데드라인이 주기보다 긴 멀티 태스크를 가진 실시간 시스템을 위한 최적 체크포인트 배치

논 문
61-1-25

Optimal Checkpoint Placement for Real-Time Systems with Multi-Tasks Having Deadlines Longer Than Periods

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

Abstract - For a successful checkpointing strategy, we should place checkpoints so as to optimize fault-tolerance capability of real-time systems. This paper presents a novel scheme of checkpoint placement for real-time systems with periodic multi-tasks. Under the influence of transient faults, multi-tasks are scheduled by the Rate Monotonic (RM) algorithm. The optimal checkpoint intervals are derived to maximize the probability of task completion. In particular, this paper is concerned about the general case that the deadline of a task is longer than the period. Compared with the special condition that the deadline is equal to or less than the period, this general case causes a more complicate test procedure for schedulability of the RM algorithm with respect to a given set of checkpoint re-execution vectors. The probability of task completion is also derived in a more complex form. A case study is given to show the applicability of the proposed scheme.

Key Words : Checkpoints, Rate Monotonic (RM) algorithm, Transient faults, Probabilistic optimization.

1. 서 론

인공위성이나 기차, 비행기 같은 운송 시스템 등 높은 신뢰도가 요구되는 환경에서 동작하는 실시간 시스템(real-time systems)은 대부분 하드웨어나 소프트웨어적으로 내고장성(fault-tolerance)을 갖추고 있다. 하드웨어적인 내고장성은 중복구조(Double Modular Redundancy: DMR)나 삼중구조(Triple Modular Redundancy: TMR) 같이 똑같은 일을 하는 모듈(module)을 복수 개 이용하여 여유도(redundancy)를 구현한다[1]. 소프트웨어적인 내고장성은 실시간 시스템의 프로세서 커널(kernel)에 내고장성 프로그램을 장착하는 방법으로 최근 컴퓨터의 처리 성능 향상과 OS(Operating System) 기술의 발전으로 실제 실시간 시스템에 널리 적용되고 있다[2].

체크포인트 기법(checkpointing)은 소프트웨어적으로 실시간 시스템 내고장성을 구현하는 대표적인 방법 중의 하나이다[3],[4]. 수행되는 태스크(task) 모듈의 일정 간격마다 태스크 상태를 저장하고 현재 구간에서 발생하는 고장을 탐지하는 체크포인트를 삽입한다. 태스크 실행 중 고장이 발생하면 프로세서는 고장이 발생하기 직전에 위치한 체크포인트로 되돌아가(rollback) 체크포인트에 저장된 정보를 받아서 실행을 재개한다. 즉각적인 고장 탐지와 복구 능력이 갖춘 체크포인트 기법은 외부의 전기적, 기계적 환경 변화에 의해서 주로 발생하는 과도 고장(transient fault)을 대처하는 데 우수한 성능을 보인다[5].

본 논문의 목적은 최적화 기법을 이용하여 체크포인트 구간 길이를 구하는 일이다. 체크포인트 구간 길이는 인접한 체크포인트간의 간격을 말하며, 체크포인트 운용 기법의 성능을 결정하는 중요한 요소이다. 이번 논문에서 제안하는 최적화 목적 함수는 과도 고장이 발생할 때 멀티 태스크가 데드라인(deadline) 이내에 모든 작업(job)을 수행 완료할 확률 값이다. 체크포인트를 많이 삽입하면 과도 고장에 대해서 즉각적인 롤백(rollback)이 가능하기 때문에 내고장성이 커지고 태스크 성공 확률도 높아지지만 체크포인트를 거치면서 프로세서가 추가로 수행해야 할 부담(overhead)도 늘어나므로 성공 확률이 무작정 높아지지는 않는다[2],[6].

본 논문에서 실시간 시스템의 멀티 태스크는 일정한 주기마다 작업이 호출(release)되는 주기적 태스크들이라고 설정한다. 또 태스크들은 대표적인 고정 우선순위 (fixed priority) 스케줄링 기법인 선점형(preemptive) Rate Monotonic (RM) 알고리듬[7],[8]으로 스케줄링 된다고 가정한다. 태스크 한 주기 안에 삽입되는 체크포인트 개수에 대한 멀티 태스크 성공 확률을 폐형식(closed-form)으로 유도하고 체크포인트 개수가 만족시켜야 할 여러 가지 제한조건을 포함시켜 최적화 문제(optimization problem)를 구성한다. 이 확률 값을 최대로 만드는 체크포인트 개수와 구간 길이가 최적의 해가 된다.

저자가 수행한 체크포인트 배치에 대한 선행 연구로는 단일 태스크에 대해서 신뢰도(reliability)를 최적화하는 체크포인트 배치 기법과[9] 임의 주기를 가지는 멀티 태스크에 대한 체크포인트 구간 선정 방법 등이 있다[10]. 이번 연구는 [10]에서 설정한 임의 주기 멀티 태스킹 모델과 태스크 수행

* 정 회 원 : 계명대 전자공학과 부교수
† 교신저자, 정회원 : 대구가톨릭대 전자공학과 부교수
E-mail : jmyang@cu.ac.kr
접수일자 : 2011년 11월 11일
최종완료 : 2011년 12월 23일

성공 확률에 기반한 체크포인트 구간 결정 기법을 따른다. 하지만 선행 연구와 비교하여 이번 연구가 가지는 차별성은 실시간 시스템의 태스크가 주기보다 큰 데드라인을 가지도록 일반적으로 설정했다는 점이다. [9]와 [10]에서는 모든 태스크의 실행 주기가 태스크 데드라인과 같다고 가정하였다. 이 가정은 실시간 시스템이 성능 향상을 위해 데드라인보다 주기를 짧게 정하는 경우가 빈번하다는 사실과 맞지 않아 제한적이다[11]. 예를 들어 과학기술위성 1호의 동작 중 위성의 자세를 제어하는 태스크는 데드라인이 1~2초 정도로 정의되었지만 실제 태스크는 매 0.5초 주기로 실행되고 있다[12].

본 논문의 구성은 다음과 같다. 2장에서는 실시간 시스템을 도입하고 체크포인팅 기법을 모델링한다. 또 태스크 수행 성공 확률을 구하는 과정을 설명한다. 3장에서는 주어진 체크포인트 재수행 횟수에 대하여 RM 알고리듬이 성공할 가능성을 판별하는 기법을 제안한다. 4장에서는 3장에서 구한 스케줄링 가능성 함수를 기반으로 멀티 태스크의 수행 성공 확률을 구하고 최적화 문제를 풀어서 최적의 체크포인트 구간 길이를 구한다. 5장에서는 모의실험을 통해 제안한 기법의 효용성을 입증한다. 마지막으로 6장에서 본 논문의 결론을 내린다.

2. 문제 설정

2.1 실시간 시스템 모델링

실시간 시스템은 단일 프로세서에서 m개의 주기적 멀티 태스크 $\mathbf{T}=\{T_1, T_2, ..., T_m\}$을 실행한다($m\geq 2$). 태스크 T_i는 $T_i=(p_i, e_i, d_i)$로 정의한다. p_i는 T_i의 주기, e_i는 T_i의 원(original) 실행시간, d_i는 T_i의 상대적 데드라인(relative deadline)이다. 본 논문에서는 태스크 $T_1, T_2, ..., T_m$이 우선순위에 따라서 열거되었다고 정한다. 즉 $p_1 < p_2 < \cdots < p_m$이며 $d_1 < d_2 < \cdots < d_m$이다. 또한 각 태스크는 다음과 같은 세 가지 조건을 만족한다고 가정한다.

가정 1: $\mathbf{T}$에서 임의 태스크의 주기는 그 태스크보다 우선순위가 높은 태스크 주기의 정수배로 이루어진다는 'simply periodic' 조건[7]이 만족된다. 즉 $i<j$인 T_i와 T_j에 대해서 다음 식을 만족시키는 자연수 $k(i,j)$가 항상 존재한다.

$$p_j = k(i,j)p_i,\ k(i,j)\in \mathbb{N}$$

서론에서 밝혔듯이 논문에서 다루는 태스크는 주기보다 상대적 데드라인이 큰 종류이므로 임의의 T_i에 대해서 $p_i<d_i$이다. 그런데 통상 상대적 데드라인은 주기의 두 배를 넘지 않는다.

가정 2: 각 태스크의 p_i와 d_i는 아래 식을 만족시킨다.

$$p_i < d_i < 2p_i,\ \forall i=1,...,m$$

$\Phi_i=\{T_1, T_2, ..., T_i\}$인 집합 Φ_i를 정의하고($1\leq i\leq m-1$) 추후 기술에서 사용하기로 한다.

가정 3: 실시간 시스템에서 일어나는 과도 고장의 발생 빈도가 발생율 $\lambda(\lambda>0)$인 Poisson 분포를 보인다고 설정한다. T_i의 임의 체크포인트 구간 Δ_i에서 과도 고장이 한 번도 발생하지 않을 확률을 p_i, 적어도 한 번 이상 발생할 확률을 q_i라 하면 p_i와 q_i는 아래와 같다[3].

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

Simply periodic 조건을 가지는 주기적 멀티 태스크에 RM 알고리듬을 적용하면 일정한 구간마다 동일한 스케줄링 패턴이 반복된다. $\mathbf{T}$가 simply periodic 조건을 만족시키므로 패턴이 반복되는 구간 길이는 마지막 태스크 T_m의 한 주기 p_m임을 알 수 있다. T_m보다 우선순위가 높은 태스크는 길이 p_m의 구간 안에서 정수배의 주기를 반복한다. 따라서 T_m의 작업(job) 하나가 호출(release)된 시점에서 그 작업이 끝나야 하는 종점까지의 구간만 고려하면 모든 시간 구간에서 문제를 푼 것과 동일한 효과를 낸다.

각 태스크는 매 주기마다 자신의 태스크 작업 하나를 호출(release)해야 하며 상대적 데드라인 동안 실행 완료해야 한다. 태스크의 특정 주기에서 실행되는 작업을 표기하기 위해서 매개 변수 $J_{i,j}$를 도입한다. $J_{i,j}$는 태스크 T_i의 j번째 주기에서 실행되는 작업을 가리킨다.

스케줄링이 최초 시작되는 시각을 t=0이라 하면 T_m의 (a+1)번째 주기가 시작되는 시각은 $t=ap_m$이며(a는 $a\geq 0$인 정수) 이때 작업 $J_{m,(a+1)}$이 호출된다. $J_{m,(a+1)}$이 가지는 구간 $[ap_m, (a+1)p_m]$에서 호출되는 태스크 T_i의 작업의 개수를 v_i라 하면 v_i는 simply periodic 조건에 의해서 다음과 같이 구해진다(i=1,2,...,m).

$$v_i = \frac{p_m}{p_i}$$

$J_{m,(a+1)}$이 호출되는 시점은 $t=ap_m=av_ip_i$이므로 $J_{m,(a+1)}$이 호출될 때 T_i의 호출 $J_{i,(av_ip_i+1)}$이 동시에 호출된다. 따라서 구간 $[ap_m, (a+1)p_m]$에서 호출되는 T_i의 작업은

$$J_{i,(av_ip_i+1)}, J_{i,(av_ip_i+2)}, ..., J_{i,(av_ip_i+v_i)}$$

이다. 본 논문에서는 RM 스케줄링과 태스크 성공 확률 계산을 구간 $[ap_m, (a+1)p_m]$에서 호출되는 작업으로 한정한다. 그런데 작업 $J_{i,(av_ip_i+j)}$의 색인(index) 표기가 너무 길기 때문에 본 논문에서는 앞으로 아래첨자-색인 av_ip_i를 빼기로 한다. 즉 $J_{i,(av_ip_i+j)}$ 대신에 간단하게 $J_{i,j}$로 표기하여(j=1,...,v_i) $[ap_m, (a+1)p_m]$에서 j번째로 호출되는 T_i의 작업을 나타내기로 한다. 또 구간 $[ap_m, (a+1)p_m]$도 기술의 편의성을 위해서 시점과 종점에서 각각 ap_m을 빼서 $[0, p_m]$이라고 간주한다.

태스크의 상대적 데드라인이 주기보다 크기 때문에 구간 $[ap_m, (a+1)p_m]$에서 호출되는 작업으로 한정해도 실제 확률 계산을 할 때 고려해야 하는 구간은 $[0, p_m]$를 넘어선다는 사실을 유념해야 한다. 마지막 태스크 T_m의 작업 $T_{m,1}$을 예를 들면 이 작업은 시각 0에서 호출되지만 $d_m>p_m$이므로 구간 $[0, d_m]$이내에만 완료되면 된다. 따라서 확률 계산 시 체크해야 하는 구간은 $[0, p_m]$이 아닌 $[0, d_m]$이다.

확률 계산을 하는 구간이 t=0에서 시작한다고 간주했기 때문에 각 태스크 작업의 '절대적 데드라인(absolute deadline)'을 구할 수 있다. $J_{i,j}$의 절대적 데드라인을 $d_{i,j}$라 하면 $d_{i,j}$는 $J_{i,j}$가 수행 완료되어야 하는 시각을 t=0에서부터 측정한 값이다. $J_{i,j}$가 $t=(j-1)p_i$에서 호출되므로 $d_{i,j}$는

$$d_{i,j} = (j-1)p_i + d_i$$

이다.

2.2 체크포인트 삽입과 태스크 성공 확률

체크포인트는 각 태스크에 일정한 간격으로 삽입된다. 하지만 본 논문에서 다루는 실시간 시스템이 멀티 태스크로 구성되기 때문에 태스크 종류마다 삽입되는 체크포인트 구

간 길이와 개수는 각각 다르다. T_i에 삽입하는 체크포인트 개수를 n_i라 하고 T_i의 체크포인트 구간을 Δ_i라 하자. 또 체크포인트 부담(overhead)을 t_{cp}라고 정의한다. t_{cp}는 태스크의 상태를 저장하는 데 걸리는 시간과 고장 탐지 알고리듬을 수행하는 데 걸리는 시간의 합이다[2]. 프로세서가 체크포인트 한 곳을 거치면 t_{cp}의 시간이 더 소요되어 실행시간이 늘어난다. 따라서 T_i에 삽입되는 체크포인트의 등(等)간격 Δ_i는 T_i의 원 실행시간 e_i를 n_i번 나눈 값에 체크포인트 부담 t_{cp}를 더해서 구한다.

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

어떤 체크포인트 구간에서 과도 고장이 발생하면 체크포인트 구간이 재수행되는 횟수에 따라서 태스크의 실제 실행시간이 더 늘어난다. 태스크 작업(job)의 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.

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

여기서 $l_{i,j}\geq 0$인 $l_{i,j}$는 T_i의 j번째 주기의 작업, 즉 $J_{i,j}$가 실행될 때 재수행되는 체크포인트 구간 횟수를 말한다.

체크포인트 부담과 고장에 의한 재수행을 모두 고려하여 태스크 작업의 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_it_{cp}만큼 더 늘어나야 한다. 또한 $J_{i,j}$에서 재수행되는 체크포인트 구간 횟수는 재수행 벡터 L_i로부터 $l_{i,j}$로 나온다. 종합하면 $e_{i,j}$는 아래와 같이 유도된다.

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

논문의 핵심 사항인 태스크 수행 성공 확률의 개념을 본 절에서 먼저 설명한다. 우선 전체 태스크 **T**에 과도 고장이 발생하지 않고 또 체크포인트가 하나도 삽입되지 않았다면 **T**는 RM 알고리듬으로 당연히 스케줄링 가능해야 한다. 그렇지 않다면 문제 자체가 성립하지 않는다. 각 태스크 T_i에 n_i개의 체크포인트를 삽입하면 위에서 기술했듯이 실행시간이 n_it_{cp}만큼 무조건 늘어난다. 고장에 의해서 체크포인트 구간이 재수행된다면 실행시간은 더욱 늘어나고 따라서 태스크의 작업이 가지는 여유시간(slack time)이 줄어들어 RM 알고리듬으로 **T**를 스케줄링하지 못하는 경우도 생긴다. 그런데 T_i에 대한 재수행 벡터 L_i는 고장에 의해서만 결정되므로 우리가 그 값을 정확하게 예측할 수 없고 오직 확률적으로 발생을 산출해야 한다. 확률적으로 일어나는 재수행 벡터 $L_1,...,L_m$이 주어지면 각 태스크 작업의 실제 실행시간 $e_{i,j}$가 유도되며 따라서 우리는 전체 태스크 **T**가 RM 알고리듬으로 스케줄링될 수 있는지를 판단할 수 있다. 주어진 재수행 벡터를 가지고 RM 스케줄링이 가능하면 현재 재수행 벡터가 발생하는 확률을 전체 태스크 수행 성공 확률에 더하며, 반면에 스케줄링이 불가능하면 현재 재수행 벡터의 발생 확률 값을 버린다.

결론적으로 체크포인트 구간 길이, 즉 삽입되는 체크포인트 개수 $n_1,n_2...,n_m$이 정해지면 체크포인팅 운용을 하는 실시간 시스템이 모든 태스크 **T**를 완료할 성공 확률이 결정된다. 우리는 이 성공 확률을 목적 함수로 하는 최적화 문제를 설정하여 최적의 체크포인트 구간 길이를 구할 것이다.

3. 스케줄링 가능성 판별

앞에서 밝혔듯이 태스크 수행 성공 확률을 찾기 위해서는 재수행 벡터에 대한 RM 알고리듬의 스케줄링 가능성(schedulability)을 판별하는 일이 필수적이다. 멀티 태스크가 RM 알고리듬으로 스케줄링 가능하기 위한 필요충분조건은 모든 태스크 작업이 0 이상의 여유시간을 가지는 것이다.

재수행 벡터 $L_1,...,L_m$을 가진 실시간 시스템에 RM 알고리듬을 적용했을 때 구간 $[0, p_m]$에서 호출(release)된 T_i의 j번째 작업 $J_{i,j}$가 가지는 여유시간을 $s_{i,j}$라 하자. 구간 $[0, p_m]$에서 v_i개의 작업 $J_{i,1},...,J_{i,vi}$가 호출되므로 이 작업들의 여유시간이 모두 0 이상이어야 스케줄링 가능하다. 또한 i=1,...,m에 대해서 이 조건이 모두 만족되어야 한다. 재수행 벡터 $L_1,...,L_m$에 대한 스케줄링 가능성 함수를 $\Gamma(L_1,...,L_m)$라고 하자. $s_{i,j}$로 이 함수를 표현하면 다음과 같다.

$$\Gamma(L_1,L_2,...,L_m)=\prod_{i=1}^{m}\prod_{j=1}^{v_i}u(s_{i,j})$$

u(x)는 단위 계단 함수(unit-step function)로서 $x<0$에서 $u(x)=0$이고 $x\geq 0$에서 $u(x)=1$이다.

여유시간 $s_{i,j}$를 구하려면 선점형 RM 알고리듬의 특성을 이용해야 한다. RM 스케줄링의 특성상 어떤 태스크의 여유시간을 계산할 때에는 그 태스크보다 우선순위가 높거나 같은 태스크들만 고려하면 된다[7]. 즉 우선순위가 가장 높은 태스크 T_1부터 스케줄링 가능성 검사를 시작하며, 태스크 T_i의 작업 $J_{i,j}$의 여유시간을 계산할 때에는 Φ_i에 속한 태스크들이 $J_{i,j}$의 호출 시각부터 절대적 데드라인까지 차지하는 실행시간만 찾으면 된다.

여유시간을 구할 때 중점적으로 고려해야 할 또 하나의 사항은 태스크 T_i의 상대적 데드라인 d_i가 주기 p_i보다 크다는 사실이다. 상대적 데드라인과 주기가 동일한 설정에서는 [6, 10] 어떤 작업의 호출 시점에서 절대적 데드라인 사이에 호출되는 현재 태스크의 작업은 항상 자기 자신뿐이었다. 같은 구간에서 현재 태스크 T_i보다 우선순위가 높은 태스크의 T_j의 작업도 simply periodic 조건에 의해서 p_i/p_j개가 호출된다. 그러나 $d_i>p_i$인 본 논문의 조건 하에서는 현재 태스크 T_i의 작업이 수행 완료될 때까지 최소한 한 개 이상의 새로운 T_i의 작업이 호출될 수 있다. 마찬가지로 T_i보다 우선순위가 높은 태스크의 작업들도 $d_i=p_i$인 경우보다 더 많이 호출되므로 이전 연구에서 사용했던 기법과는 다른 형태의 스케줄링 가능성 판별식을 찾아야 한다.

스케줄링 가능성 판별을 위해서 'level-i busy 구간' 개념을 도입한다. 고정 우선순위 알고리듬으로 스케줄링되는 멀티 태스크 **T**의 실행 과정에서 level-i busy 구간의 정의는 다음과 같다[7].

정의 1: level-i busy 구간 $(t_s, t_e]$: 프로세서가 Φ_i에 속하는 태스크의 작업들만 실행하면서 "busy"한 구간이다. t_s 이전에 호출된 Φ_i의 작업들은 모두 t_s 이전에 실행 완료되어야 한다. 즉 t_s가 Φ_i 작업의 실행 도중에 위치해서는 안 된다. 또 t_s 이후에 호출된 Φ_i의 작업들은 모두 t_e 이전까지 실행 완료되어야 하며, $(t_s, t_e]$ 구간 안에 프로세서가 idle하는 시간이 하나도 없어야 한다.

level-i busy 구간이 현재 작업의 절대적 데드라인 근처에 존재한다면 프로세서는 level-i busy 구간이 시작하기 전에 현재의 작업을 수행 완료해야 한다. 즉 level-i busy 구간의 존재는 현재 작업의 절대적 데드라인이 줄어드는 효과를 낸다. 이 현상을 정량적으로 표시하기 위한 매개 변수가 '유효 데드라인(effective deadline)'이다[13].

$J_{i,j}$의 유효 데드라인을 $d^e_{i,j}$라고 정의하자. $d^e_{i,j}$는 $J_{i,j}$가 실제 만족시켜야 할 (절대적) 데드라인을 말한다. Level-i busy 구간을 이용하여 $d^e_{i,j}$의 식을 표현하면 아래와 같다.

$$d^e_{i,j} = \begin{cases} d_{i,j} & d_{i,j}\text{가 level-i busy 구간에 포함되지 않을 때} \\ t_s & d_{i,j}\text{가 level-i busy 구간 } (t_s, t_e]\text{에 포함될 때} \end{cases}$$

유효 데드라인 $d^e_{i,j}$와 더불어 $J_{i,j}$의 '누적 여유시간(cumulative slack time)'을 $\sigma_{i,j}$라고 정의한다. $\sigma_{i,j}$는 스케줄링 시점 0부터 시작하여 T_i의 j번째 작업 $J_{i,j}$가 실행될 때까지 누적된 T_i의 여유시간의 합이다. 즉 $\sigma_{i,j}$는 $J_{i,1}$, $J_{i,2}$,...,$J_{i,j}$가 가지는 여유시간을 모두 더한 값이다. 유효 데드라인 $d^e_{i,j}$를 이용하여 $\sigma_{i,j}$를 표현하면 다음과 같다[7].

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

위 식에서 $\lceil x \rceil$는 x보다 크거나 같은 최소 정수를 의미한다. 정의에 의해서 $J_{i,j}$의 여유시간 $s_{i,j}$를 $\sigma_{i,j}$로 나타내면

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

이다.

재수행 벡터 L_1,...,L_m이 주어질 때 선점형 RM 알고리듬의 스케줄링 가능성 검사를 하는 절차를 요약하면 다음과 같다.

알고리듬 1: 스케줄링 가능성 검사

1) 고장이 일어나지 않은 초기 상태에서 선점형 RM 알고리듬을 적용하고 level-i busy 구간을 구한다.
2) 재수행 벡터 L_1,...,L_m의 값을 대입하여 job의 실제 실행시간 $e_{i,j}$를 구한다.
3) $e_{i,j}$를 적용하여 level-i busy 구간 값을 갱신한다.
4) 유효 데드라인 $d^e_{i,j}$를 구한다.
5) $d^e_{i,j}$를 이용하여 누적 여유시간 $\sigma_{i,j}$와 여유시간 $s_{i,j}$를 찾는다.
6) $\sigma_{i,j}$ 값을 함수 $\Gamma(L_1,...,L_m)$에 대입하여 스케줄링 가능성을 판별한다.

예제 1: m=2, 즉 $\mathbf{T}=\{T_1, T_2\}$이고 $p_2=2p_1$, $d_2=2.6p_1$인 실시간 시스템이 있다고 하자. 그림 1(a)는 $\mathbf{T}$가 어떤 재수행 벡터 L_1과 L_2를 가지고 RM 알고리듬으로 스케줄링된 예를 보여준다. 고장에 의한 재수행이 발생했다고 가정하므로 태스크 각 작업의 실행시간은 서로 다를 수 있다. 그림 1(a)에서 d_2가 level-2 busy 구간 안에 있지 않으므로 $J_{2,1}$의 유효 데드라인 $d^e_{2,1}$은 절대적 데드라인 d_2와 동일하다. (마지막 태스크 T_m의 절대적 데드라인 $d_{m,1}$은 항상 상대적 데드라인 d_m과 같다.) 따라서 식 (1)에서 r_k는

$$r_1 = \lceil d_2/p_1 \rceil = \lceil 2.6p_1/p_1 \rceil = 3,$$
$$r_2 = \lceil d_2/p_2 \rceil = \lceil 2.6p_1/2p_1 \rceil = 2$$

이고 $\sigma_{2,1}$는

$$\sigma_{2,1} = d_2 - (e_{1,1}+e_{1,2}+e_{1,3}+e_{2,1}+e_{2,2})$$

와 같이 나온다. 한편 그림 1(b)는 $J_{1,3}$과 $J_{2,2}$에서 체크포인트 재수행이 그림 1(a)보다 더 많이 발생하여 실행시간이 늘어난 경우이다. 그림에서 볼 수 있듯이 $J_{1,3}$과 $J_{2,2}$가 만드는 level-2 busy 구간이 절대적 데드라인 d_2를 포함하므로 $J_{2,1}$의 유효 데드라인 $d^e_{2,1}$은 level-2 busy 구간의 시작 시점 $t=2p_1$로 줄어든다. 이때 식 (1)의 r_k는 그림 1(a)와 달리 $r_1=2$, $r_2=1$이 되며 누적 여유시간 $\sigma_{2,1}$은

$$\sigma_{2,1} = d_2 - (e_{1,1}+e_{1,2}+e_{2,1})$$

이다. $J_{2,1}$의 여유시간 $s_{2,1}$은 정의에 의해서 $s_{2,1} = \sigma_{2,1}$이다.

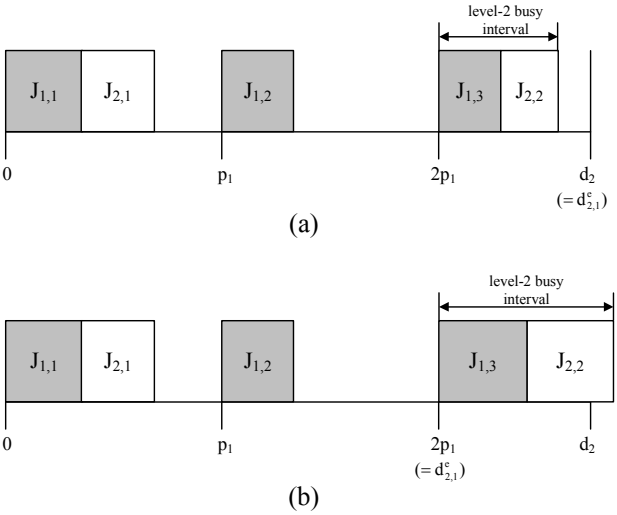

그림 1 2-태스크 실시간 시스템의 여유 시간: (a) $d^e_{2,1}=d_{2,1}$, (b) $d^e_{2,1}<d_{2,1}$.

Fig. 1 Slack time of a 2-task real-time system: (a) $d^e_{2,1}=d_{2,1}$, (b) $d^e_{2,1}<d_{2,1}$.

4. 최적 체크포인트 구간

4.1 T_i의 성공 확률

태스크 T_i가 재수행 벡터 $L_i=[l_{i,1}\ l_{i,2}\ \cdots\ l_{i,vi}]$를 가지면 T_i의 j번째 작업 $J_{i,j}$의 실행 시 체크포인트 구간 재수행이 $l_{i,j}$번 일어난다는 뜻이다. 구간 재수행 횟수 $l_{i,j}$를 가진 $J_{i,j}$의 수행이 성공적으로 완료될 확률을 먼저 구한다.

T_i의 각 작업에 n_i개의 체크포인트를 삽입하므로 $J_{i,j}$는 체크포인트가 하나씩 삽입된 n_i개의 모듈로 나누어져 실행된다. $J_{i,j}$가 수행 완료되려면 프로세서가 이 n_i개의 모듈을 모두 실행해야 한다. 재수행 횟수 $l_{i,j}$가 의미하는 바는 $J_{i,j}$에 과도 고장이 발생하여 체크포인트 롤백(rollback)이 $l_{i,j}$번 일어난다는 뜻이다. 즉 프로세서가 $J_{i,j}$를 완료하기 위해 총 $n_i+l_{i,j}$번의 모듈 실행을 한다. n_i개의 모듈 중 고장이 발생한 모듈은 체크포인트 롤백에 의해서 두 번 이상 실행될 것이다. 그런데 $n_i+l_{i,j}$번의 모듈 실행 중 맨 마지막 실행에서는 과도 고장이 한 번도 발생하지 않는다. 그렇지 않고 고장이 발생한다면 체크포인트 롤백이 일어나야 하므로 이 모듈 실행이 맨 마지막이라는 전제와 모순된다.

$l_{i,j}$번의 재수행 횟수를 가지면서 $J_{i,j}$가 성공적으로 수행 완료되는 경우의 수를 구해보자. 이 값은 프로세서가 총 $n_i+l_{i,j}$번의 모듈 실행을 하고 그 중 고장에 의한 rollback이 $l_{i,j}$번

일어나는 경우의 수와 같다. 그런데 앞에서 맨 마지막 모듈 실행에서는 고장이 발생하지 않아야 한다고 했다. 따라서 구하고자 하는 총 경우의 수는 $_{n_i+l_{i,j}-1}C_{l_{i,j}}$이다. $_xC_y$는 x개에서 y개를 선택하는 조합(combination)이다.

과도 고장이 Poisson 분포를 보이기 때문에 $J_{i,j}$에서 어떠한 조합으로 $l_{i,j}$번의 체크포인트 구간 재수행이 발생하든지간에 각 경우에 대한 발생 확률은 동일하다. T_i의 임의 체크포인트 구간에서 과도 고장이 한 번도 발생하지 않을 확률이 p_i, 적어도 한 번 이상 발생할 확률이 q_i이므로 $J_{i,j}$에서 $l_{i,j}$번의 체크포인트 구간 재수행이 발생하는 단위 확률은 $p_i^{n_i}q_i^{l_{ij}}$이다. $l_{i,j}$번의 재수행 구간을 가진 $J_{i,j}$의 실행이 성공적으로 끝날 확률을 $\Psi_{ij}(l_{i,j})$라 하면 $\Psi_{i,j}(l_{i,j})$는 앞 설명으로부터 다음과 같이 유도된다.

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

시간 구간 [0, p_m]에서 재수행 벡터 L_i를 가지는 T_i의 v_i개의 작업이 모두 성공적으로 실행될 확률을 $\Psi_i(L_i)$라고 정의하자. $\Psi_i(L_i)$는 T_i의 각 작업의 성공 확률을 모두 곱한 값이므로

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

이다.

4.2 최적화 문제 설정

$\Psi_i(L_i)$는 재수행 벡터 L_i에 대한 단일 태스크 T_i의 실행 성공 확률이다. $\Psi_i(L_i)$ 표현을 전체 태스크 $\mathbf{T}$로 확장시킨다. $\mathbf{T}$가 가지는 m개의 재수행 벡터가 $L_1,L_2,...,L_m$일 때 구간 [0, p_m] 동안 호출되는 $\mathbf{T}$의 모든 job이 수행 성공할 확률을 $\Psi(L_1,...,L_m)$이라 하자. $\Psi(L_1,...,L_m)$은 $\mathbf{T}$가 가지는 m개의 확률변수 $\Psi_i(L_i)$를 모두 곱하여 구할 수 있다. 그런데 3장에서 확인했듯이 재수행 벡터 $L_1,L_2,...,L_m$에 대해서 RM 알고리듬의 스케줄링이 가능하지 않을 수 있다. 앞에서 구한 스케줄링 가능성 함수 $\Gamma(L_1,...,L_m)$을 곱하면 RM 알고리듬의 스케줄링이 불가능할 때 태스크 수행 성공 확률을 0으로 만들 수 있다. 따라서 확률 $\Psi(L_1,...,L_m)$은 다음과 같이 유도된다.

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

$\Psi(L_1,...,L_m)$은 체크포인트 개수 $n_1,n_2,...,n_m$이 재수행 벡터 $L_1,L_2,...,L_m$이 주어질 때 가지는 전체 태스크 수행 성공 확률이다. 앞서 밝혔듯이 재수행 벡터 $L_1,L_2,...,L_m$은 과도 고장에 의해서 확률적으로 결정되므로 실시간 시스템의 운용 시 가능한 모든 재수행 벡터가 발생한다고 봐야 한다. 체크포인트 개수 $n_1,n_2,...,n_m$이 가지는 전체 태스크 수행 성공 확률은 모든 재수행 벡터에 대한 확률 값을 다시 더해야 완전하게 구해진다. 따라서 우리는 과도 고장이 하나도 발생하지 않은 초기 상태인 $L_1=L_2=\cdots=L_m=0$부터 시작하여 주어진 멀티태스킹에서 가능한 모든 재수행 벡터의 경우의 수를 찾아야 한다.

T_i의 재수행 벡터 $L_i=[l_{i,1},...,l_{i,vi}]$의 원소 $l_{i,j}$가 가질 수 있는 최대값을 $l^m_{i,j}$라 하자. $l^m_{i,j}$를 구하기 위해 $s_{i,j}(0)$을 고장이 발생하지 않은 초기 상태에서 RM 알고리듬을 적용했을 때 $J_{i,j}$가 가지는 여유시간이라고 정의하자. $J_{i,j}$에서 체크포인트 구간이 k번 재수행되면(k≥0) 여유시간 $s_{i,j}(0)$에서 $k\Delta_i$만큼의 체크포인트 구간이 소비된다. 따라서 $J_{i,j}$에서 재수행될 수 있는 체크포인트 구간의 최대 횟수 $l^m_{i,j}$는 아래와 같이 유도된다.

$$l^m_{i,j} = \left\lfloor \frac{s_{i,j}(0)}{\Delta_i} \right\rfloor$$

위 식에서 ⌊x⌋는 x보다 작거나 같은 최대 정수이다. 본 논문에서 다루는 태스크는 데드라인이 주기보다 크기 때문에 초기 상태에서 작업 $J_{i,j}$가 가지는 여유시간 $s_{i,j}(0)$도 데드라인이 주기와 똑같은 경우보다 더 큰 값이 나온다. 위 식에서 보듯이 $l^m_{i,j}$ 값도 따라서 커지기 때문에 가능한 재수행 벡터의 가지 수도 커지고 전체 태스크 성공 확률도 증가할 것이다.

간편한 표기를 위해서 $\mathbf{L}$:={$L_1,L_2,...,L_m$}이라 하고 $\mathbf{L}$이 가질 수 있는 가능한 모든 경우의 집합을 $\mathbf{\Lambda}$이라 하자. $\mathbf{\Lambda}$은 L_i의 각 원소 $l_{i,j}$를 0,1,...,$l^m_{i,j}$까지 변화시켜 가면서 구할 수 있다.

시간 구간 [0, p_m]에서 호출되는 멀티 태스크 $\mathbf{T}$의 모든 작업이 성공적으로 실행될 확률을 $P(\mathbf{T})$라고 하자. $P(\mathbf{T})$는 $\mathbf{\Lambda}$에 속한 모든 재수행 벡터에 대한 $\Psi(L_1,...,L_m)$을 더한 값이다. 아래는 $P(\mathbf{T})$의 최종 표현식이다.

$$P(\mathbf{T}) = \sum_{\mathbf{L}\in\mathbf{\Lambda}}\left[\Gamma(L_1,...,L_m)\prod_{i=1}^{m}\Psi(L_1,...,L_m)\right]$$

위 확률은 체크포인트 구간 길이 Δ_i, 즉 삽입되는 체크포인트 개수 $n_1,...,n_m$에 대한 함수이다. 그러므로 가능한 모든 체크포인트 개수에 대한 $P(\mathbf{T})$의 최대값을 찾는 최적화 문제가 성립하며, $P(\mathbf{T})$를 최대로 하는 체크포인트 개수를 찾으면 최적의 체크포인트 구간 $\Delta_1,...,\Delta_m$을 얻을 수 있다.

최적화 문제의 목적 함수를 구한 후에는 체크포인트 개수 n_i가 가지는 제한 조건(constraint)을 찾는다. 먼저 n_i는 1 이상의 자연수이므로

$$n_i \in \mathbb{N}, \forall i=1,...,m$$

이다. 다음으로 n_i가 변할 수 있는 범위를 구하기 위해서 $\mathbf{T}$에 체크포인트를 하나도 삽입하지 않고 고장도 한 번도 발생하지 않은 상태를 생각하자. $\mathbf{T}$ 중 우선순위가 가장 낮은 태스크 T_m의 여유시간을 체크하면 n_i의 경계 값을 구할 수 있다. T_m은 시간 구간 [0, p_m]에서 하나의 작업 $J_{m,1}$을 호출하며, 이 작업은 [0, $d^e_{m,1}$] 동안 실행을 끝내야 한다. 태스크 T_i의 job 중 [0, $d^e_{m,1}$] 동안 실행이 끝나야 하는 작업의 개수를 w_i라 하자. w_i번째 작업 J_{i,w_i}의 유효 데드라인 d^e_{i,w_i}는 $d^e_{m,1}$보다 작거나 같고 w_i+1번째 작업 $J_{i,(w_i+1)}$의 유효 데드라인 $d^e_{i,(w_i+1)}$는 $d^e_{m,1}$보다 크다. 따라서 w_i는 다음 식을 만족한다.

$$d^e_{i,w_i} \le d^e_{m,1} < d^e_{i,(w_i+1)}$$

체크포인트가 없고 고장도 발생하지 않은 초기 상태에서 작업 $J_{m,1}$이 가지는 여유시간을 I_m이라 하면 I_m은 아래와 같이 유도된다.

$$I_m = d^e_{m,1} - (w_1e_1 + w_2e_2 + \cdots + w_me_m)$$

태스크 T_i에 n_i개의 체크포인트를 삽입하면 구간 [0, $d^e_{m,1}$]에서 T_i의 작업이 w_i개 완료되어야 하므로 총 w_in_i개의 체크포인트가 삽입된다. 또 체크포인트 한 개당 부담(overhead) t_{cp}를 필요로 하므로 T_i가 요구하는 체크포인트 부담 총량은 $t_{cp}w_in_i$이다. 그런데 $\mathbf{T}$에 속한 모든 태스크의 체크포인트 부

담 총량을 더한 값은 앞에서 구한 I_m보다 클 수가 없다. 따라서 체크포인트 개수 $n_1,...,n_m$은 다음 조건을 만족시켜야 한다.

$$w_1n_1 + w_2n_2 + \cdots + w_mn_m \le I_m/t_{cp}$$

멀티 태스크의 수행 성공 확률을 최대로 하는 체크포인트 개수를 $\{n^*_1,n^*_2,...,n^*_m\}$이라 하면 $\{n^*_1,n^*_2,...,n^*_m\}$은 $P(\mathbf{T})$를 목적 함수로 하고 위에서 구한 I_m에 대한 제한 조건을 가지는 다음 비선형 최적화 문제의 해이다.

$$\text{Maximize } P(\mathbf{T})$$
$$\text{Subject to } w_1n_1 + w_2n_2 + \cdots + w_mn_m \le I_m/t_{cp}$$
$$n_1, n_2, ..., n_m \in \mathbb{N}$$

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

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

5. 모의실험

두 개의 태스크 $\mathbf{T} = \{T_1, T_2\}$를 가지는 실시간 시스템에 대해서 본 논문에서 제안한 최적화 기법을 적용한다. 데드라인과 주기의 상대적인 크기에 따른 최적화 결과 차이를 알아보기 위해서 각 태스크의 매개 변수를 변화시켜 가면서 모의실험을 실시한다.

i) $d_1 > p_1,\ d_2 > p_2$

먼저 데드라인이 주기보다 큰 경우를 생각하자. T_1=(2, 0.6, 3), T_2=(4, 1.3, 5)라고 설정한다. 우선순위가 낮은 T_2의 주기가 4이므로 구간 [0, 4]에서 T_1, T_2의 작업의 개수는 각각 v_1=2, v_2=1이다. 그런데 T_2의 데드라인이 5이므로 구간 [4, 5]에서 T_1의 세 번째 작업 $J_{1,3}$도 실행된다(그림 1 참조). 따라서 T_2의 여유시간을 구할 때는 $J_{1,3}$의 실행 형태도 고려해야 한다. 실시간 시스템에서 발생하는 과도 고장은 발생율 λ=0.1인 Poisson 분포를 따르고 체크포인트 부담은 t_{cp}=0.05라고 가정한다.

그림 2는 체크포인트 개수 n_1과 n_2 변화에 따른 성공 확률 $P(\mathbf{T})$를 나타낸다. $P(\mathbf{T})$의 최대값은 그림 2에서 나타내었듯이 $\{n^*_1, n^*_2\}$={3, 6}일 때 0.9999이다. 따라서 최적의 체크포인트 구간 Δ^*_i는 아래와 같이 나온다.

$$\Delta^*_1 = 0.6/3 + 0.05 = 0.250$$
$$\Delta^*_2 = 1.3/6 + 0.05 = 0.267$$

ii) $d_1 = p_1,\ d_2 = p_2$

다음으로 각 태스크의 데드라인과 주기가 동일하다고 하고 T_1=(2, 0.6, 2), T_2=(4, 1.3, 4)이라고 하자. i)과 비교하면 각 태스크의 주기와 실행시간은 변화가 없지만 데드라인이 줄어들었다. 그림 3은 본 설정을 가지는 실시간 시스템에 대한 태스크 성공 확률 $P(\mathbf{T})$를 계산한 결과를 나타낸다. $P(\mathbf{T})$의 최대값은 $\{n^*_1, n^*_2\}$={2, 5}일 때 0.9996으로 나왔으며, 그림 2와 비교하여 확률 값이 약간 감소했음을 알 수 있다. 뿐만 아니라 확률 $P(\mathbf{T})$를 최대로 만드는 체크포인트 수도 차이가 났다. 본 실험 결과는 최적의 체크포인트 개수가 각 태스크의 실행시간과 주기, 그리고 데드라인의 상대적인 크기에 따라서 결정된다는 사실을 알려준다. 또 그림 2와 그림 3의 결과에서 볼 수 있듯이 데드라인이 주기보다 큰 경우의 태스크 성공 확률은 일반적으로 데드라인이 주기와 동일한 경우보다 크게 유도된다.

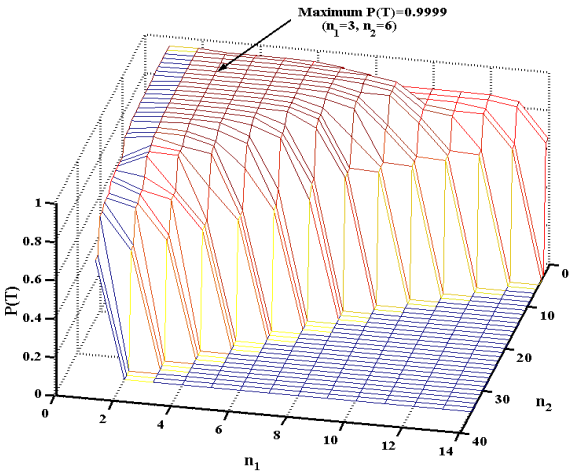

그림 2 T_1=(2, 0.6, 3), T_2=(4, 1.3, 5)일 때 체크포인트 수에 따른 $P(\mathbf{T})$ 확률 변화.

Fig. 2 $P(\mathbf{T})$ vs. checkpoint numbers for the system with T_1=(2, 0.6, 3) and T_2=(4, 1.3, 5).

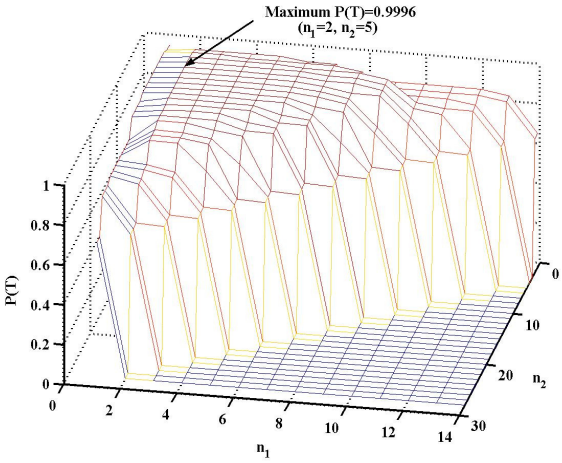

그림 3 T_1=(2, 0.6, 2), T_2=(4, 1.3, 4)일 때 체크포인트 수에 따른 $P(\mathbf{T})$ 확률 변화.

Fig. 3 $P(\mathbf{T})$ vs. checkpoint numbers for the system with T_1=(2, 0.6, 2) and T_2=(4, 1.3, 4).

6. 결 론

본 논문에서는 데드라인이 주기보다 큰 태스크들로 이루어진 실시간 시스템의 체크포인트 구간을 결정하는 기법을 다루었다. 데드라인이 주기보다 큰 태스크들을 RM 알고리듬으로 스케줄링하면 각 태스크 작업이 만족시켜야 할 실제 데드라인이 주기마다 다르게 나오므로 데드라인과 주기가 동일한 경우와는 다른 해석법이 필요하였다. 본 논문에서는 확률 최적화 방법을 이용하여 멀티 태스크의 수행 성공 확

률을 최대화하는 체크포인트 개수와 구간 길이를 유도하였다. 주어진 재수행 벡터에 대해서 RM 알고리듬이 스케줄링 가능한지 여부를 판별하는 방법을 제안하고 최적화 목적 함수에 반영하였다. 데드라인이 주기보다 큰 경우 태스크 수행 성공 확률과 삽입되는 최적의 체크포인트 개수가 그렇지 않은 경우보다 더 늘어난다는 사실을 모의실험을 통해서 입증하였다.

참 고 문 헌

[1] L. Sterpone and M. Violante, “A new reliability-oriented place and route algorithm for SRAM-based FPGAs,” IEEE Transactions on Computers, vol. 55, no. 6, pp. 732-744, 2006.

[2] C. M. Krishina and K. G. Shin, Real-Time Systems, New York: McGraw-Hill, 1997.

[3] 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.

[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] A. Ziv and J. Bruck, “Analysis of checkpointing schemes with task duplication,” IEEE Transactions on Computers, vol. 47, no. 2, pp. 222-227, 1998.

[6] S. W. Kwak and B. K. Kim, “Task-scheduling strategies for reliable TMR controllers using task grouping and assignment,” IEEE Transactions on Reliability, vol. 49, no. 4, pp. 355-362, 2000.

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

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

[9] S. W. Kwak, B. J. Choi, and B. K. Kim, “An optimal checkpointing-strategy for real-time control systems under transient faults,” IEEE Transactions on Reliability, vol. 50, no. 3, pp. 293-301, 2001.

[10] 곽성우, 양정민, "임의 주기를 가지는 실시간 멀티 태스크를 위한 체크포인트 구간 최적화," 전기학회논문지, 제60권 제1호, pp. 193-200, 2011.

[11] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, “On task schedulability in real-time control systems,” 17th IEEE Real-Time Systems Symposium (RTSS ‘96), pp. 13-21, 1996.

[12] “과학기술위성 1호 상세점검회의 자료집,” 인공위성연구센터, 2002.

[13] M. R. Garey and D. S. Johnson, “Two processor scheduling with start time and deadlines,” SIAM Journal of Computing, vol. 6, pp. 416-426, 1977.

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 2월 한국과학기술원 전기 및 전자공학과 졸업(공박). 2000년～2002년 인공위성연구센터 선임연구원, 연구교수. 2003년～현재 계명대학교 전자공학과 부교수. 주관심분야: 실시간시스템, 내고장성 기법, 비동기 시스템, 위성 시스템 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