1$ 时,CDBS 算法的复杂度由转折点数量决定,且与转折点个数成正比.

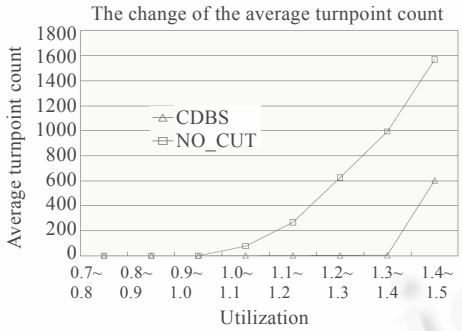

Fig.3 Change of the average turnpoint count

图 3 转折点平均个数变化图

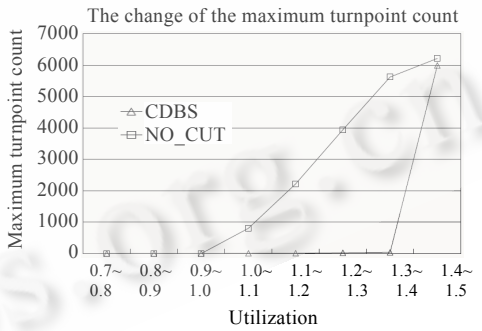

Fig.4 Change of the maximum turnpoint count

图 4 转折点最大个数变化图

(2) 随机选取 100 组任务,每组包含 20 个任务,固定任务的计算时间 C 为 1,相对截止期 $D=T$,随机分配任务的周期 T,使其在[1,100]区间内变化,同时使得每组任务的利用率 u 在 1.2 附近变化,m 从 1 到 3 变化,目标成功执行率 p 从 50%增加到 90%,每次增加 5%.图 5、图 6 给出基于裁剪(CDBS)和不进行裁剪(NO_CUT)情况下,转折点数量随 m 和 p 的变化情况.从图中可以看出:当 $u\cdot p<1$ 时,执行裁剪时转折点平均个数小于基准窗口长度,转折点最大个数变化缓慢,而不执行裁剪时转折点的平均个数和最大个数都远远大于基准窗口;当 $u\cdot p>1$ 时,转折点平均个数和最大个数激增,执行裁剪的情况仍然优于不执行裁剪的情况.CDBS 算法的调度费用与实验 1(1)中的结果相同.

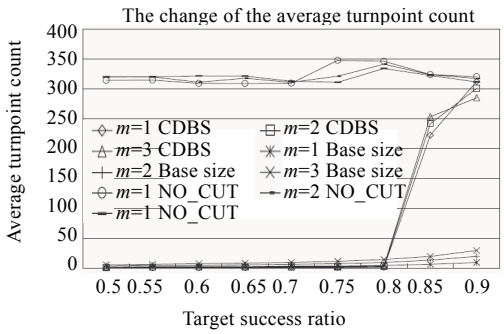

Fig.5 Change of the average turnpoint count

图 5 转折点平均个数变化图

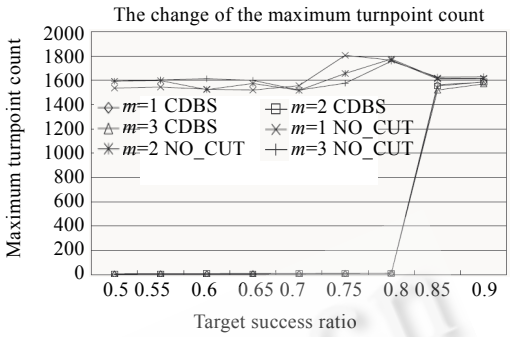

Fig.6 Change of the maximum turnpoint count

图 6 转折点最大个数变化图

3.2 性能比较

本节将 CDBS 与 EDF,DBP 和 DWCS 等经典算法进行性能的比较,为了合理地评估各种算法,定义如下评估参数:

- (*m*,*k*)动态失效:由于 DBP 和 DWCS 都是基于固定窗口大小的弱硬实时约束进行设计的,为了使性能比较更具说服力,这里统一根据(*m*,*k*)约束进行动态失效的判断,将执行序列记录下来,逐一判断在每个时间窗口内是否违背约束,得到(*m*,*k*)动态失效情况;
- 成功执行率:由于某些调度算法会放弃某些计算时间长的任务从而降低整体的动态失效次数,有时甚至会把某些任务饿死,因此,为了能够衡量调度算法是否公平,引入每类任务的成功执行率(最小成功率)作为评估参数.

随机选取 100 组任务,每组包含 20 个任务,固定任务的计算时间 C 为 1,相对截止期 $D=T$,随机分配任务的周期 T,使其在[1,100]区间内变化,同时使得每组任务的利用率 u 在[0.7,1.6]区间内变化,固定 m=3,目标成功执行率 p=70%,DBP 算法中任务的约束统一为(7,10),DWCS 算法中任务的约束统一为(3,10).图 7、图 8 给出了(*m*,*k*)动态失效率和最小成功率的变化情况.可以看出,与其他算法比较,CDBS 算法同时提供了适当的(*m*,*k*)动态失效率和最小成功率,提供了折衷的性能.

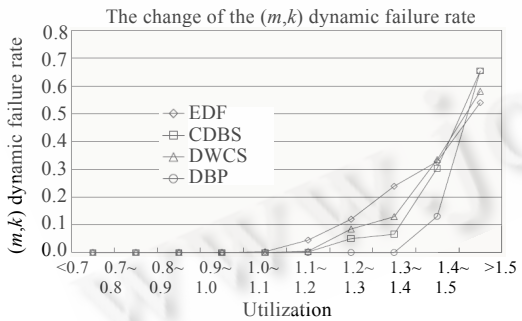

Fig.7 Change of the (*m*,*k*) dynamic failure rate

图 7 (*m*,*k*)动态失效率变化图

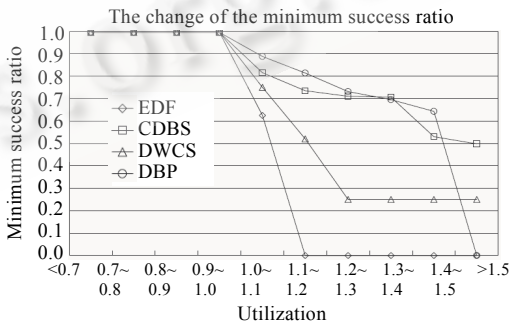

Fig.8 Change of the minimum success ratio

图 8 最小成功率变化图

4 结束语

本文基于 $(\overline{m},p)$ 弱硬实时约束,提出了一种不受限于固定窗口长度的基于裁剪的弱硬实时调度算法,对任务的执行序列进行有效的裁剪,并引入转折点的概念,使得对 $(\overline{m},p)$ 弱硬实时约束满足性判别的复杂度大幅度降低.文中给出了裁剪算法正确性的证明,并通过实验验证了其有效性;基于距离 m 次连续错过截止期的距离分配任务的优先级,并结合任务可能出现的 4 种状态进行调度.最后,将算法与 EDF,DWCS,DBP 等算法进行比

较,CDBS 算法在动态失效和最小成功率方面都提供了适当的折衷,与其他算法相比,具有相当的性能.

References:

[1] Hamdaoui M, Ramanathan P. A dynamic priority assignment technique for streams with (*m*,*k*)-firm deadlines. IEEE Trans. on Computers, 1995,44(4):1443−1451.

[2] Koren G, Shasha D. Skip-Over: Algorithm and complexity for overloaded systems that allow skips. In: Proc. of the 16th IEEE Real-Time Systems Symp. Pisa: IEEE Computer Society Press, 1995. 110−117. http://doi.ieeecomputersociety.org/resolve?ref_id=doi:10.1109/REAL.1995.495201&rfr_id=trans/td/2003/05/l0433.xml

[3] Wedde H, Lind J. Building large, complex, distributed safety-critical operating systems. Real-Time Systems, 1997,13(3):277−302.

[4] Bernat G. Specification and analysis of weakly hard real-time system [Ph.D. Thesis]. University of York, 1998.

[5] Bernat G, Burns A, Llamosi A. Weakly hard real-time systems. IEEE Trans. on Computers, 2001,50(4):308−321.

[6] Bernat G, Cayssials R. Guaranteed on-line weakly-hard real-time systems. In: Proc. of the 22nd IEEE Real-Time Systems Symp. London: IEEE Computer Society Press, 2001. 25−35. http://csdl.computer.org/redirectDL.jsp?path=proceedings/rtss/2001/1420/00/14200025abs.htm

[7] West R, Zhang YT, Schwan K, Poellabauer C. Dynamic window-constrained scheduling of real-time streams in media servers. IEEE Trans. on Computers, 2004,53(6):744−759.

[8] West R, Schwan K, Poellabauer C. Scalable scheduling support for loss and delay constrained media streams. In: Proc. of the 5th IEEE Real-Time Technology and Applications Symp. Vancouver: IEEE Computer Society Press, 1999. 24−33. http://csdl.computer.org/comp/proceedings/rtas/1999/0194/00/01940024abs.htm

[9] West R, Poellabauer C. Analysis of a window constrained scheduler for real-time and best-effort packet streams. In: Proc. of the 21st IEEE Real-Time Systems Symp. Orlando: IEEE Computer Society Press, 2000. 239−248. http://doi.ieeecomputersociety.org/resolve?ref_id=doi:10.1109/REAL.2000.896013&rfr_id=trans/td/2003/07/l0655.xml

[10] West R, Ganev I, Schwan K. Window-Constrained process scheduling for linux systems. In: Proc. of the 3rd Real-Time Linux Workshop. Milan: Real Time Linux Foundation, 2001. http://www.realtimelinuxfoundation.org/events/rtlws-2001/papers.html#PAPER_West

[11] Zhang YT, West R, Qi X. A virtual deadline scheduler for window-constrained service guarantees. In: Proc. of the 25th IEEE Real-Time Systems Symp. Lisbon: IEEE Computer Society Press, 2004. 151−160. http://csdl.computer.org/comp/proceedings/rtss/2004/2247/00/22470151abs.htm

[12] Chen JM, Song YQ, Sun YX. Research on constraint specification of weakly hard real-time system. Journal of Software, 2006, 17(12):2601−2608 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/17/2601.htm

附中文参考文献:

[12] 陈积明,宋叶琼,孙优贤.弱硬实时系统约束规范研究.软件学报,2006,17(12):2601−2608. http://www.jos.org.cn/1000-9825/17/2601.htm

吴彤(1979－),男,辽宁沈阳人,博士生,主要研究领域为弱硬实时调度.

刘华锋(1976－),男,博士,主要研究领域为传感器网络,虚拟现实.

金士尧(1937－),男,教授,博士生导师,CCF 高级会员,主要研究领域为实时系统,分布计算与仿真.

陈积明(1978－),男,博士,副教授,主要研究领域为弱硬实时调度,无线传感器/执行器网络.