한국지능시스템학회 논문지 2011, Vol. 21, No. 1, pp. 120-127
DOI : 10.5391/JKIIS.2011.21.1.120

# 확률 최적화를 이용한 비선점형 Rate Monotonic 스케줄링의 체크포인트 구간 결정

# Determining Checkpoint Intervals of Non-Preemptive Rate Monotonic Scheduling Using Probabilistic Optimization

곽성우* · 양정민**+
Seong Woo Kwak and Jung-Min Yang

* 계명대학교 전자공학과
** 대구가톨릭대학교 전자공학과

## 요 약

체크포인트 기법은 실시간 시스템의 내고장성을 구현하는 대표적인 방법이다. 본 논문에서는 확률 최적화를 이용하여 체크포인트 구간을 결정하는 기법을 제시한다. 본 논문에서 다루는 실시간 시스템은 멀티 태스크(multi-task)들로 구성되며 Poisson 분포를 가지는 과도 고장이 발생한다. 또 멀티 태스크들은 비선점형 Rate Monotonic 알고리듬으로 스케줄링된다. 이번 연구에서는 멀티 태스크들의 수행 성공 확률을 체크포인트 삽입 개수로 표현하는 최적화 문제를 설정하고 이 확률 값을 최대로 만드는 체크포인트 개수와 구간 길이를 구한다. 제안된 확률 계산 과정은 체크포인트 재수행 횟수에 대한 비선점형 RM 알고리듬의 스케줄링 가능성을 판별하는 방법도 포함한다. 사례 연구를 통해서 제안된 기법의 적용가능성을 입증한다.

**키워드** : 체크포인트, Rate Monotonic 알고리듬, 비선점형, 과도 고장.

## Abstract

Checkpointing is one of common methods of realizing fault-tolerance for real-time systems. This paper presents a scheme to determine checkpoint intervals using probabilistic optimization. The considered real-time systems comprises multiple tasks in which transient faults can happen with a Poisson distribution. Also, multi-tasks are scheduled by the non-preemptive Rate Monotonic (RM) algorithm. In this paper, we present an optimization problem where the probability of task completion is described by checkpoint numbers. The solution to this problem is the optimal set of checkpoint numbers and intervals that maximize the probability. The probability computation includes schedulability test for the non-preemptive RM algorithm with respect to given numbers of checkpoint re-execution. A case study is given to show the applicability of the proposed scheme.

**Key Words** : Checkpoints, Rate Monotonic (RM) algorithm, Non-preemptiveness, Transient faults.

## 1. 서 론

태스크(task) 실행이 데드라인(deadline)까지 완료되어야 하는 실시간 시스템(real-time systems)의 성공적인 운용을 위해서는 시스템의 신뢰도(reliability)를 높이는 일이 필수적이다[1]. 또 외부 영향의 변화가 심한 환경에서 작동하는 시스템이 고(高)신뢰도를 가지려면 태스크 실행 시 발생하는 고장을 즉시 탐지하고 복구할 수 있는 내고장성(fault-tolerance)을 반드시 보유해야 한다. 체크포인트 기법(checkpointing)은 실시간 시스템의 내고장성을 구현하는 대표적인 방법 중의 하나이다[2-4]. 각 태스크 내부에 삽입된 체크포인트는 현재까지 수행된 태스크 모듈의 정보를 저장한다. 태스크 실행 중 고장이 발생하면 프로세서는 고장이 발생하기 직전에 위치한 체크포인트로 되돌아가(rollback) 체크포인트에 저장된 정보를 받아서 실행을 재개한다.

체크포인트 기법을 운용할 때 핵심적인 사항은 태스크에 몇 개의 체크포인트를 삽입하며, 각 체크포인트 사이의 구간을 어느 정도 벌려 놓는지를 결정하는 일이다. 체크포인트를 많이 삽입하면 발생하는 고장을 탐지할 확률이 높아지고 따라서 고장에 대한 재수행(re-execution) 부하(負荷)를 줄여서 고장 극복 속도를 높이는 장점을 가진다. 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 체크포인트의 개수와 구간을 알맞게 결정하는 일은 체크포인트 운용 기법에서 매우 중요하다[5].

이번 논문에서는 확률적 최적화 방법을 이용하여 체크포인트 구간 길이를 구하는 기법을 제시한다. 본 논문에서 다루는 실시간 시스템은 복수 개의 독립적인 멀티 태스크

접수일자 : 2010년 8월 6일
완료일자 : 2010년 10월 30일
+ 교신저자

(multi-task)들로 구성되며, 태스크 실행 시 프로세서에 Poisson 분포를 가지는 과도 고장(transient fault)이 발생한다. 과도 고장은 발생 즉시 고장 극복 메커니즘에 의해서 시스템이 오류 상태에서 정상적인 상태로 복귀 가능한 고장 종류이다[1]. 산업 현장에서 발생하는 실시간 시스템 고장들을 분석한 결과에 따르면 소자의 영구적 결함과 같은 하드웨어적인 원인보다는 외부의 전기적, 기계적 환경 변화 등으로 일어나는 과도 고장이 다수를 이룬다[6]. 따라서 본 논문에서도 고장 종류를 과도 고장으로 설정하였다. 과도 고장은 또한 중복 구조(redundancy)를 이용한 하드웨어적 방법보다는 본 논문의 주제인 체크포인트 삽입 등과 같은 소프트웨어적인 방법으로 더 잘 대처할 수 있다.

이번 연구에서 실시간 시스템의 멀티 태스크들은 대표적인 고정 우선순위(fixed priority) 스케줄링 기법인 Rate Monotonic(RM) 알고리듬[7, 8]으로 스케줄링된다고 가정한다. 태스크에서 고장이 발생하면 고장이 일어난 체크포인트 구간이 재수행되므로 태스크의 실행시간은 늘어나며, RM 알고리듬이 멀티 태스크들을 스케줄링하지 못할 경우도 발생한다. 본 논문에서는 체크포인트 재수행 횟수에 대한 RM 알고리듬의 스케줄링 가능성(schedulability)을 판별하는 방법을 개발한다. 또 스케줄링 가능성 판별법을 활용하여 모든 태스크들이 성공적으로 수행될 확률을 체크포인트 삽입 개수로 표현하는 최적화 문제를 설정한다. 이 확률 값을 최대로 만드는 체크포인트 개수와 구간 길이가 최적의 해가 된다.

저자의 선행 연구[9, 10]에서도 실시간 시스템에 삽입되는 체크포인트 구간을 선정하는 문제가 다루어졌다. 하지만 선행 연구들에서 고려한 RM 알고리듬은 모두 선점형(preemptive) 스케줄링[7], 즉 우선순위가 높은 태스크가 우선순위가 낮은 태스크의 실행을 막고 먼저 실행될 수 있는 스케줄링 원칙을 사용하였다. 본 논문에서는 태스크가 실행이 시작되면 우선순위에 상관없이 그 태스크의 실행을 절대 선점할 수 없는 비선점형 방식으로 RM 알고리듬을 구동한다. 비선점형 RM 알고리듬은 태스크 실행 순서가 우선순위를 뒤집는 현상(priority inversion)이 일어날 수 있기 때문에 태스크 수행 성공 확률 계산 과정과 스케줄링 가능성 판별법 등에서 선행 연구와 많은 차이점을 보인다. 본 논문의 사례 연구를 통해서 제안된 기법의 장점을 보이고 적용가능성을 분석한다.

## 2. 실시간 태스크 모델링

실시간 시스템은 단일 프로세서에서 m개의 주기적 멀티 태스크 $T=\{T_1, T_2, ..., T_m\}$을 실행한다($m\geq 2$). 태스크 $T_i$는 $T_i=(p_i, e_i)$로 정의하며 $p_i$는 $T_i$의 주기, $e_i$는 $T_i$의 원(original) 실행시간이다. 본 논문에서는 각 태스크의 데드라인이 태스크 주기와 동일하다고 가정하고 $T_1, T_2, ..., T_m$이 우선순위에 따라서 열거되었다고 정한다. 즉 $p_1 < p_2 < \cdots < p_m$이다. 또한 T에서 임의 태스크의 주기는 그 태스크보다 우선순위가 높은 태스크 주기의 정수배로 이루어진다는 'simply periodic' 조건[7]이 만족된다고 가정한다. 즉 $i<j$인 $T_i$와 $T_j$에 대해서 다음 식을 만족시키는 자연수 K(i,j)가 항상 존재한다.

$$p_j = K(i,j)p_i,\ K_{i,j}\in \mathbb{N} \tag{1}$$

또 $\Phi_i=\{T_1, T_2, ..., T_i\}$이라 정의하고($1\leq i\leq m-1$) 추후 사용하기로 한다.

식 (1)의 조건을 가지는 주기적 멀티 태스크에 RM 알고리듬을 적용하면 단위 태스크의 최대 주기, 즉 마지막 태스크의 주기 $p_m$마다 동일한 패턴이 반복된다. 따라서 최초 스케줄링 시작 시각 0에서 $p_m$까지의 구간 $[0, p_m]$만 고려하면 모든 구간에서 문제를 푼 것과 동일하다. 본 논문에서도 $[0, p_m]$ 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 $T_i$의 j번째 주기에서 실행되는 job을 가리킨다.

$$v_i = p_m/p_i \tag{2}$$

라 정의하면 $T_i$는 시간 구간 $[0, p_m]$에서 $J_{i,1}, J_{i,2}, ..., J_{i,vi}$ 등 총 $v_i$개의 job을 가진다.

본 논문에서 사용하는 체크포인트는 체크포인트 구간의 거리를 일정하게(equidistant) 하는 일반적인 방법으로[11] 태스크에 삽입된다. 하지만 논문에서 다루는 실시간 시스템이 멀티 태스크로 구성되기 때문에 태스크 종류마다 삽입되는 체크포인트 구간 길이와 개수는 각각 다르다는 사실을 유념해야 한다.

태스크 $T_i$에 삽입하는 체크포인트 개수를 $n_i$라 하고 $T_i$의 체크포인트 구간을 $\Delta_i$라 하자. 또 체크포인트 오버헤드를 $t_{cp}$라고 정의한다. $t_{cp}$는 태스크의 상태를 저장하는 데 걸리는 시간과 고장 탐지 알고리듬을 수행하는 데 걸리는 시간의 합이다[2, 3]. 프로세서가 체크포인트 한 곳을 거치면 $t_{cp}$의 시간이 더 소요되어 실행시간이 늘어난다. 따라서 $T_i$에 삽입되는 체크포인트의 등(等)간격 $\Delta_i$는 $T_i$의 원 실행시간 $e_i$를 $n_i$번 나눈 값에 오버헤드 $t_{cp}$를 더해서 구한다.

$$\Delta_i = e_i/n_i + t_{cp} \tag{3}$$

어떤 체크포인트 구간에서 과도 고장이 발생하면 체크포인트 구간이 재수행되는 횟수에 따라서 태스크의 실제 실행시간이 더 늘어난다. 태스크 job의 실제 실행시간을 정량적으로 나타내기 위해서 각 태스크의 '재수행 벡터'를 도입한다. $T_i$의 재수행 벡터를 $L_i$라 하면 $L_i$는 $1\times v_i$ 크기를 가지는 다음과 같은 행벡터이다.

$$L_i = [l_{i,1}\ l_{i,2}\ \cdots\ l_{i,vi}] \tag{4}$$

여기서 $l_{i,j}$는 $T_i$의 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 롤백(rollback)을 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 $n_i$개 있으므로 $e_{i,j}$는 우선 원 실행시간 $e_i$보다 $n_it_{cp}$만큼 더 늘어나야 한다. 또한 $J_{i,j}$에서 재수행되는 체크포인트 구간 횟수는 식 (4)의 재수행 벡터 $L_i$로부터 $l_{i,j}$로 나온다. 종합하면 $e_{i,j}$는 아래와 같이 유도된다.

$$e_{i,j} = e_i + n_it_{cp} + l_{i,j}\Delta_i \tag{5}$$

## 3. 스케줄링 가능성

### 3.1 비선점형 특성

확률 최적화 방법으로 체크포인트 구간을 찾는다는 것은 멀티 태스크들의 작업 수행이 데드라인 이내에서 성공적으로 끝날 확률을 최대로 하는 체크포인트 구간을 구한다는 뜻이다. 그런데 식 (5)에서 보였듯이 고장에 의해서 체크포인트 재수행이 일어나면 태스크 job의 실행시간이 늘어나기 때문에 태스크들이 비선점형 RM 알고리듬으로 스케줄링되지 못하는 경우가 생긴다. 따라서 재수행 벡터 $L_1$, $L_2$, ..., $L_m$이 주어질 때 비선점형 RM 알고리듬이 유효한지를 체크하는 '스케줄링 가능성(schedulability)' 검사를 반드시 거쳐야 한다.

RM 알고리듬의 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[7]. 즉 우선순위가 가장 높은 태스크 $T_1$부터 스케줄링 가능성 검사를 시작하며, 태스크 $T_i$의 job $J_{i,j}$의 여유시간을 계산할 때에는 $\Phi_i$에 속한 태스크들이 $J_{i,j}$의 해당 주기에서 차지하는 실행시간만 찾으면 된다.

그런데 본 논문에서 다루는 RM 알고리듬은 비선점형으로 구동하므로 $T_i$보다 우선순위가 낮은 태스크가 $T_i$의 job이 release되는 시각에 수행을 계속 하고 있을 때에는 $T_i$의 job이 우선순위가 더 높다 하더라도 프로세서를 선점할 수 없다.

위에서 말한 두 가지 성질을 반영하여 비선점형 RM 알고리듬의 스케줄링 가능성 검사 기법을 제안한다. 먼저 각 태스크 $T_i$에는 $n_i$개의 등거리 체크포인트가 삽입되었다고 가정하자. 재수행 벡터 $L_1$, $L_2$, ..., $L_m$이 주어질 때 $T_i$의 j번째 job $J_{i,j}$가 가지는 여유시간을 $s_{i,j}$라고 정의한다(j=1,...,$v_i$). 또 재수행 벡터가 모두 0인 경우, 즉 고장이 한 번도 발생하지 않은 초기 상태일 때 $J_{i,j}$가 가지는 여유시간을 $s_{i,j}(0)$이라고 하자.

$T_i$의 j번째 job $J_{i,j}$는 시각 $(j-1)p_i$에서 release되어 절대적 데드라인(absolute deadline) $t=jp_i$까지 수행 완료되어야 한다. RM 알고리듬의 원칙상 시간 구간 $[(j-1)p_i, jp_i]$내에서 release되는 $\Phi_i$의 모든 job, 즉 $T_i$보다 우선순위가 높거나 같은 태스크들의 job은 모두 그 구간이 끝날 때까지 수행 완료되어야 한다. 구간 $[(j-1)p_i, jp_i]$에서 $\Phi_i$의 job들이 실현되기 위해서 필요한 시간(time demand)을 $w_{i,j}(\Phi_i)$라 하면 $w_{i,j}(\Phi_i)$은 다음과 같이 구해진다.

$$w_{i,j}(\Phi_i) = e_{i,j} + \sum_{g=1}^{i-1} \sum_{h=\alpha}^{\beta} e_{g,h} \qquad (6)$$
$$\alpha = (j-1)K(g,i)+1$$
$$\beta = jK(g,i)$$

위 식에서 $e_{i,j}$는 job $J_{i,j}$ 자신의 실행시간이고 $e_{g,h}$는 $\Phi_i$에 속한 태스크 job 중 구간 $[(j-1)p_i, jp_i]$에서 release되는 것들의 실행시간이다. 식 (1)에서 $K(g,i)=p_i/p_g$로 정의되었다. $T_i$의 job $J_{i,j}$가 release되는 시각 $(j-1)p_i$를 $K(g,i)$로 나타내면 $(j-1)K(g,i)p_g$이므로 $T_g$는 $\alpha=(j-1)K(g,i)+1$번째 job $J_{g,\alpha}$를 release한다. 또 한 주기 $p_i$에서 release되는 $T_g$ job의 총 개수는 $p_i/p_g=K(g,i)$이므로 식 (6)에서 표현한 대로 $\beta=jK(g,i)$번째 job $J_{g,\beta}$까지 실행되어야 한다.

앞에서 말했듯이 비선점형 RM 알고리듬을 구동할 시에는 $w_{i,j}(\Phi_i)$ 이외에도 직전 주기에서 수행 완료되지 않고 현재 주기까지 침범하는 우선순위가 낮은 태스크 job들의 실행시간도 고려해야 한다. 구간 $[(j-1)p_i, jp_i]$에서 job $J_{i,j}$가 실행될 때 $T_i$보다 우선순위가 낮은 태스크 job들이 침범하는 부분을 $b_{i,j}$라고 하자. 비선점형 RM 알고리듬이 job $J_{i,j}$의 구간 $[(j-1)p_i, jp_i]$에서 구동될 때 $J_{i,j}$가 가지는 여유시간 $s_{i,j}$는 다음과 같이 구해진다.

$$s_{i,j} = p_i - w_{i,j}(\Phi_i) - b_{i,j} \qquad (7)$$

$J_{i,j}$의 구간 $[(j-1)p_i, jp_i]$에서 자기보다 우선순위가 높거나 같은 태스크들의 job이 모두 실행되고($w_{i,j}(\Phi_i)$) 또 비선점형 원칙에 의해서 직전 주기에서 넘어온 우선순위가 낮은 태스크 job들도 모두 실행된 후($b_{i,j}$) 남는 여유시간 $s_{i,j}$가 0 이상일 때 $J_{i,j}$는 스케줄링 가능하다.

비선점형 RM 알고리듬에서 우선순위 뒤집힘 현상이 발생하므로 $b_{i,j}$를 구하는 일은 간단하지 않다[7]. 더구나 본 논문에서 다루는 태스크들은 체크포인트 구간 재수행 때문에 실행시간이 각 주기마다 다르게 나오므로 재수행 벡터가 주어질 때 $b_{i,j}$를 구할 수 있는 일반적인 방법을 찾아야 한다.

$b_{i,j}$를 구하는 방법을 설명하기 위해서 m=3인 멀티 태스크 시스템 예를 든다. $p_2=2p_1$이고 $p_3=4p_1$라고 하자. 즉 $K(1,2)=2$, $K(1,3)=4$이다. 그림 1은 구간 $[0, 3p_1]$에서 비선점형 RM 알고리듬이 구동된 예이다. 먼저 그림 1(a)는 과도 고장이 아직 한 번도 발생하지 않은 초기 상태의 결과이다. 그림에서 $T_1$의 두번째 job $J_{1,2}$의 여유시간을 구해보자. $T_1$은 우선순위가 가장 높은 태스크이므로 식 (6)의 $w_{1,2}(\Phi_1)$은 자신의 실행시간 $e_{1,2}$만 포함한다. 또 그림 1(a)에서 우선순위가 낮은 태스크 $T_2$와 $T_3$의 job이 직전 주기에서 넘어와 구간 $[p_1, 2p_1]$을 차지하는 부분이 없으므로 $b_{1,2}=0$이다. 따라서 $J_{1,2}$의 여유시간 $s_{1,2}$는

$$s_{1,2} = p_1 - e_{1,2} \qquad (8)$$

이다.

그림 1(b)는 $J_{1,1}$의 수행 중 과도 고장의 발생으로 체크포인트 구간이 재수행되어 실행시간 $e_{1,1}$이 늘어난 경우이다. 즉 식 (5)에서 재수행 횟수가 $l_{1,1}\neq 0$이고 따라서 $J_{1,1}$에서 $l_{1,1}$번의 체크포인트 구간이 재수행되었다. $J_{1,1}$의 실행시간이 늘어나면 $J_{1,1}$ 직후에 오는 $J_{2,1}$과 $J_{3,1}$의 시작 시간도 뒤로 밀린다. 그런데 RM 알고리듬이 비선점형으로 동작하므로 그림 1(b)와 같이 $J_{3,1}$이 최고 우선순위 태스크 $T_1$의 다음 구간 시점인 $p_1$을 넘어서도 $J_{1,2}$가 실행될 수 없다. 만약 선점형 RM 알고리듬을 사용한다면 시각 $p_1$에서 $J_{1,2}$가 $J_{3,1}$을 제치고 프로세서를 선점하여 먼저 실행될 것이다. $J_{1,2}$는 $J_{3,1}$의 수행이 끝날 때까지 기다려야 하며 결과적으로 여유시간이 다음과 같이 줄어든다.

$$s_{1,2} = p_1 - e_{1,2} - b_{1,2} \qquad (9)$$

위 식에서 $b_{1,2}$는 그림 1(b)에서 표시된 대로 $J_{3,1}$이 구간 $[p_1, 2p_1]$에 침범한 부분의 길이이다.

그림 1(c)는 $J_{1,1}$뿐만 아니라 $J_{2,1}$에도 과도 고장이 발생하여 두 job의 실행시간이 모두 증가한 경우이다. 그림 1(b)와 비교하여 그림 1(c)가 가지는 차이점은 그림 1(c)의 경우에는 프로세서가 $J_{1,1}$과 $J_{2,1}$의 실행만으로 구간 $[0, p_1]$을 초과하여 다음 구간 $[p_1, 2p_1]$로 job의 일부분이 넘어온다는 점이다. 즉 그림 1(c)에서도 $b_{1,2}$가 0보다 크게 나오지만 $b_{1,2}$를 만드는 태스크가 그림 1(b)처럼 $T_3$이 아니라 $T_2$라는 사실이다. 따라서 $J_{1,2}$의 여유시간 공식은 식 (8)과 동일하지만

job의 실행 순서가 초기 상태 결과와 달라져서 $J_{1,2}$의 실행이 끝난 후 $J_{3,1}$이 실행된다.

$b_{i,j}$를 구하기 위해서 염두에 두어야 할 또 하나의 사실은 직전 주기에서 생긴 $b_{i,(j-1)}$이 현재 주기의 $b_{i,j}$에 영향을 미친다는 점이다. 그림 1(d)가 이러한 경우를 나타낸다. 그림 1(d)에서 $J_{1,1}$과 $J_{2,1}$의 실행시간이 증가하여 0보다 큰 $b_{1,2}$가 생긴 것은 그림 1(c)와 유사하다. 그런데 그림 1(c)와 달리 그림 1(d)에서는 $J_{1,2}$와 $J_{3,1}$에서도 과도 고장이 발생하여 실행시간 $e_{1,2}$와 $e_{3,1}$이 각각 증가하였다. 또 $J_{3,1}$의 수행이 시각 $2p_1$에서 완료되지 못하고 $T_1$의 다음 구간 $[2p_1, 3p_1]$에 침범하여 0보다 큰 $b_{1,3}$을 야기한다. 결국 $T_1$의 세번째 job $J_{1,3}$의 여유시간이 $b_{1,3}$만큼 줄어든다. 그런데 그림 1(d)를 관찰하면 $b_{1,3}$을 구하기 위해서는 $J_{3,1}$의 실행 시작 시각을 알아야 하고 또 $J_{3,1}$의 실행 시작 시각은 $b_{1,2}$와 연관된다는 사실을 알 수 있다.

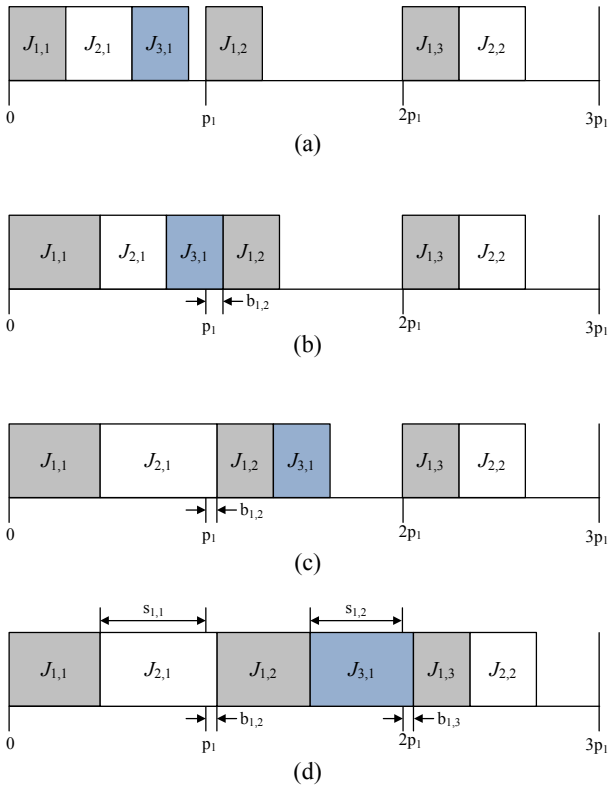


그림 1. 3-태스크 실시간 시스템에 대한 비선점형 RM 알고리듬 예 (a) 고장 미 발생, (b) $J_{1,1}$ 실행시간 증가, (c) $J_{1,1}$, $J_{2,1}$ 실행시간 증가, (d) $J_{1,1}$, $J_{2,1}$, $J_{1,2}$, $J_{3,1}$ 실행시간 증가
Fig. 1. Example of non-preemptive RM algorithm for a 3-task real-time system. (a) no transient faults, (b) increase in the execution time of $J_{1,1}$, (c) increase in the execution time of $J_{1,1}$ and $J_{2,1}$ and (d) increase in the execution time of $J_{1,1}$, $J_{2,1}$, $J_{1,2}$, and $J_{3,1}$.

### 3.2 여유시간 계산 알고리듬

재수행 벡터 값 $L_1, L_2, ..., L_m$이 주어질 때 모든 태스크 job에 대한 $b_{i,j}$와 $s_{i,j}$를 알려면 변화된 실행시간 $e_{i,j}$를 구한 후 비선점형 RM 알고리듬을 전체 구간 $[0, p_m]$에 다시 적용하면 된다. 그러나 이러한 방법은 구하는 데 시간이 많이 걸리는 소모적인 기법이므로 가능하면 비선점형 RM 알고리듬 계산을 최소화하면서 $b_{i,j}$를 구하는 것이 더 바람직하다.

본 논문에서는 고장이 한 번도 일어나지 않은 초기 상태에서 비선점형 RM 알고리듬을 한 번 적용한 결과만 이용하여 임의의 재수행 벡터 $L_1, L_2, ..., L_m$에 대한 $b_{i,j}$와 여유시간 $s_{i,j}$를 구하는 기법을 제안한다. 본 기법은 수학적 귀납법을 이용한다. 즉 $T_i$의 job $J_{i,(j-1)}$의 여유시간 $s_{i,(j-1)}$과 $b_{i,(j-1)}$을 알 때 다음 job $J_{i,j}$의 $s_{i,j}$와 $b_{i,j}$를 구한다. 제안하는 알고리듬을 단계별로 설명하면 다음과 같다.

1) 고장이 일어나지 않은 초기 상태에서 비선점형 RM 알고리듬을 적용하고 결과를 저장한다. job 실행 순서를 저장하는 변수 $\Omega(i,j)$를 다음과 같이 정의한다.

$$\Omega(i,j) = \{J_{k,l} |\ J_{i,j} \text{ 이전에 실행되는 } T_i\text{보다 우선순위가 낮은 모든 태스크 job}\} \quad (10)$$

그림 1(a)에서 $T_1$에 대한 $\Omega(i,j)$를 구하면

$$\Omega(1,1)=\varnothing,\ \Omega(1,2)=\{J_{2,1},J_{3,1}\},\ \Omega(1,3)=\{J_{2,1},J_{3,1}\} \quad (11)$$

이다. 정의에 의해서 $\Omega(i,1)\subseteq\Omega(i,2)\subseteq\cdots\subseteq\Omega(i,v_i)$인 관계가 성립한다.

2) 주어진 재수행 벡터 값 $L_1, L_2, ..., L_m$을 식 (5)에 적용하여 늘어난 실행시간 $e_{i,j}$를 구한다(i=1,...,m, j=1,...,$v_i$). $T_i$의 첫번째 job $J_{i,1}$은 시점 0에서 release되므로 $b_{i,1}$은 항상 0이다. 따라서 식 (7)에서 $b_{i,1}$=0으로 놓고 여유시간 $s_{i,1}$을 구한다. 예를 들어 $J_{1,1}$의 여유시간은 $s_{1,1}=p_1-e_{1,1}$이다.

3) $T_i$의 두번째 job $J_{i,2}$의 $b_{i,2}$는 직전 구간 $[0, p_i]$의 스케줄링 결과와 연관이 있다. $J_{i,1}$의 여유시간이 $s_{i,1}$이므로 구간 $[0, p_i]$에서 $T_i$보다 우선순위가 낮은 태스크의 job들은 이 여유시간 $s_{i,1}$ 동안 수행된다. 그런데 각 job들의 실행시간이 늘어났기 때문에 어떤 job은 구간 $[0, p_i]$에서 시작하여 (선점되지 않고) 현재 구간 $[p_i, 2p_i]$까지 넘어와서 끝날 수가 있다. 현재 구간까지 넘어오는 부분을 구하기 위해서 직전 구간 $[0, p_i]$에서 실행될 수 있는 우선순위 낮은 태스크 job들, 즉 앞에서 정의한 $\Omega(i,1)$에 속한 job들의 실행시간을 우선순위 내림차순으로 차례차례 더한다. 더한 값이 여유시간 $s_{i,1}$를 넘는 순간 $b_{i,2}$를 구한다. 이 과정을 식으로 나타내면 다음과 같다.

$$b_{i,2} = \sum_{J_{k,l}\in\Omega(i,2)} e_{k,l} - s_{i,1} \quad (12)$$

위 식에서 $J_{k,l}$은 $\Omega(i,2)$에 속한 job 중 $\sum_{J_{k,l}\in\Omega(i,2)} e_{k,l}$가 $s_{i,1}$보다 크면서 최소가 되는 원소들을 지칭한다. 이러한 조건을 만족시키는 $J_{k,l}$의 집합이 존재하지 않으면 $b_{i,2}$=0이다. 그림 1(b)에서 $b_{1,2}$를 예로 들어 설명한다. 앞에서 $\Omega(1,2) = \{J_{2,1},J_{3,1}\}$로 구했다. 그림 1(b)에서 알 수 있듯이 $J_{2,1}$의 실행시간 $e_{2,1}$은 $J_{1,1}$의 여유시간 $s_{1,1}$보다 작다. 그런데 $e_{2,1}$과 $J_{3,1}$의 실행시간 $e_{3,1}$을 합치면 분명히 $s_{1,1}$보다 크게 나온다. 따라서 식 (12)로부터 $b_{1,2}$를 구하면

$$b_{1,2} = (e_{2,1}+e_{3,1}) - s_{1,1} \quad (13)$$

이다. 한편 그림 1(c)에서는 $J_{2,1}$의 실행시간 $e_{2,1}$만으로도 여유시간 $s_{1,1}$을 초과한다. 역시 식 (12)를 적용하여 $b_{1,2}$를 구하면

$$b_{1,2} = e_{2,1} - s_{1,1} \quad (14)$$

로 유도된다(그림 1(d)도 식 (12)와 동일하게 나온다).

4) $b_{i,2}$를 구한 다음에는 모든 $\Omega(i,j)$(j>2)에서 $b_{i,2}$를 유도하

는 데 사용된 $T_i$보다 우선순위가 낮은 태스크 job들을 지운다. 식 (13)의 경우에는 $J_{2,1}$과 $J_{3,1}$을 지우며 식 (14)의 경우에는 $J_{2,1}$을 지운다. 어떤 job을 $\Omega(i,j)$에서 삭제한다는 의미는 그 job이 이전 주기에서 이미 수행 완료되었다는 사실을 기록하기 위함이다. 바꾸어 말하면 $\Omega(i,j)$에 남아 있는 job들은 $J_{i,(j-1)}$이 끝날 때까지 아직 실행이 되지 않은 것들이므로 $b_{i,j}$를 구할 때 계속 고려해야 한다. 그림 1(c)와 (d)를 예로 들면 식 (14)로부터 $J_{2,1}$이 완료되었다는 사실을 인지했으므로 $\Omega(1,3) = \{J_{3,1}\}$이 된다. 즉 $J_{1,2}$가 완료될 때까지 $J_{3,1}$이 아직 실행이 되지 않았다. 고장이 발생하지 않은 초기 상태인 그림 1(a)에서는 $T_1$의 첫번째 주기 동안 $J_{3,1}$이 수행 완료된다. 따라서 상기한 과정은 고장에 의한 재수행 과정 때문에 스케줄링 순서가 바뀐 결과를 기록하는 효과를 낸다.

5) 3)에서 기술한 유도 과정을 일반화하기 위해서 $T_i$의 job $J_{i,(j-1)}$의 여유시간 $s_{i,(j-1)}$과 $b_{i,(j-1)}$을 구했다고 가정하자. 또 job 집합 $\Omega(i,j)$도 4)의 방법대로 갱신되었다고 가정하자. 식 (12)를 다시 써서 $b_{i,j}$를 구하면 아래와 같다.

$$b_{i,j} = \sum_{J_{k,l} \in \Omega(i,j)} e_{k,l} - s_{i,(j-1)} \tag{15}$$

앞서와 마찬가지로 위 식에서 $J_{k,l}$은 $\Omega(i,j)$에서 우선순위 내림차순으로 차례차례 뽑아서 더해야 한다. $b_{i,j}$를 구한 후에는 (7)을 이용하여 여유시간 $s_{i,j}$를 구한다. 그림 1(d)를 다시 예로 들어 설명한다. 본 알고리듬을 $T_1$의 두번째 주기까지 적용하여 그림 1(d)에서 표시된 $b_{1,2}$와 $s_{1,2}$를 구했다고 하자. 앞에서 $\Omega(1,3) = \{J_{3,1}\}$라고 유도했으므로 식 (15)를 $T_1$의 세번째 주기에 적용하면 $b_{1,3}=e_{3,1}-s_{1,2}$가 나온다. 이 값은 그림 1(d)에 표시된 길이와 일치한다.

6) i=1,...,m, j=1,...,$v_i$에 대해 2)~5)의 과정을 반복한다(단계 1)은 한 번만 거치면 된다).

위에서 제안한 방법을 이용하여 임의의 재수행 벡터 값 $L_1$, $L_2$, ..., $L_m$을 가지고 비선점형 RM 알고리듬을 구동할 때 생기는 여유시간 $s_{i,j}$를 모든 태스크 job에 대해서 구할 수 있다.

## 4. 확률 최적화

### 4.1 단위 태스크 수행 성공 확률

본 논문에서는 실시간 시스템에서 일어나는 과도 고장의 발생 빈도가 발생율 $\lambda(\lambda>0)$인 Poisson 분포를 보인다고 설정한다. $T_i$의 임의 체크포인트 구간 $\Delta_i$에서 과도 고장이 한 번도 발생하지 않을 확률을 $p_i$, 적어도 한 번 이상 발생할 확률을 $q_i$라 하면 $p_i$와 $q_i$는 아래와 같다[2].

$$p_i = e^{-\lambda \Delta_i}, \quad q_i = 1 - e^{-\lambda \Delta_i} \tag{16}$$

앞에서 태스크 $T_i$에는 $n_i$개의 체크포인트가 삽입되고 $T_i$의 각 job마다 고장이 발생하여 $L_i=[l_{i,1}\ l_{i,2}\ \cdots\ l_{i,vi}]$의 재수행이 일어난다고 하였다. 이 정보를 바탕으로 $T_i$의 job $J_{i,j}$가 성공적으로 끝날 확률을 먼저 구한다. $J_{i,j}$에서 재수행되는 체크포인트 구간 횟수가 $l_{i,j}$이므로 $J_{i,j}$에서 실행되는 체크포인트 총 구간 수는 $n_i+l_{i,j}$이고, 이 중 $n_i$개의 구간에서 고장이 한 번도 발생하지 않아야 $J_{i,j}$가 실행 완료된다. $n_i+l_{i,j}$개의 구간 중 $n_i$개에 고장이 발생하지 않고 $l_{i,j}$개에 고장이 발생할 확률을 식 (16)의 매개 변수로 표현하면

$$\text{Prob.} = p_i^{n_i} q_i^{l_{ij}} \tag{17}$$

이다. 그런데 $J_{i,j}$의 실행이 성공할 확률은 식 (17)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 $n_i-1$개에서 고장이 한 번도 발생하지 않는 경우의 수와 같다. $l_{i,j}$번의 재수행 구간을 가진 $J_{i,j}$의 실행이 성공적으로 끝날 확률을 $\Psi_{ij}(l_{i,j})$라 하면 $\Psi_{i,j}(l_{i,j})$를 식 (17)과 앞 설명으로부터 다음과 같이 유도할 수 있다.

$$\psi_{i,j}(l_{i,j}) = {}_{n_i+l_{i,j}-1}C_{l_{i,j}} p_i^{n_i} q_i^{l_{i,j}} \tag{18}$$

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

총 시간 구간 $[0, p_m]$에서 재수행 벡터 $L_i$를 가지는 $T_i$의 $v_i$개의 job이 모두 성공적으로 실행될 확률을 $\Psi_i(L_i)$라고 정의하자. $\Psi_i(L_i)$는 $T_i$의 각 job의 성공 확률을 모두 곱한 값이므로

$$\begin{aligned}\psi_i(L_i) &= \psi_{i,1}(l_{i,1})\psi_{i,2}(l_{i,2}) \cdots \psi_{i,v_i}(l_{i,v_i}) \\ &= \left({}_{n_i+l_{i,1}-1}C_{l_{i,1}} p_i^{n_i} q_i^{l_{i,1}}\right) \cdots \left({}_{n_i+l_{i,v_i}-1}C_{l_{i,v_i}} p_i^{n_i} q_i^{l_{i,v_i}}\right)\end{aligned} \tag{19}$$

로 유도된다.

### 4.2 최적 체크포인트 구간

앞에서 유도한 단위 태스크에 대한 실행 성공 확률을 이용하여 전체 태스크 실행 성공 확률을 구한다. 과도 고장이 발생하여 체크포인트 재수행 벡터 $L_i$를 가지는 태스크 $T_i$의 job이 구간 $[0, p_m]$ 동안 모두 성공적으로 실행될 확률을 식 (19)에서 $\Psi_i(L_i)$로 유도하였다. 그렇다면 전체 태스크 T에 대한 재수행 벡터가 $L_1,L_2,...,L_m$으로 주어질 때 모든 태스크의 job들이 $[0, p_m]$ 동안 성공적으로 실행될 확률은 m개의 확률 변수 $\Psi_i(L_i)$를 모두 곱하면 된다. 이 값을 나타내기 위해서 아래와 같은 변수 $\Psi(L_1,...,L_m)$을 정의하자.

$$\Psi(L_1,L_2,...,L_m) = \prod_{i=1}^{m} \Psi_i(L_i) \tag{20}$$

전체 태스크의 최종적인 실행 성공 확률은 임의의 재수행 벡터 $L_1,...,L_m$에 대한 각각의 성공 확률을 위 식으로부터 구한 후 다시 이 확률들을 전부 더한 값으로 나온다. 따라서 우리는 우선 과도 고장이 하나도 발생하지 않은 초기 상태인 $L_1=L_2=\cdots=L_m=0$부터 시작하여 주어진 멀티 태스킹에서 가능한 모든 재수행 벡터의 경우의 수를 찾아야 한다.

앞에서 $s_{i,j}(0)$을 고장이 발생하지 않은 초기 상태에서 비선점형 RM 알고리듬을 적용했을 때 $J_{i,j}$가 가지는 여유시간이라고 정의하였다. $J_{i,j}$에서 체크포인트 구간이 k번 재수행되면 여유시간 $s_{i,j}(0)$에서 $k\Delta_i$만큼의 체크포인트 구간이 소비된다. 따라서 $J_{i,j}$에서 재수행될 수 있는 체크포인트 구간의 최대 횟수를 $l_{i,j}$라 하면 $l_{i,j}$는 아래와 같이 유도된다.

$$\overline{l_{i,j}} = \left\lfloor \frac{s_{i,j}(0)}{\Delta_i} \right\rfloor \tag{21}$$

위 식에서 ⌊x⌋는 x보다 작거나 같은 최대 정수이다.

태스크 $T_i$에 대한 재수행 벡터 $L_i$의 경계값이 $l_{i,j}$이므로 $L_i$는 $L_i=[0\ 0\ \cdots\ 0]$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.

$$\sum_{L_i=0}^{\overline{L_i}} := \sum_{l_{i,1}=0}^{\overline{l_{i,1}}} \sum_{l_{i,2}=0}^{\overline{l_{i,2}}} \cdots \sum_{l_{i,v_i}=0}^{\overline{l_{i,v_i}}} \tag{22}$$

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

각각의 재수행 벡터에 대한 확률 값을 더하기 전에 고려해야 할 사항은 주어진 재수행 벡터에 대해서 비선점형 RM 알고리듬이 유효한지를 확인하는 일이다. 앞 장에서 제안한 방법대로 각 태스크 job의 유효시간을 구하여 모든 유효시간 값이 0 이상임을 확인하면 현재 재수행 벡터가 가지는 확률 식 (20)을 전체 확률식에 더한다. 유효시간이 한 개라도 0보다 작다고 나오면 비선점형 RM 알고리듬이 실패한다는 의미이므로 현재 재수행 벡터의 태스크 성공 확률을 버려야 한다.

구간 $[0, p_m]$에서 태스크 $T_i$가 매 주기마다 가지는 여유시간은 $s_{i,1}, s_{i,2},\ldots,s_{i,vi}$이다. $s_{i,1}, s_{i,2},\ldots,s_{i,vi}$에 대한 함수 $\Gamma_i$를 다음과 같이 정의한다.

$$\Gamma_i = u(s_{i,1})u(s_{i,2})\cdots u(s_{i,vi}) \tag{23}$$

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

재수행 벡터 $L_1,L_2,\ldots,L_m$을 가지는 전체 태스크 T에 대한 비선점형 RM 알고리듬의 스케줄링 결과가 유효한지를 나타내는 함수를 $\Gamma(L_1,L_2,\ldots,L_m)$라고 표기하자. 위에서 정의한 $\Gamma_i$를 이용하여 $\Gamma(L_1,L_2,\ldots,L_m)$을 유도하면 다음과 같다.

$$\Gamma(L_1,L_2,\ldots,L_m)=\prod_{i=1}^{m}\Gamma_i \tag{24}$$

$\Gamma(L_1,\ldots,L_m)=1$이면 T가 재수행 벡터 $L_1,\ldots,L_m$을 가질 때 비선점형 RM 알고리듬으로 스케줄링될 수 있다. 반면에 $\Gamma(L_1,\ldots,L_m)=0$이면 RM 스케줄링될 수 없으므로 태스크 실행 성공 확률을 계산할 때 해당 재수행 벡터 $L_1,\ldots,L_m$이 만드는 경우를 제외해야 한다.

멀티 태스크 T의 모든 job이 시간 구간 $[0, p_m]$에서 성공적으로 실행될 확률을 P(T)라고 하자. 식 (20), (22), (24)를 이용하여 P(T)를 찾을 수 있다. 즉 임의의 재수행 벡터에 대해서 전체 태스크 T가 수행 성공할 확률을 식 (20)에서 구한 후 모든 재수행 벡터에 대한 이 확률 값을 더한다(식 (22)). 마지막으로 비선점형 RM 알고리듬이 불가능한 경우를 제외하기 위해서 각 확률 값에 식 (24)에서 정의한 함수 $\Gamma(L_1,L_2,\ldots,L_m)$를 곱함으로써 P(T)를 완성한다. 아래는 P(T)의 최종 표현식이다.

$$P(\mathbf{T})=\sum_{L_1=0}^{\overline{L_1}}\sum_{L_2=0}^{\overline{L_2}}\cdots\sum_{L_m=0}^{\overline{L_m}}\left(\Gamma(L_1,\ldots,L_m)\Psi(L_1,\ldots,L_m)\right) \tag{25}$$

위 확률은 체크포인트 구간 길이 $\Delta_i$, 즉 삽입되는 체크포인트 개수 $n_1,\ldots,n_m$에 대한 함수이다. 그러므로 가능한 모든 체크포인트 개수에 대한 P(T)의 최대값을 찾는 최적화 문제가 성립하며, P(T)를 최대로 하는 체크포인트 개수를 찾으면 식 (3)에 의해서 최적의 체크포인트 구간을 얻을 수 있다.

체크포인트 개수 $n_i$가 변할 수 있는 범위를 구하기 위해서 T에 체크포인트를 하나도 삽입하지 않은 상태를 생각하자. 구간 $[0, p_m]$에서 고장 없이 모든 태스크 job이 성공적으로 실행된 후 남는 시간을 $I_m$이라 하면 simply periodic 조건에 따라서 $I_m$은 아래와 같이 나온다.

$$I_m = p_m - (v_1e_1 + v_2e_2 + \cdots + v_me_m) \tag{26}$$

위 식에서 $v_i$는 식 (2)에서 정의한 대로 $v_i=p_m/p_i$이다. 태스크 $T_i$에 $n_i$개의 체크포인트를 삽입하면 구간 $[0, p_m]$에서 $T_i$의 job이 $v_i$개 있으므로 총 $v_in_i$개의 체크포인트가 삽입된다. 또 체크포인트 한 개당 오버헤드 $t_{cp}$를 필요로 하므로 $T_i$가 요구하는 오버헤드 총량은 $t_{cp}v_in_i$이다. 그런데 T에 속한 모든 태스크의 오버헤드 총량을 더한 값은 앞에서 구한 $I_m$보다 클 수가 없다. 따라서 체크포인트 개수 $n_1,\ldots,n_m$은 다음 조건을 만족시켜야 한다.

$$v_1n_1 + v_2n_2 + \cdots + v_mn_m \leq I_m/t_{cp} \tag{27}$$

위 식과 함께 체크포인트 개수 $n_i$가 자연수라는 조건도 당연히 만족되어야 한다.

P(T)를 최대로 하는 체크포인트 개수를 $\{n^*_1,n^*_2,\ldots,n^*_m\}$이라 하면 $\{n^*_1,n^*_2,\ldots,n^*_m\}$은 다음 최적화 문제의 해이다.

$$\begin{aligned}&\text{maximize } P(T)\\ &\text{subject to } v_1n_1 + v_2n_2 + \cdots + v_mn_m \leq I_m/t_{cp}\\ &\qquad n_1,\ n_2,\ \ldots,\ n_m \in \mathbb{N}\end{aligned} \tag{28}$$

$\{n^*_1,n^*_2,\ldots,n^*_m\}$을 찾은 다음에는 식 (3)을 이용하여 최적의 체크포인트 구간 $\Delta^*_i$를 다음과 같이 구한다.

$$\Delta^*_i = e_i/n^*_i + t_{cp},\ i=1,\ldots,m \tag{29}$$

## 5. 사례 연구

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

세 개의 태스크를 가지는 실시간 시스템을 가정하자. $T=\{T_1, T_2, T_3\}$이며(m=3) $T_1=(1, 0.35)$, $T_2=(2, 0.55)$, $T_3=(4, 0.5)$로 설정한다. 우선순위가 가장 낮은 $T_3$의 주기가 4이므로 구간 $[0, 4]$에서 $T_1, T_2, T_3$의 job의 개수는 각각 $v_1=4$, $v_2=2$, $v_3=1$이다. 실시간 시스템에서 발생하는 과도 고장은 발생율 $\lambda=0.1$인 Poisson 분포를 따르고 체크포인트 오버헤드는 $t_{cp}=0.05$라고 가정한다. 식 (26)에서

$$I_3 = 4 - (4\times0.35 + 2\times0.55 + 1\times0.5) = 1$$

이며, 식 (27)로부터 체크포인트 개수 $n_1, n_2, n_3$은 다음의 부등식을 만족하여야 한다.

$$4n_1 + 2n_2 + n_3 \leq 1/0.05 = 20$$

표 1은 $n_1, n_2, n_3$ 변화에 따른 P(T)를 나타낸다. 표 1에서는 위 부등식을 만족하는 체크포인터 개수 $(n_1,n_2,n_3)$에 대한 확률 값만 표시되었다. 이외의 영역에서는 식 (23)의 여유시간 존재 여부를 나타내는 $\Gamma_i$값 중 하나 이상이 0이 되므로 P(T)=0이다. P(T)의 최대값은 표 1에서 나타내었듯이 $\{n^*_1,n^*_2,n^*_3\}=\{1,1,2\}$일 때 0.9502이다. 식 (29)에 의해서 최적의 체크포인트 구간 $\Delta^*_i$는 아래와 같이 나온다.

$$\Delta^*_1 = 0.35/1 + 0.05 = 0.4$$

$$\Delta^*_2 = 0.55/1 + 0.05 = 0.6$$
$$\Delta^*_3 = 0.5/2 + 0.05 = 0.3$$

그림 2는 앞에서 구한 최적 체크포인터 개수 $\{n^*_1,n^*_2,n^*_3\}=\{1,1,2\}$를 각 태스크에 삽입하였을 때 얻어지는 비선점형 RM 스케줄링 결과를 나타낸다. 그림 2는 과도 고장이 한 번도 발생하지 않은 초기 상태이다. 과도 고장이 발생하여 체크포인트 롤백(rollback)이 일어난다면 실행시간이 늘어나고 또한 비선점형 특성에 따른 우선순위 뒤집힘 현상도 일어날 것이다. 또 각 태스크가 가지는 체크포인트 구간 길이가 서로 다르므로 롤백(rollback)에 의해서 늘어나는 시간도 달라진다.

이번 사례 연구에서는 3-태스크 실시간 시스템을 다루었지만 본 논문에서 제안한 기법은 n≥2인 임의의 n-태스크 시스템에 수정 없이 적용될 수 있다.

표 1. $T_1$=(1, 0.35), $T_2$=(2, 0.55), $T_3$=(4, 0.5)를 가지는 실시간 시스템의 체크포인트 수에 따른 P(T)의 확률 변화.
Table 1. P(T) vs. checkpoint numbers for the real-time system with $T_1$=(1,0.35), $T_2$=(2,0.55), and $T_3$=(4,0.5).

| $(n_1,n_2,n_3)$ | P(**T**) | $(n_1,n_2,n_3)$ | P(**T**) | $(n_1,n_2,n_3)$ | P(**T**) |
|---|---|---|---|---|---|
| (1,1,1) | 0.9491 | (1,2,9) | 0.6805 | (2,1,5) | 0.7385 |
| **(1,1,2)** | **0.9502** | (1,2,10) | 0.6771 | (2,1,6) | 0.7382 |
| (1,1,3) | 0.8390 | (1,2,11) | 0.6737 | (2,1,7) | 0.6805 |
| (1,1,4) | 0.8386 | (1,3,1) | 0.7497 | (2,1,8) | 0.6771 |
| (1,1,5) | 0.8383 | (1,3,2) | 0.7872 | (2,1,9) | 0.6737 |
| (1,1,6) | 0.8378 | (1,3,3) | 0.7869 | (2,2,1) | 0.7869 |
| (1,1,7) | 0.7557 | (1,3,4) | 0.7387 | (2,2,2) | 0.6907 |
| (1,1,8) | 0.7557 | (1,3,5) | 0.7385 | (2,2,3) | 0.6873 |
| (1,1,9) | 0.7556 | (1,3,6) | 0.6839 | (2,2,4) | 0.6839 |
| (1,1,10) | 0.7556 | (1,3,7) | 0.6805 | (2,2,5) | 0.6805 |
| (1,1,11) | 0.7516 | (1,3,8) | 0.6771 | (2,2,6) | 0.6771 |
| (1,1,12) | 0.7512 | (1,4,1) | 0.7458 | (2,3,1) | 0.6873 |
| (1,1,13) | 0.6737 | (1,4,2) | 0.6907 | (2,3,2) | 0.6839 |
| (1,1,14) | 0.6703 | (1,4,3) | 0.6873 | (2,3,3) | 0.6805 |
| (1,2,1) | 0.9099 | (1,4,4) | 0.6839 | (3,1,1) | 0.6873 |
| (1,2,2) | 0.8743 | (1,4,5) | 0.6805 | (3,1,2) | 0.6839 |
| (1,2,3) | 0.8736 | (1,5,1) | 0.6873 | (3,1,3) | 0.6805 |
| (1,2,4) | 0.7928 | (1,5,2) | 0.6839 | (3,1,4) | 0.6771 |
| (1,2,5) | 0.7482 | (2,1,1) | 0.7948 | (3,2,1) | 0.6805 |
| (1,2,6) | 0.7456 | (2,1,2) | 0.8320 | Others | 0 |
| (1,2,7) | 0.7454 | (2,1,3) | 0.8315 | | |
| (1,2,8) | 0.7451 | (2,1,4) | 0.8309 | | |

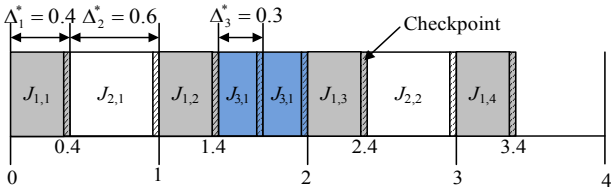


그림 2. $T_1$=(1, 0.35), $T_2$=(2, 0.55), $T_3$=(4, 0.5)인 태스크에 최적 체크포인트 개수 $\{n^*_1,n^*_2,n^*_3\}=\{1,1,2\}$를 삽입했을 때의 비선점형 RM 스케줄링.
Fig. 2. Non-preemptive RM scheduling for the tasks $T_1$=(1, 0.35), $T_2$=(2, 0.55), and $T_3$=(4, 0.5) with optimal checkpoint numbers $\{n^*_1,n^*_2,n^*_3\}=\{1,1,2\}$.

## 6. 결 론

이번 논문에서는 비선점형 RM 알고리듬으로 스케줄링되는 실시간 멀티 태스크에서 체크포인트 구간을 설정하는 새로운 방법을 제안하였다. 제안된 기법의 핵심은 체크포인트를 이용한 고장 진단 및 극복 과정에서 멀티 태스크가 성공적으로 수행 완료될 확률을 최대로 하도록 체크포인트 구간을 설정한다는 것이다. 또 주어진 재수행 벡터에 대해서 비선점형 RM 알고리듬이 스케줄링 가능한지 여부를 판별하는 방법을 제안하고 확률 최적화 계산에 활용하였다. 사례 연구의 모의실험 결과를 통해 본 논문에서 제시한 방법의 유용성을 입증하였다.

추후 연구에서는 본 논문에서 제안한 기법을 간헐적 고장(intermittent fault) 등 Poisson 분포를 가지지 않는 고장의 극복 과정에 적용하는 방법을 다룰 예정이다.

## 참 고 문 헌

[1] C. M. Krishina and K. G. Shin, *Real-Time Systems*, New York: McGraw-Hill, 1997.
[2] A. Ziv and J. Bruck, "Performance optimization of checkpointing schemes with task duplication," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1381-1386, 1997.
[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] S.-M. Ryu, "Reliability analysis for static checkpointing in embedded real-time systems," 2007 *International Symposium on Advanced Intelligent Systems*, pp. 965-970, 2007.
[5] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, pp. 303-312, 2006.
[6] H. Kim and K. G. Shin, "Design and analysis of an optimal instruction retry policy for TMR controller computers," *IEEE Transactions on Computers*, vol 45, no. 11, pp. 1217-1225, 1996.
[7] J. W. S. Liu, *Real-Time Systems*, New Jersey: Prentice Hall, 2000.
[8] 허보진, 손동철, 김창석, 이상용, "이동통신에서 무선채널할당과 CPU 작업 스케줄링 관계 연구," *한국지능시스템학회 논문지*, 제16권 제5호 pp. 575-580, 2006.
[9] 곽성우, 정용주, "RM 스케줄링된 실시간 태스크에서의 최적 체크포인터 구간 선정," *전기학회논문지*, 제56권 제6호, pp. 1122-1129, 2007.
[10] 곽성우, 정용주, "성능 함수를 고려한 실시간 제어 태스크에서의 최적 체크 포인터 구간 선정," *전기학회논문지*, 제57권 제5호 pp. 875-880, 2008.
[11] 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.

## 저 자 소 개

**곽성우(Seong-Woo Kwak)**

1993년 : 한국과학기술원 전기및전자공학과 (공학사)
1995년 : 한국과학기술원 전기및전자공학과 (공학석사)
2000년 : 한국과학기술원 전기및전자공학과 (공학박사)
2003년~현재 : 계명대학교 전자공학과 부교수

관심분야 : 실시간 시스템 고장 진단 및 극복, 우주용 디지털시스템 설계, 비동기 머신 교정 제어
E-mail : ksw@kmu.ac.kr

**양정민(Jung-Min Yang)**

1993년 : 한국과학기술원 전기및전자공학과 (공학사)
1995년 : 한국과학기술원 전기및전자공학과 (공학석사)
1999년 : 한국과학기술원 전기및전자공학과 (공학박사)
2001년~현재 : 대구가톨릭대학교 전자공학과 부교수

관심분야 : 비동기 머신 교정 제어, 실시간 시스템 고장 진단 및 극복, 보행로봇 걸음새 연구
E-mail : jmyang@cu.ac.kr