임의 주기를 가지는 실시간 멀티 태스크를 위한 체크포인트 구간 최적화

논 문
60-1-30

Optimizing Checkpoint Intervals for Real-Time Multi-Tasks with Arbitrary Periods

곽 성 우* · 양 정 민†
(Seong Woo Kwak · Jung-Min Yang)

Abstract - This paper presents an optimal checkpoint strategy for fault-tolerance in real-time systems. In our environment, multiple real-time tasks with arbitrary periods are scheduled in the system by Rate Monotonic (RM) algorithm, and checkpoints are inserted at a constant interval in each task while the width of interval is different with respect to the task. We propose a method to determine the optimal checkpoint interval for each task so that the probability of completing all the tasks is maximized. Whenever a fault occurs to a checkpoint interval of a task, the execution time of the task would be prolonged by rollback and re-execution of checkpoints. Our scheme includes the schedulability test to examine whether a task can be completed with an extended execution time. A numerical experiment is conducted to demonstrate the applicability of the proposed scheme.

Key Words : Checkpointing, Real-time systems, Fault-tolerance, Rate Monotonic (RM) scheduling, Arbitrary period

1. 서 론

실시간 시스템(real-time systems)은 태스크(task)들의 실행이 주어진 데드라인(deadline)까지 완료되어야 하는 시스템을 말한다[1]. 디지털 제어(digital control), 원거리 통신 시스템(telecommunication systems), 항공전자공학(avionic) 등이 실시간 시스템 이론이 적용되는 대표적인 분야이다. 실시간 시스템의 대상은 주로 고신뢰도(safety-critical)를 요구하는 환경에서 작동되므로 여러 가지 고장에 대한 즉각적인 대처 능력은 실시간 시스템이 보유해야 할 필수적인 요소이다.

체크포인트 기법(checkpointing)은 실시간 시스템의 내고장성(fault tolerance)을 구현하는 대표적인 방법 중의 하나이다. 각 태스크 내부에 삽입된 체크포인트는 현재까지 수행된 태스크 모듈의 정보를 저장한다. 태스크 실행 중 고장(fault)이 발생하면 프로세서는 고장이 발생하기 직전에 위치한 체크포인트로 되돌아가(rollback) 체크포인트에 저장된 정보를 받아서 실행을 재개한다.

이렇듯 체크포인트 기법은 과도 고장(transient fault)에 대한 재수행 부하를 줄여서 고장 극복 속도를 높이는 장점이 있다. 하지만 프로세서가 체크포인트를 거칠 때마다 태스크의 정보를 저장하고 고장 발생을 검사하기 위한 시간, 즉 체크포인트 오버헤드(overhead)가 늘어난다. 따라서 시스템이 적절한 성능을 낼 수 있도록 태스크에 삽입되는 체크포인트의 개수와 구간을 알맞게 결정하는 일은 체크포인트 기법에서 매우 중요하다.

체크포인트 구간 결정은 체크포인트와 관련된 성능 지수를 정의하여 이 성능 지수를 최대화하는 구간을 계산하는 최적화 문제로 해석할 수 있다. 기존 연구에서 적용된 성능 지수로는 체크포인트 삽입으로 늘어나는 기대 시간[2],[3], 태스크의 availability[4], response time[5] 등이 있다. 한편 저자의 선행 연구[6]에서는 확률적으로 일어나는 과도 고장의 모든 발생 경우에 대해서 태스크의 스케줄링 성공 확률을 최대화하는 체크포인트 구간을 구하였다.

본 연구는 실시간 주기적 멀티 태스크(multi-task)를 위한 최적의 체크포인트 구간을 결정하는 새로운 방법을 제시한다. 본 연구에서 사용하는 실시간 시스템의 성능 지수 역시 태스크의 스케줄링 성공 확률이다. 그러나 이번 연구에서는 임의 주기(arbitrary period)를 가지는 멀티 태스크에 대한 문제를 다룬다. 선행 연구[6]는 멀티 태스크의 주기들이 “harmonic period” 조건[7], 즉 임의의 주기가 다른 임의의 주기의 2^n배가 된다는 조건 하에 이루어졌다. 실제 운용되고 있는 실시간 시스템에서는 harmonic period 조건을 만족시키지 않는 멀티 태스크가 일반적이므로 [6]의 결과는 매우 제한적이라고 말할 수 있다. 기존 대부분의 연구들이 단일 태스크[2],[3] 또는 주기에 제한조건이 있는 멀티 태스크[7],[9],[10]에 대하여 최적의 체크포인터 구간을 구했으나 본 연구는 태스크의 주기에 제한이 없는 일반적인 태스크를 대상으로 한다.

본 논문에서는 먼저 실시간 멀티 태스크에서 발생하는 과도 고장을 확률적으로 정의한 후 체크포인트 재수행 횟수에 따른 개별 태스크 실행이 끝날 확률을 구한다. 그런 다음 각 태스크가 가지는 여유 시간(slack time)을 유도하여 과도 고장이 태스크에 발생하는 모든 경우에 대하여 전체 태스크

* 정 회 원 : 계명대 전자공학과 부교수
† 교신저자, 정회원 : 대구가톨릭대 전자공학과 부교수
E-mail : jmyang@cu.ac.kr
접수일자 : 2010년 6월 1일
최종완료 : 2010년 10월 19일

의 실행이 끝날 확률을 최대화하는 체크포인트 구간을 찾는다. 또한 몇 가지 예제에 대한 모의실험을 통하여 제안된 방법의 우수성과 적용 가능성을 검증한다.

2. 모델링

2.1 실시간 멀티 태스크

먼저 본 논문에서 정의하고 사용할 매개 변수를 요약하면 다음 표와 같다.

표 1 논문에서 정의된 매개 변수 요약.

Table 1 Parameters used in this paper.

Notation	Description
$e_{i,j}$	태스크 T_i의 j 번째 job $J_{i,j}$의 실제 실행시간
p_k	태스크 T_k의 주기
p_i	구간 Δ_i에서 과도 고장이 발생하지 않을 확률
q_i	구간 Δ_i에서 과도 고장이 한 번 이상 발생할 확률
$l_{i,j}$	Job $J_{i,j}$에서 재수행되는 구간 횟수
L_i	태스크 T_i의 재수행 벡터 $L_i = [l_{i,1}\ l_{i,2}\ \dots\ l_{i,vi}]$
$\Psi_{ij}(l_{i,j})$	$l_{i,j}$번의 재수행 구간을 가진 $J_{i,j}$의 실행이 끝날 확률
$\Psi_i(L_i)$	재수행 벡터 L_i를 가진 태스크 T_i의 모든 job의 실행이 끝날 확률
$d_{i,j}$	Job $J_{i,j}$가 준수해야 할 절대 데드라인
$d^e_{i,j}$	$J_{i,j}$가 실제 만족시켜야 할 상대적인 데드라인
$\sigma_{i,j}(0)$	스케줄링 시점 0부터 j번째 절대 데드라인까지 측정한 태스크 T_i의 누적 여유시간
$s_{i,j}$	T_i의 j번째 주기에서 측정한 여유시간
$\phi_i(t)$	시각 t 이후에 실행이 시작되는 태스크 T_i의 job 중 최소 release 시각과 t와의 시간차이
$\Gamma(L_1,\dots,L_m)$	재수행 벡터가 $L_1,\dots,L_m$으로 주어질 때 전체 태스크 $\mathbf{T}$에 대한 RM 스케줄링 결과가 유효한지를 나타내는 함수
$\Psi(L_1,\dots,L_m)$	재수행 벡터가 $L_1,\dots,L_m$으로 주어질 때 모든 태스크의 job들이 [0, H] 동안 실행을 끝낼 확률
$P(\mathbf{T},H)$	멀티 태스크 $\mathbf{T}$의 모든 job이 시간 구간 [0, H]에서 실행을 끝낼 확률

본 논문에서는 태스크들이 RM(Rate Monotonic) 알고리듬에 따라서 스케줄링 된다고 설정한다. RM 알고리듬은 대표적인 고정 우선순위(fixed priority) 주기 스케줄링 기법 중 하나로서[1] 태스크의 우선순위가 데드라인에 반비례한다. 따라서 가장 짧은 데드라인을 가진 태스크의 우선순위가 가장 높고, 가장 긴 데드라인을 가진 태스크의 우선순위가 가장 낮다.

실시간 시스템은 단일 프로세서에서 m개의 주기적 멀티 태스크 $\mathbf{T}=\{T_1, T_2, \dots, T_m\}$을 실행한다($m \geq 1$). 태스크 T_i는 $T_i=(p_i, e_i)$로 표시한다. p_i는 T_i의 주기, e_i는 태스크의 실행시간이다. 본 논문에서는 각 태스크의 데드라인이 태스크 주기와 동일하다고 가정하고 $T_1, T_2, \dots, T_m$이 우선순위에 따라서 열거되었다고 정한다. 즉 $p_1 < p_2 < \cdots < p_m$이다. 또한 $\mathbf{T}_i=\{T_1, T_2, \dots, T_i\}$이라 정의한다.

모든 주기 $p_1, p_2, \dots, p_m$의 최소공배수를 H라고 하고 $\mathbf{T}$의 hyper-period라고 부른다. RM 알고리듬은 주기 스케줄링이므로 RM 알고리듬의 결과는 hyper-period H마다 동일한 패턴을 반복한다. 따라서 최초 스케줄링 시작 시각 0에서 H까지의 시간 구간만 고려하면 모든 구간에서 문제를 푼 것과 동일하다. 본 논문에서도 [0, H] 시간 구간만을 다룬다.

각 태스크는 매 주기마다 자신의 태스크 job(또는 instance)을 한 번씩 실행해야 한다. 태스크의 특정 주기에서 실행되는 job을 표기하기 위해서 매개 변수 $J_{i,j}$를 도입한다. $J_{i,j}$는 태스크 T_i의 j번째 주기에서 실행되는 job을 가리킨다. $v_i = H/p_i$라 하면 T_i는 시간 구간 [0, H]에서 $J_{i,1}, J_{i,2}, \dots, J_{i,vi}$ 등 총 v_i개의 job을 가진다.

2.2 등거리 체크포인트

본 논문에서 각 태스크 안에 삽입되는 체크포인트의 개수는 태스크 종류마다 다르고 한 태스크 안에 있는 체크포인트 사이의 거리는 일정하게(equidistant) 하는 일반적인 방법[2],[8]을 따른다.

태스크 T_i에 삽입하는 체크포인트 개수를 n_i라 하고 T_i의 체크포인트 구간을 Δ_i라 하자. 또 체크포인트 오버헤드를 t_{cp}라고 정의한다. 앞에서 기술했듯이 t_{cp}는 태스크의 상태를 저장하는 데 걸리는 시간과 고장 탐지 알고리듬을 수행하는 데 걸리는 시간의 합이다. 프로세서가 체크포인트 한 곳을 거치면 t_{cp}의 시간이 더 소요되어 실행시간이 늘어난다. 따라서 체크포인트의 등(等)간격 Δ_i와 개수 n_i 사이에는 다음과 같은 관계가 성립한다(e_i는 T_i의 실행시간).

$$\Delta_i = e_i/n_i + t_{cp} \tag{1}$$

어떤 체크포인트 구간에서 과도 고장이 발생하면 프로세서는 고장을 탐지한 체크포인트 직전에 위치한 체크포인트

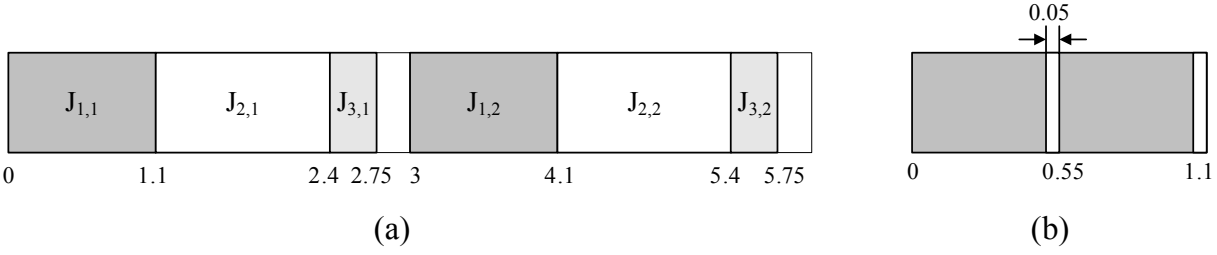

그림 1 $T_1=(3, 1.0)$, $T_2=(4, 1.2)$, $T_3=(5, 0.3)$, $n_1=2$, $n_2=2$, $n_3=1$, $t_{cp}=0.05$에 대한 RM 스케줄링.

Fig. 1 RM scheduling for $T_1=(3, 1.0)$, $T_2=(4, 1.2)$, $T_3=(5, 0.3)$, $n_1=2$, $n_2=2$, $n_3=1$, and $t_{cp}=0.05$.

로 되돌아가서 고장 난 구간을 재수행한다고 하였다. 이것은 재수행되는 횟수에 따라서 태스크의 실제 실행시간이 더 늘어남을 의미한다. 또 일반적으로 태스크의 각 주기마다 재수행되는 횟수가 다 다르므로 과도 고장 환경 하에서 태스크 job의 실행시간은 주기마다 다르게 나온다.

태스크 job의 실제 실행시간을 정량적으로 나타내기 위해서 각 태스크의 '재수행 벡터'를 도입한다. T_i의 재수행 벡터를 L_i라 하면 L_i는 $1\times v_i$ 크기를 가지는 다음과 같은 행벡터이다.

$$L_i = [l_{i,1}\ l_{i,2}\ \cdots\ l_{i,vi}] \tag{2}$$

여기서 $l_{i,j}$는 T_i의 j번째 주기의 job, 즉 $J_{i,j}$가 실행될 때 재수행된 체크포인트 구간 횟수를 말한다. $l_{i,j}$가 $J_{i,j}$에서 발생한 과도 고장 횟수와 반드시 일치하지는 않는다. 예를 들어 동일한 체크포인트 구간에서 고장이 여러 번 발생하여도 프로세서는 한 번만 rollback을 실행하므로 재수행되는 체크포인트 구간 횟수는 1이 된다.

체크포인트 오버헤드와 고장에 의한 재수행을 모두 고려하여 태스크 job의 실제 실행시간을 표현해보자. $e_{i,j}$를 $J_{i,j}$의 실제 실행시간이라고 정의한다. 체크포인트가 n_i개 있으므로 $e_{i,j}$는 우선 원 실행시간 e_i보다 $n_i t_{cp}$만큼 더 늘어나야 한다. 또한 $J_{i,j}$에서 재수행되는 체크포인트 구간 횟수는 식 (2)의 재수행 벡터 L_i로부터 $l_{i,j}$로 나온다. 종합하면 $e_{i,j}$는 아래와 같이 유도된다.

$$e_{i,j} = e_i + n_i t_{cp} + l_{i,j}\Delta_i \tag{3}$$

예제 1: m=3이고 매개 변수가 T_1=(3, 1.0), T_2=(4, 1.2), T_3=(5, 0.3)인 멀티 태스크가 RM 알고리듬에 의해서 스케줄링된다고 하자. 또 각 태스크에 오버헤드 t_{cp}=0.05인 체크포인트를 삽입하며, 체크포인트 개수를 n_1=2, n_2=2, n_3=1로 설정하자. 먼저 체크포인트 구간을 구하면 식 (1)에 의해서 다음과 같이 나온다.

Δ_1 = 1.0/2 + 0.05 = 0.55

Δ_2 = 1.2/2 + 0.05 = 0.65

Δ_3 = 0.3/1 + 0.05 = 0.35

편의상 재수행 벡터 L_1, L_2, L_3가 모두 0, 즉 과도 고장이 하나도 발생하지 않았다고 가정하고 이 멀티 태스크에 대한 RM 스케줄링을 수행하자. 그림 1(a)는 시간 구간 [0, 6]에 대한 스케줄링 결과를 나타낸다. 각 태스크의 첫 번째 job이 가지는 증가된 실행시간을 구하면 식 (3)으로부터

$e_{1,1}$ = 1.0 + 2×0.05 + 0×0.55 = 1.1

$e_{2,1}$ = 1.2 + 2×0.05 + 0×0.65 = 1.3

$e_{3,1}$ = 0.3 + 1×0.05 + 0×0.35 = 0.35

으로 계산된다. 재수행 벡터가 0이라고 가정하였으므로 두 번째 job의 실행시간도 위 값과 동일하다. 그림 1(b)는 $J_{1,1}$ 내부에 삽입된 체크포인트 위치를 상세하게 도시한 것이다.

3. 태스크 실행이 끝날 확률

3.1 과도 고장

체크포인트 연구 결과들은 발생 빈도가 Poisson 분포를 보이는 과도 고장에 대한 문제 설정을 주로 다루었다[2],[7],[9]. 본 논문에서도 실시간 시스템에서 일어나는 과도 고장의 발생 빈도가 발생율 $\lambda(\lambda>0)$인 Poisson 분포를 보인다고 가정한다. 현재의 확률이 과거의 사건과 무관하게 결정되는 Poisson 분포는 과도 고장 발생을 나타내기에 적절한 모델이다. 또한 실시간 시스템에서 발생하는 과도 고장을 Poisson 분포로 모델링한 사례도 많이 존재한다[2],[7].

길이 t인 임의 시간 구간에서 과도 고장이 n번 발생할 확률 $\alpha_n(\lambda,t)$는 다음과 같이 구해진다.

$$\alpha_n(\lambda,t) = \frac{(\lambda t)^n}{n!}e^{-\lambda t} \tag{4}$$

따라서 길이 t인 임의 시간 구간에서 과도 고장이 한 번도 발생하지 않을 확률은 $\alpha_0(\lambda,t) = e^{-\lambda t}$이며, 길이 t인 시간 구간에서 과도 고장이 한 번 이상 발생할 확률은 $1-e^{-\lambda t}$이다. T_i의 임의 체크포인트 구간 Δ_i에서 과도 고장이 한 번도 발생하지 않을 확률을 p_i, 한 번 이상 발생할 확률을 q_i라 하면 p_i와 q_i는 아래와 같다.

$$p_i = e^{-\lambda\Delta_i},\quad q_i = 1-e^{-\lambda\Delta_i} \tag{5}$$

3.2 태스크 실행이 끝날 확률 유도

앞에서 태스크 T_i에는 n_i개의 체크포인트가 삽입되고 T_i의 각 job마다 고장이 발생하여 L_i의 재수행이 일어난다고 하였다. 이 정보를 바탕으로 T_i의 각 주기에서 실행되는 job이 성공적으로 끝날 확률을 구한다. job $J_{i,j}$에서 재수행되는 구간 횟수는 식 (2)에서 정의했듯이 $l_{i,j}$이다. 따라서 $J_{i,j}$에서 실행되는 체크포인트 총 구간 수는 $n_i+l_{i,j}$이고, 이 중 n_i개의 구간에서 고장이 한 번도 발생하지 않아야 $J_{i,j}$가 실행 완료된다. $n_i+l_{i,j}$개의 구간 중 n_i개에 고장이 발생하지 않고 $l_{i,j}$개에 고장이 발생할 확률을 식 (5)의 매개 변수로 표현하면

$$\text{Prob.} = p_i^{n_i}q_i^{l_{ij}} \tag{6}$$

이다. 그런데 $J_{i,j}$의 실행이 끝날 확률은 식 (6)과 같은 값을 가지는 모든 경우의 확률을 더해야 구해진다. 이때 중요한 사실은 $J_{i,j}$의 마지막 체크포인트 구간에서는 고장이 발생하지 말아야 한다는 점이다. 만약 마지막 구간에서 고장이 일어나면 프로세서는 직전 체크포인트로 되돌아가므로 $J_{i,j}$의 실행이 완료되지 않았음을 의미하기 때문이다.

위 내용을 정리하면 $l_{i,j}$번의 재수행 구간이 있을 때 $J_{i,j}$의 실행이 주기 내에 끝날 경우는 총 $(n_i+l_{i,j}-1)$개의 체크포인트 구간 중 $l_{i,j}$개에 고장이 한 번 이상 발생하고 나머지 n_i-1개에서 고장이 한 번도 발생하지 않는 경우의 수와 같다 (맨 끝 구간에서는 항상 고장이 발생하지 않음). $l_{i,j}$번의 재수행 구간을 가진 $J_{i,j}$의 실행이 끝날 확률을 $\Psi_{ij}(l_{i,j})$라 하면 $\Psi_{i,j}(l_{i,j})$ 식 (6)과 앞 설명으로부터 다음과 같이 유도할 수 있다.

$$\psi_{i,j}(l_{i,j}) = {}_{n_i+l_{i,j}-1}C_{l_{i,j}}p_i^{n_i}q_i^{l_{i,j}} \tag{7}$$

위 식에서 ${}_xC_y$는 x개에서 y개를 선택하는 조합(combination)을 말한다.

[0, H] 시간 구간에서 재수행 벡터 L_i를 가진 태스크 T_i의 모든 job의 실행이 끝날 확률을 $\Psi_i(L_i)$라고 정의하자. $\Psi_i(L_i)$는 T_i의 각 job의 실행이 끝날 확률을 모두 곱한 값이므로

$$\psi_i(L_i) = \psi_{i,1}(l_{i,1})\psi_{i,2}(l_{i,2})\cdots\psi_{i,v_i}(l_{i,v_i}) \quad (8)$$
$$= \left({}_{n_i+l_{i,1}-1}C_{l_{i,1}} p_i^{n_i} q_i^{l_{i,1}}\right)\cdots\left({}_{n_i+l_{i,v_i}-1}C_{l_{i,v_i}} p_i^{n_i} q_i^{l_{i,v_i}}\right)$$

로 유도된다([0, H]안에 존재하는 T_i의 job은 v_i개이다).

4. Schedulability 검사

4.1 스케줄링 매개 변수

식 (8)의 $\Psi_i(L_i)$는 재수행 벡터 L_i이 주어질 때 태스크 T_i의 모든 job의 실행이 끝날 확률을 나타낸다. 하지만 $\Psi_i(L_i)$ 값만으로는 전체 태스크 **T**에 적용된 RM 스케줄링 자체의 성공 여부를 알지 못한다. 식 (3)에서 알 수 있듯이 재수행 구간과 체크포인트 오버헤드 때문에 job의 실제 실행시간은 늘어난다. 따라서 태스크에 대한 재수행 벡터가 주어질 때 RM 스케줄링이 유효한지를 검사한 후 RM 스케줄링이 실패하는 경우에 대한 확률값은 제외시켜야 올바른 결과를 얻을 수 있다.

RM 스케줄링이 유효한지를 검사하는 방법은 태스크의 각 주기에 남아 있는 여유시간(slack time)을 찾아서 모든 여유시간이 0 이상임을 확인하는 것이다. 그런데 RM 스케줄링의 특성상 어떤 태스크의 여유시간을 계산할 때에는 그 태스크보다 우선순위가 높거나 같은 태스크들만 고려하면 된다[1]. 즉 우선순위가 높은 태스크부터 schedulability 검사를 시작하며, T_i의 여유시간을 계산할 때에는 $\mathbf{T}_i$에 속한 태스크들이 T_i의 해당 주기에서 차지하는 실행시간만 찾으면 된다.

태스크 여유시간을 유도하기 위해서 스케줄링 이론에서 통용되는 몇 가지 개념을 도입한다. 상세한 설명은 [1]에서 찾아볼 수 있다.

▪ 절대 데드라인(absolute deadline) $d_{i,j}$: job $J_{i,j}$이 준수해야 할 데드라인을 스케줄링 시작 시점 0에서 경과한 시간으로 측정한 값이다. 앞에서 T_i의 데드라인은 주기 p_i와 동일하다고 하였다. 따라서 $J_{i,j}$의 절대 데드라인은

$$d_{i,j} = j \times p_i \quad (9)$$

이다.

▪ 유효 데드라인(effective deadline) $d^e_{i,j}$: $J_{i,j}$가 실제 만족시켜야 할 데드라인을 말한다. 고정 우선순위 스케줄링에서는 어떤 태스크 job의 절대 데드라인 근처에 그 태스크보다 우선순위가 높은 태스크의 job들이 실행되는 경우가 생긴다. 이러한 상황에서 $J_{i,j}$가 제대로 실행되기 위해서는 절대 데드라인에 존재하는 우선순위가 높은 job의 실행이 시작되기 전에 $J_{i,j}$의 실행을 모두 끝내야 한다.

▪ level-i busy 구간 (t_s, t_e]: 프로세서가 $\mathbf{T}_i$에 속하는 태스크의 job들만 실행하면서 "busy"한 구간이다. t_s 이전에 release된 $\mathbf{T}_i$의 job들은 모두 t_s 이전에 실행 완료되어야 한다. 즉 t_s가 $\mathbf{T}_i$ job의 실행 도중에 위치해서는 안 된다. 또 t_s 이후에 release된 $\mathbf{T}_i$의 job들은 모두 t_e 이전까지 실행 완료되어야 하며, (t_s, t_e] 구간 안에 프로세서가 idle하는 시간이 하나도 없어야 한다.

▪ 누적 여유시간 $\sigma_{i,j}(0)$: 스케줄링 시점 0부터 j번째 절대 데드라인, 즉 $j \times p_i$까지 측정한 태스크 T_i의 누적 여유시간이다. 정의에 의해서 $\sigma_{i,j}(0)$는 $J_{i,1}$, $J_{i,2}$,...,$J_{i,j}$가 가지는 여유시간을 모두 더한 값이다.

위 개념들 외에도 본 논문에서는 '상대적(relative) phase' 개념을 도입하여 여유시간을 측정하는 데 사용한다. 아래에서 정의되는 상대적 phase는 스케줄링 시작 시점 t=0에서부터 태스크 T_i의 맨 처음 job이 release되기까지의 시간 경과를 가리키는 (절대적) phase와[1] 약간 다른 개념이다.

▪ 상대적(relative) phase $\phi_i(t)$ ($\phi_i(t) \geq 0$): 어떤 시각 t 이후에 실행이 시작되는 태스크 T_i의 job 중 최소 release 시각과 t와의 시간차이로서 정의된다. 시간차가 T_i의 주기 p_i 이상이면 0으로 정한다. 일반적으로 태스크 종류마다 상대적 phase가 다를 수 있다. RM 스케줄링에서는 태스크 T_i의 주기 시작 시각 $j \times p_i$에서(j=0,1,2,...) T_i의 상대적 phase가 무조건 0이다. 즉 시각 $j \times p_i$에서 job $J_{i,(j+1)}$가 release된다. 또 스케줄링 시작 시점 t=0에서 모든 태스크의 첫 번째 job이 동시에 release되므로 상대적 phase는 모두 0이다. 다시 말하면 $\phi_i(0)=0$, $\forall i=1,...,m$이다.

누적 여유시간 $\sigma_{i,j}(0)$를 구하면 T_i의 각 주기마다 생기는 여유시간을 구할 수 있다. T_i의 j번째 주기에서 측정한 여유시간을 $s_{i,j}$라 하면 $s_{i,j}$와 $\sigma_{i,j}(0)$는 다음과 같은 관계를 가진다.

$$s_{i,j} = \sigma_{i,j}(0) - \sigma_{i,(j-1)}(0),\ s_{i,1} = \sigma_{i,1}(0) \quad (10)$$

$\sigma_{i,j}(0)$는 유효 데드라인 $d^e_{i,j}$를 이용하여 다음과 같이 표현된다[1].

$$\sigma_{i,j}(0) = d^e_{i,j} - \sum_{k=1}^{j}\sum_{h=1}^{r_{i,j}} e_{k,h},\ r_{i,j} = \left\lceil \frac{d^e_{i,j}}{p_k} \right\rceil \quad (11)$$

위 식에서 ⌈ x ⌉는 x보다 크거나 같은 최소 정수를 의미한다. 앞의 정의에서 설명했듯이 유효 데드라인 $d^e_{i,j}$의 값은 데드라인 부근에 T_i보다 우선순위가 높은 태스크의 job이 실행되고 있는지의 여부로 결정된다. level-(i-1) busy 구간을 이용하여 $d^e_{i,j}$의 식을 표현하면 아래와 같다[1],[10].

▪ $d^e_{i,j} = d_{i,j}$ $d_{i,j}$가 level-(i-1) busy 구간에 포함되지 않을 때

▪ $d^e_{i,j} = t_s$ $d_{i,j}$가 level-(i-1) busy 구간 (t_s, t_e]에 포함될 때

위 식들에서 알 수 있듯이 $\sigma_{i,j}(0)$은 유효 데드라인 $d^e_{i,j}$의 함수이고 $d^e_{i,j}$는 level-(i-1) busy 구간에 따라서 결정된다. 따라서 여유시간 $s_{i,j}$를 찾기 위해서는 모든 level-(i-1) busy 구간의 위치를 찾아야 한다.

4.2 level-i busy 구간 찾기

본 논문에서는 [1],[10]의 결과를 바탕으로 RM 알고리듬으로 스케줄링된 멀티 태스크가 가지는 level-i busy 구간을 찾는 방법을 제안한다. 기존 방법[1],[10]들은 임의 주기를 가진 태스크에는 적용할 수 없었지만 제안된 방법은 태스크의 주기에 제한을 두지 않는다.

앞에서 도입한 상대적 phase $\phi_i(t)$ 값에 따라서 level-i busy 구간을 다음 두 가지 종류로 분류할 수 있다.

▪ in-phase level-i busy 구간 (t_s, t_e]: t_s에서 $\mathbf{T}_i$에 속한 모든 태스크의 상대적 phase가 0으로 동일한 구간이다. 다시 말하면 $\phi_i(t_s)=0$, $\forall i=1,...,i$이다.

▪ arbitrary-phase level-i busy 구간 (t_s, t_e]: in-phase level-i busy 구간 정의를 만족시키지 않는 level-i busy 구

간들이다.

먼저 level-i busy 구간 (t_s, t_e]의 시작 시점 t_s를 알고 있을 때 이 구간의 길이를 구하는 방법을 기술한다. (t_s, t_e]의 길이를 B라 하면 구간의 종점은 $t_e=t_s+B$로 계산할 수 있다. (t_s, t_e]가 in-phase level-i 구간일 경우 B를 계산하는 식은 알려져 있다[1],[10]. 그런데 [1],[10]에 나와 있는 공식은 태스크의 실행시간이 일정한 경우에만 적용된다. 본 논문에서 다루는 멀티 태스크는 체크포인트 삽입 및 고장에 의한 재수행 때문에 주기마다 실제 실행시간이 서로 다를 수 있다.

주기마다 실행시간이 달라지는 멀티 태스킹에 적용할 수 있는 구간 길이 공식은 다음과 같다. 시작 시점이 t_s인 in-phase level-i busy 구간의 길이 B는 아래 등식을 만족하는 t 중 최소값이다.

$$t=\sum_{k=1}^{i}\sum_{h=a+1}^{b} e_{k,h},\ a=\left\lfloor \frac{t_s}{p_k} \right\rfloor,\ b=\left\lceil \frac{t}{p_k} \right\rceil \qquad (12)$$

위 식에서 $\lfloor x \rfloor$는 x보다 작거나 같은 최대 정수이다. 식 (12)는 구간 길이 B가 t_s에서 시작하는 level-i busy 구간에서 실행되어야 할 모든 job의 실행시간을 더한 값이라는 의미이다. 예를 들어 $t_s=0$이고 $p_k=2$라 하자.

$a=\lfloor 0/2 \rfloor=0$

이므로 식 (12)에 의해서 T_k의 첫 번째 job $J_{k,1}$의 실행시간 $e_{k,1}$이 포함된다. 또 B=3이 나왔다고 하면 식 (12)에서 b의 값은

$b=\lceil 3/2 \rceil=2$

이 된다. 즉 이 구간 동안 실행되는 T_k의 job은 $J_{k,1}$과 $J_{k,2}$ 두 개이며 실행시간 $e_{k,1}$과 $e_{k,2}$가 구간 안에 포함된다.

(t_s, t_e]가 arbitrary-phase level-i 구간이라면 t_s에서 $\mathbf{T}_i$에 속한 태스크 job들의 release 시간이 각각 다르므로 식 (12)가 바로 적용될 수 없다. 태스크 T_i job의 실행이 시작되어야 할 시점은 t_s에서 상대적 phase $\phi_i(t_s)$가 흐른 시각이다. 따라서 arbitrary-phase level-i 구간 (t_s, t_e]의 길이 B를 구하는 공식은 다음과 같이 유도된다. B는 아래 식을 만족시키는 t 중 최소값이다.

$$t=\sum_{k=1}^{i}\sum_{h=a+1}^{b} e_{k,h},\ a=\left\lfloor \frac{t_s}{p_k} \right\rfloor,\ b=\left\lceil \frac{t-\phi_k(t_s)}{p_k} \right\rceil \qquad (13)$$

level-i busy 구간의 시작 시점이 주어질 때 구간 길이를 구하는 방법을 식 (12)와 (13)과 같이 찾았다. 남은 일은 현재의 level-i busy 구간을 알고 있을 때 그 다음 level-i busy 구간의 시작 시점을 찾는 작업이다. 앞의 정의에서 level-i busy 구간 (t_s, t_e] 안에서 release된 $\mathbf{T}_i$의 모든 job들은 이 구간 내에서 실행 완료되어야 한다고 하였다. 따라서 (t_s, t_e] 직후에 위치하는 level-i busy 구간의 시작 시점은 t_e 이후에 생기는 $\mathbf{T}_i$ job의 release 시각 중 t_e와 가장 가까운 값이 된다.

현재까지 기술한 내용을 바탕으로 시간 범위 [0, H] 안에 존재하는 모든 level-i busy 구간을 찾는 알고리듬을 꾸미면 다음과 같다.

알고리듬 1: level-i busy 구간 찾기

1. RM 스케줄링 시작 시각 t=0에서 식 (12)를 참조하여 첫 번째 (in-phase) level-i busy 구간을 구하고 이 구간을 (t_0(=0), t_1]이라 하자.
2. 그 다음 level-i busy 구간을 (t_2, t_3]라 하자. t_2는 t_1 이후에 $\mathbf{T}_i$에 속한 태스크의 어떤 job이 release되는 시각 중 t_1과 가장 가까운 값이다.
3. 시각 t_2에서 $\mathbf{T}_i$의 모든 상대적 phase $\phi_i(t_2)$를 구한다. 각 태스크의 주기를 알고 있으므로 상대적 phase 값도 모두 구할 수 있다.
4. 3에서 구한 상대적 phase 값들을 식 (13)에 대입하여 (t_2, t_3] 구간의 길이 B를 구한다. 즉 두 번째 level-i busy 구간은 (t_2, t_3(=t_2+B)]이다.
5. 2~4 단계를 hyper-period H까지 수행한다. level-i busy 구간은 길이 H마다 동일한 패턴을 반복한다.

예제 2: 예제 1의 멀티 태스킹에서 level-2 busy 구간을 구한다. 첫 번째 구간 (t_0, t_1]은 스케줄링 시점 0에서 시작하므로 $t_0=0$이다. 또 식 (12)를 이용하여 구간의 길이를 구하면 B=2.4이다. 이 값이 맞는지를 검산하기 위해 식 (12)의 t를 2.4로 치환하면 (12) 우변의 a와 b는 아래와 같이 나온다.

i) k = 1: $a=\lfloor 0/3 \rfloor=0$, $b=\lceil 2.4/3 \rceil=1$

ii) k = 2: $a=\lfloor 0/4 \rfloor=0$, $b=\lceil 2.4/4 \rceil=1$

따라서 식 (12)의 우변은 $e_{1,1}+e_{2,1}=2.4$이므로 B=2.4가 구간 길이이고 첫 번째 level-2 busy 구간이 (0, 2.4]임을 확인할 수 있다. 알고리듬 1의 단계 2에 따라서 시각 2.4 이후에 release되는 $\mathbf{T}_2$의 job 중 2.4와 가장 가까운 것을 찾는다. 그림 1(a)을 보면 release 시각이 2.4와 가까운 job은 $J_{1,2}$이다 (level-2 busy 구간을 찾기 때문에 $J_{3,1}$은 제외한다). 따라서 두 번째 level-2 busy 구간 (t_2, t_3]의 시작점은 $t_2=3$이다. t_2에서 상대적인 phase는 $\phi_1(3)=0$, $\phi_2(3)=1$이다. 식 (13)을 이용하여 두 번째 level-2 busy 구간 길이를 계산하면 첫 번째 구간과 마찬가지로 2.4가 나온다. 따라서 두 번째 level-2 busy 구간은 (3, 5.4]이다.

4.3 Schedulability Indicator

알고리듬 1에 의해서 i=1,...,m-1에 대한 level-i busy 구간을 구한 다음에는 유효 데드라인 $d^e_{i,j}$를 구할 수 있고, 최종적으로 식 (10), (11)을 이용하여 태스크 각 주기마다 생기는 여유시간 $s_{i,j}$를 계산할 수 있다. 주어진 재수행 벡터 $L_1,...,L_m$에 대한 여유시간 $s_{i,j}$를 계산하면 RM 스케줄링이 성립하는지 여부를 결정할 수 있다.

시간 구간 [0, H]에서 태스크 T_i가 매 주기마다 가지는 여유시간은 $s_{i,1}$, $s_{i,2}$,...,$s_{i,vi}$이다. $s_{i,1}$, $s_{i,2}$,...,$s_{i,vi}$에 대한 함수 Γ_i를 다음과 같이 정의한다.

$$\Gamma_i = u(s_{i,1})u(s_{i,2})\cdots u(s_{i,vi}) \qquad (14)$$

위 식에서 u(x)는 단위 계단 함수(unit-step function)로서 x<0에서 u(x)=0, x≥0에서 u(x)=1이다. T_i가 매 주기마다 모두 0 이상의 여유시간을 가지면 $\Gamma_i=1$이 된다.

재수행 벡터 $L_1,...,L_m$을 가지는 전체 태스크 $\mathbf{T}$에 대한 RM 스케줄링 결과가 유효한지를 나타내는 함수를 'Schedulability Indicator'라고 하고 $\Gamma(L_1,...,L_m)$로서 표기하자. 식 (14)의 Γ_i를 이용하여 $\Gamma(L_1,...,L_m)$을 유도하면 다음과

같다.

$$\Gamma(L_1,L_2,...,L_m)=\prod_{i=1}^{m}\Gamma_i \tag{15}$$

$\Gamma(L_1,...,L_m)$=1이면 $\mathbf{T}$가 재수행 벡터 $L_1,...,L_m$을 가질 때 RM 스케줄링될 수 있다. $\Gamma(L_1,...,L_m)$=0이면 RM 스케줄링될 수 없으므로 태스크 실행이 끝날 확률을 계산할 때 재수행 벡터 $L_1,...,L_m$가 생기는 경우를 제외해야 한다.

5. 최적 체크포인트 구간

지금까지 유도한 단위 태스크에 대하여 실행이 끝날 확률과 Schedulability Indicator를 이용하여 전체 태스크의 실행이 끝날 확률을 구한다. 과도 고장이 발생하여 체크포인트 재수행 벡터 L_i를 가지는 태스크 T_i의 job이 hyper-period 구간 [0, H] 동안 모두 성공적으로 실행이 끝날 확률 $\Psi_i(L_i)$는 식 (8)에서 구하였다. 그렇다면 전체 태스크 $\mathbf{T}$에 대한 재수행 벡터가 $L_1,...,L_m$으로 주어질 때 모든 태스크의 job들이 [0, H] 동안 성공적으로 실행될 확률은 m개의 $\Psi_i(L_i)$를 곱하면 된다. 이 값을 나타내기 위해서 아래와 같은 변수 $\Psi(L_1,...,L_m)$을 정의하자.

$$\Psi(L_1,L_2,...,L_m)=\prod_{i=1}^{m}\Psi_i(L_i) \tag{16}$$

전체 태스크의 최종적인 실행이 끝날 확률은 가능한 모든 재수행 벡터 $L_1,...,L_m$에 대한 각각의 확률을 식 (16)로부터 구한 후 다시 이 확률들을 전부 더한 값으로 나온다. 따라서 우리는 우선 과도 고장이 하나도 발생하지 않은 상태인 $L_1=L_2=\cdots=L_m=0$부터 시작하여 주어진 멀티 태스킹에서 가능한 모든 재수행 벡터의 경우의 수를 찾아야 한다.

과도 고장이 한 번 발생하면 프로세서가 체크포인트 rollback을 한 번 실행하기 때문에 job의 실행시간은 Δ_i만큼 증가한다고 하였다(식 (3) 참조). 실행시간이 Δ_i만큼 늘어나면 반대로 이 job이 가지는 여유시간은 Δ_i만큼 줄어든다. 따라서 태스크의 한 주기 내에서 rollback으로 재수행될 수 있는 체크포인트 구간 횟수는 태스크 job이 보유한 최대 여유시간에서 체크포인트 구간을 나눈 값이다.

태스크 job이 가질 수 있는 최대 여유시간은 프로세서가 시간 구간 [0, H]에서 재수행을 한 번도 실시하지 않은 경우, 즉 과도 고장이 한 번도 발생하지 않은 경우이다. 식 (10)에서 job $J_{i,j}$의 여유시간을 $s_{i,j}$라 하였다. $J_{i,j}$의 최대 여유시간을 $s^1_{i,j}$라 표기한다. $s^1_{i,j}$는 식 (3)에서 재수행 횟수 $l_{i,j}$를 0으로 두고 $e_{i,j}$를 구한 후(j에 상관없이 $e_{i,j}$는 동일한 값) 다시 식 (10)과 (11)을 이용하여 계산할 수 있다.

태스크 T_i의 job이 실행되는 주기에서 프로세서가 구동할 수 있는 최대 재수행 횟수를 $l_{i,j}$라 하자. 앞서 기술한 바를 식으로 구현하여 $l_{i,j}$를 표시하면 아래와 같다.

$$\overline{l_{i,j}}=\left\lfloor \frac{s^1_{i,j}}{\Delta_i} \right\rfloor \tag{17}$$

태스크 T_i에 대한 재수행 벡터 L_i의 경계값이 $l_{i,j}$이므로 L_i는 $L_i=[0\ 0\ \cdots\ 0]$에서 $L_i=[l_{i,1}\ l_{i,2}\ \cdots\ l_{i,vi}]$까지 변화할 수 있다. 모든 $L_i=[l_{i,1}\ l_{i,2}\ \cdots\ l_{i,vi}]$의 경우에 대한 확률의 합을 나타내기 위해서 다음과 같은 합(合) 표기 기호를 도입한다.

$$\sum_{L_i=0}^{\overline{L_i}} := \sum_{l_{i,1}=0}^{\overline{l_{i,1}}}\sum_{l_{i,2}=0}^{\overline{l_{i,2}}}\cdots\sum_{l_{i,v_i}=0}^{\overline{l_{i,v_i}}} \tag{18}$$

위 식은 재수행 벡터 L_i가 $[0\ 0\ \cdots\ 0]$에서 $[l_{i,1}\ l_{i,2}\ \cdots\ l_{i,vi}]$까지 변할 때의 모든 경우의 수를 의미한다.

멀티 태스크 $\mathbf{T}$의 모든 job이 시간 구간 [0, H]에서 성공적으로 실행될 확률을 P($\mathbf{T}$,H)라고 하자. 식 (15), (16), (17)을 이용하여 P($\mathbf{T}$,H)를 찾을 수 있다. 이때 중요한 사항은 RM 스케줄링이 불가능한 재수행 벡터의 조합은 확률 계산에서 제외해야 한다는 사실이다. 이 과정은 재수행 벡터가 가지는 확률에 Schedulability Indicator $\Gamma(L_1,...,L_m)$을 곱함으로써 이루어진다. 아래는 P($\mathbf{T}$,H)의 최종 표현식이다.

$$P(\mathbf{T},H)=\sum_{L_1=0}^{\overline{L_1}}\sum_{L_2=0}^{\overline{L_2}}\cdots\sum_{L_m=0}^{\overline{L_m}}\left(\Gamma(L_1,...,L_m)\Psi(L_1,...,L_m)\right) \tag{19}$$

식 (8), (15)에서 유추할 수 있듯이 P($\mathbf{T}$,H)는 체크포인트 개수 n_i에 따라서 값이 바뀐다. 그러므로 가능한 모든 체크포인트 개수에 대한 P($\mathbf{T}$,H)의 최대값을 찾는 최적화 문제가 성립하며, P($\mathbf{T}$,H)를 최대로 하는 체크포인트 개수를 찾으면 식 (1)에 의해서 최적의 체크포인트 구간을 얻을 수 있다.

체크포인트 개수 n_i가 변할 수 있는 범위는 재수행 벡터 변수의 최대값 $l_{i,j}$와 마찬가지로 식 (3)을 이용하여 구할 수 있다. 즉 체크포인트를 가장 많이 삽입할 수 있는 경우는 과도 고장이 하나도 발생하지 않는 때이다. $s^0_{i,j}$를 과도 고장이 하나도 발생하지 않고 체크포인트도 없을 때 job $J_{i,j}$의 최초의 여유시간으로 정의하자. 즉 식(3)에서 $l_{i,j}$=0, n_i=0로 하고 식 (11)~(13)에서 $e_{i,j}=e_i$로 하면 job $J_{i,j}$에 대한 $s^0_{i,j}$를 구할 수 있다.

체크포인트 오버헤드가 t_{cp}이므로 $J_{i,j}$에 삽입할 수 있는 최대 체크포인트 수는 $\lfloor s^0_{i,j}/t_{cp} \rfloor$ 이다. 그런데 같은 태스크에 대하여 체크포인트 구간은 동일하므로 $J_{i,1},\ ...,\ J_{i,vi}$에 동일한 체크포인트가 삽입되어야한다. 따라서 n_i의 최대값 n_i는 다음과 같이 구할 수 있다.

$$\overline{n_i}=\min\left(\left\lfloor \frac{s^0_{i,1}}{t_{cp}} \right\rfloor,...,\left\lfloor \frac{s^0_{i,v_i}}{t_{cp}} \right\rfloor\right) \tag{20}$$

P($\mathbf{T}$,H)를 최대로 하는 체크포인트 개수를 $\{n^*_1,n^*_2,...,n^*_m\}$이라 하면 $\{n^*_1,n^*_2,...,n^*_m\}$은 다음 최적화 문제의 해이다. 즉 P($\mathbf{T}$,H)를 최대로 하는 체크포인트 개수 $\{n_1,n_2,...,n_m\}$를 구하는 것이다. 이때 최대로 삽입할 수 있는 체크포인터 개수는 제한되어 있으므로 체크포인터 개수는 아래 식과 같은 부등식을 만족하여야 한다.

$$\begin{aligned}\{n^*_1,n^*_2,...,n^*_m\} = \ & \arg\max\ P(\mathbf{T},H)\\ & 1 \le n_1 \le \overline{n_1}\\ & \cdots\\ & 1 \le n_m \le \overline{n_m}\end{aligned} \tag{21}$$

$\{n^*_1,n^*_2,...,n^*_m\}$을 찾은 다음에는 식 (1)을 이용하여 최적의 체크포인트 구간 Δ^*_i를 다음과 같이 구한다.

$$\Delta^*_i = e_i/n^*_i + t_{cp},\ i=1,...,m \quad (22)$$

6. 모의실험

이번 논문에서 제시한 체크포인트 구간 설정 방법을 실시간 시스템 스케줄링 문제에 직접 적용하여 그 성능을 검증한다.

먼저 두 개의 태스크를 가지는 실시간 시스템을 가정하자. $\mathbf{T}$ = {T_1, T_2}이며(m=2) T_1=(2, 1), T_2=(3, 0.5)로 설정한다. T_1과 T_2의 hyper-period는 H=lcd(2,3)=6이므로 hyper-period 구간 [0, 6]에서 T_1과 T_2의 job의 개수는 각각 v_1=3, v_2=2이다. 실시간 시스템에서 발생하는 과도 고장은 발생율 λ=0.1인 Poisson 분포를 따르고 체크포인트 오버헤드는 t_{cp}=0.05라고 한다. 체크포인트 개수의 최대 범위를 구하기 위해서 n_i=0, $l_{i,j}$=0일 때 유효 데드라인 $d^e_{i,j}$ 및 최초 여유시간 $s^0_{i,j}$를 구하면 다음과 같다.

$$d^e_{1,1}=2,\ d^e_{1,2}=4,\ d^e_{1,3}=6,\ d^e_{2,1}=2,\ d^e_{2,2}=6$$

$$s^0_{1,1}=1, s^0_{1,2}=1, s^0_{1,3}=1,\ s^0_{2,1}=0.5, s^0_{2,2}=1.5$$

식 (20)으로부터 n_i는 다음과 같이 구해진다.

$$\overline{n_1}=\min(\lfloor 1/0.05 \rfloor, \lfloor 1/0.05 \rfloor, \lfloor 1/0.05 \rfloor)=20$$

$$\overline{n_2}=\min(\lfloor 0.5/0.05 \rfloor, \lfloor 1.5/0.05 \rfloor)=10$$

그림 2는 n_1과 n_2 변화에 따른 P(**T**,H)를 나타낸다. 그림 2는 $1 \le n_1, n_2 \le 10$에 대한 확률 값을 표시한 것이다. 이외의 영역에서는 식 (15)의 Schedulability Indicator 값이 모두 0이 되므로 P(**T**,H)=0이다. P(**T**,H)의 최대값은 그림에서 표시한대로 {n^*_1, n^*_2}={2, 2}일 때이다. 식 (22)에 의해서 최적의 체크포인트 구간 Δ^*_i는 아래와 같이 나온다.

$$\Delta^*_1 = 1.0/2 + 0.05 = 0.55$$

$$\Delta^*_2 = 0.5/2 + 0.05 = 0.30$$

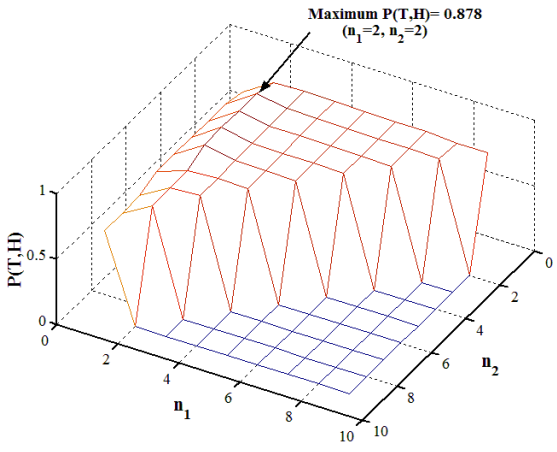

그림 2 T_1=(2, 1), T_2=(3, 0.5)를 가지는 실시간 시스템의 체크포인트 수에 따른 P(**T**,H)의 확률 변화.

Fig. 2 P(**T**,H) vs. checkpoint numbers tor the real-time system with T_1=(2, 1), T_2=(3, 0.5).

다음으로 세 개의 태스크를 가지는 실시간 시스템에 대한 체크포인트 구간 설정 문제를 생각하자. $\mathbf{T}$ = {T_1, T_2, T_3}이며(m=3) T_1=(1, 0.5), T_2=(2, 0.3), T_3=(3, 0.2)로 설정한다. hyper-period는 H=lcd(1,2,3)=6이고 [0, 6]에서 실행되는 각 태스크 job의 개수는 v_1=6, v_2=3, v_3=2이다. 과도 고장의 발생율 λ와 체크포인트 오버헤드는 앞의 예제와 동일하게 설정하였다. 먼저 n_i=0, $l_{i,j}$=0일 때 유효 데드라인 $d^e_{i,j}$ 및 최초 여유시간 $s^0_{i,j}$를 구하면 다음과 같다.

$$d^e_{1,1}=1,\ d^e_{1,2}=2,\ d^e_{1,3}=3, d^e_{1,4}=4, d^e_{1,5}=5, d^e_{1,6}=6$$

$$d^e_{2,1}=2,\ d^e_{2,2}=4, d^e_{2,3}=6,\ d^e_{3,1}=3,\ d^e_{3,2}=6$$

$$s^0_{1,1}=s^0_{1,2}=s^0_{1,3}=s^0_{1,4}=s^0_{1,5}=s^0_{1,6}=0.5$$

$$s^0_{2,1}=s^0_{2,2}=s^0_{2,3}=0.7,\ s^0_{3,1}=0.7,\ s^0_{3,2}=1$$

식 (20)로부터 체크포인트 개수 최대 범위 n_i를 구하면

$$\overline{n_1}=10,\ \overline{n_2}=14,\ \overline{n_3}=14$$

표 2는 체크포인트 개수 (n_1, n_2, n_3)를 변화시켜 가면서 얻은 P(**T**,H) 값을 보여준다. 표 2에서 나타내지 않은 (n_1, n_2, n_3)의 조합은 식 (15)의 Schedulability Indicator 값이 모두 0이 되는 경우이다. 표 2에서 P(**T**,H)의 최대값은 체크포인트 개수가 {n^*_1, n^*_2, n^*_2}={2, 1, 1}일 때 0.830으로 나왔다. 따라서 최적 체크포인트 구간 Δ^*_i는 다음과 같다.

$$\Delta^*_1 = 0.5/2 + 0.05 = 0.30$$

$$\Delta^*_2 = 0.3/1 + 0.05 = 0.35$$

$$\Delta^*_2 = 0.2/1 + 0.05 = 0.25$$

이상의 두 예제에서 볼 수 있듯이 본 논문에서 제안한 최적화 기법은 임의 주기를 가지는 멀티 태스크의 최종적인 실행이 끝날 확률을 최대화시키는 최적의 체크포인트 구간을 찾을 수 있다.

표 2 T_1=(1, 0.5), T_2=(2, 0.3), T_3=(3, 0.2)를 가지는 실시간 시스템의 체크포인트 수에 따른 P(**T**,H)의 확률 변화.

Table 2 P(**T**,H) vs. checkpoint numbers tor the real-time system with T_1=(1, 0.5), T_2=(2, 0.3), T_3=(3, 0.2).

(n_1,n_2,n_3)	P(**T**,H)	(n_1,n_2,n_3)	P(**T**,H)	(n_1,n_2,n_3)	P(**T**,H)
(1,1,1)	0.715	(1,3,4)	0.660	(2,2,2)	0.798
(1,1,2)	0.715	(1,3,5)	0.657	(2,2,3)	0.788
(1,1,3)	0.693	(1,4,1)	0.700	(2,2,4)	0.784
(1,1,4)	0.693	(1,4,2)	0.664	(2,3,1)	0.793
(1,1,5)	0.692	(1,4,3)	0.660	(2,3,2)	0.783
(1,1,6)	0.690	(1,4,4)	0.657	(2,3,3)	0.778
(1,1,7)	0.689	(1,5,1)	0.663	(2,4,1)	0.780
(1,1,8)	0.629	(1,5,2)	0.660	(2,4,2)	0.778
(1,1,9)	0.626	(1,5,3)	0.657	(2,5,1)	0.642
(1,2,1)	0.717	(1,6,1)	0.660	(3,1,1)	0.793
(1,2,2)	0.717	(1,6,2)	0.657	(3,1,2)	0.783
(1,2,3)	0.716	(1,7,1)	0.655	(3,1,3)	0.779
(1,2,4)	0.690	**(2,1,1)**	**0.830**	(3,2,1)	0.778
(1,2,5)	0.689	(2,1,2)	0.824	(3,2,2)	0.773
(1,2,6)	0.657	(2,1,3)	0.817	(3,3,1)	0.772
(1,2,7)	0.628	(2,1,4)	0.817	(4,1,1)	0.733
(1,3,1)	0.700	(2,1,5)	0.783	others	0
(1,3,2)	0.717	(2,1,6)	0.718		
(1,3,3)	0.664	(2,2,1)	0.802		

7. 결 론

실시간 시스템의 멀티 태스킹에서 삽입되는 체크포인트의 개수 및 구간 길이는 시스템의 내고장성 성능을 결정하는 중요한 변수이다. 본 논문에서는 멀티 태스크의 실행이 끝날 확률을 최대화시키는 최적의 체크포인트 구간을 찾는 기법을 제안하였다. 본 논문의 주요 기여도는 멀티 태스크들이 서로 아무런 연관성이 없는 임의 주기(arbitrary period)를 가지는 실시간 시스템에도 적용할 수 있는 방법을 제안했다는 점이다. 태스크 job의 유효 데드라인, level-i busy 구간 등의 개념을 이용하여 각 job이 가지는 여유시간 공식을 유도하였고, 여유시간으로 표현되는 Schedulability Indicator를 태스크의 확률에 곱해서 전체 태스크 실행이 끝날 확률식을 완성하였다. 또 몇 가지 예제들에 대한 모의실험 결과를 통해 본 논문에서 제시한 방법의 유용성을 입증하였다.

감사의 글

본 연구는 지식경제부・한국산업기술진흥원 지정 계명대학교 전자화자동차부품지역혁신센터의 지원에 의한 것입니다.

참 고 문 헌

[1] J. W. S. Liu, Real-Time Systems, New Jersey: Prentice Hall, 2000.

[2] J. W. Young, “A first order approximation to the optimal checkpoint intervals,” Communications of the ACM, vol. 17, no. 9, pp. 530-531, 1974.

[3] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio, “Distribution-free checkpoint placement algorithms based on min-max principle,” IEEE Transactions on Dependable and Secure Computing, vol. 3, no. 2, pp. 130-140, 2006.

[4] A. N. Tantawi and M. Ruschitzka, “Performance analysis of checkpointing strategies,” ACM Transactions on Computer Systems, vol. 2, no. 2, pp. 123-143, 1984.

[5] S. Punnekkat, A. Burns, and R. Davis, “Analysis of checkpointing for real-time systems,” International Journal of Time-Critical Computing Systems, vol. 20, pp. 83-102, 2001.

[6] 곽성우, 정용주, “RM 스케줄링된 실시간 태스크에서의 최적 체크포인터 구간 선정,” 전기학회논문지, 제56권 제6호, pp. 1122-1129, 2007.

[7] J. K. Kim and B. K. Kim, “Probabilistic schedulability analysis of harmonic multi-task systems with dual modular temporal redundancy,” Real-Time Systems, vol. 26, no. 2, pp. 199-222, 2004.

[8] A. Ranganathan and S. J. Upadhyaya, “Performance evaluation of rollback-recovery techniques in computer programs,” IEEE Transactions on Reliability, vol. 42, no. 2, pp. 220-226, 1993.

[9] Y. Ling, J. Mi, and X. Lin, “A variational calculus approach to optimal checkpoint placement,” IEEE Transactions on Computers, vol. 50, no. 7, pp. 699-708, 2001.

[10] T. S. Tia, “Utilizing slack time for aperiodic and sporadic requests scheduling in real-time systems,” Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

저 자 소 개

곽 성 우 (郭 成 祐)

1970년 3월 10일생. 1993년 2월 한국과학기술원 전기 및 전자공학과 졸업(학사). 1995년 2월 한국과학기술원 전기 및 전자공학과 졸업(석사). 2000년 한국과학기술원 전기 및 전자공학과 졸업(공박). 2000년～2002년 인공위성연구센터 선임연구원, 연구교수. 2003년～현재 계명대학교 전자공학과 부교수. 주관심분야: 실시간 시스템, 비동기 시스템 제어, 우주용 디지털 시스템 설계 등.

Tel : 053-580-5926
Fax : 053-580-5165
E-mail : ksw@kmu.ac.kr

양 정 민 (楊 正 敏)

1971년 3월 31일생. 1993년 2월 한국과학기술원 전기 및 전자공학과 졸업(학사). 1995년 2월 한국과학기술원 전기 및 전자공학과 졸업(석사). 1999년 2월 한국과학기술원 전기 및 전자공학과 졸업(공박). 1999년 3월～2001년 2월 한국전자통신연구원 컴퓨터・소프트웨어연구소 선임연구원. 2001년 3월～현재 대구가톨릭대학교 전자공학과 부교수. 주관심분야: 비동기 순차 머신 제어, 실시간 시스템 등.

Tel : 053-850-2736
Fax : 053-850-2704
E-mail : jmyang@cu.ac.kr