RM 스케줄링된 실시간 태스크에서의 최적 체크 포인터 구간 선정

論 文
56-6-20

Determination of Optimal Checkpoint Interval for RM Scheduled Real-time Tasks

郭 成 祐† · 鄭 容 朱*
(Seong-Woo Kwak · Young-Joo Jung)

Abstract - For a system with multiple real-time tasks of different deadlines, it is very difficult to find the optimal checkpoint interval because of the complexity in considering the scheduling of tasks. In this paper, we determine the optimal checkpoint interval for multiple real-time tasks that are scheduled by RM(Rate Monotonic) algorithm. Faults are assumed to occur with Poisson distribution. Checkpoints are inserted in the execution of task with equal distance in the same task, but different distances in other tasks. When faults occur, rollback to the latest checkpoint and re-execute task after the checkpoint. We derive the equation of maximum slack time for each task, and determine the number of re-executable checkpoint intervals for fault recovery. The equation to check the schedulibility of tasks is also derived. Based on these equations, we find the probability of all tasks executed within their deadlines successfully. Checkpoint intervals which make the probability maximum is the optimal.

Key Words : 체크포인터(Checkpoint), RM 스케줄링, 실시간 태스크(Real-time Task), 과도 고장(Transient Fault)

1. 서 론

제어 컴퓨터의 신뢰성을 높이는 방법으로는 하드웨어적인 중복 구조를 이용하는 방법과, 태스크 스케줄링, 체크 포인터(checkpoint) 삽입, 또는 재시도(retry) 기법과 같은 소프트웨어적인 방법들이 있다. 중복 구조를 이용한 하드웨어적인 방법들은 지금까지 많은 연구가 이루어져 왔으며, 현재 운용중인 많은 고 신뢰도 제어 시스템들은 하드웨어적인 중복 구조에 바탕을 두고 있다. 체크 포인팅 기법 또는 재시도 기법과 같은 소프트웨어적인 방법들은 부가적인 하드웨어뿐만 아니라 시간적인 오버헤드(overhead)를 요구하여 제어 시스템과 같은 실시간 시스템에 사용하는 것이 부적절한 것으로 지금까지 여겨져 왔다. 하지만 최근 컴퓨터의 처리 성능 향상과 OS(Operating System) 기술의 발전으로 재시도 또는 체크 포인팅 기법을 실제 실시간 시스템에 적용하는 것이 가능하여졌다. 특히 산업 현장에서 발생하는 시스템 고장들을 분석한 결과에 따르면 소자의 영구적 결함과 같은 하드웨어적인 원인 보다는 외부의 전기적, 기계적 환경 변화등과 같은 과도 고장에 의한 것이 다수를 이루고 있는 것으로 보고되고 있다[15]. 이와 같은 과도 고장은 중복구조를 이용한 하드웨어적 방법 보다는 체크 포인터 삽입 또는 재시도와 같은 소프트웨어적인 방법으로 더 잘 대처할 수 있다.

체크 포인팅 기법을 적용한 실시간 시스템을 해석하여 최적의 체크 포인팅 방법을 구하려는 연구는 기존의 다수 연구자들에 의해 연구 되어왔다[1,5,6,7,8,11,13,14]. Geist et. al[6]는 태스크의 수행 시간을 최소화하는 체크 포인터 삽입 방법을 연구 하였고, Shin et. al[7]은 고장의 지속시간이 없는 경우에 대하여 평균 수행 시간을 최소화 시키는 체크 포인터 삽입 방법을 구하였다. Krishna & Singh[1]은 지속 시간이 존재하는 고장의 상태를 상정하여, 체크 포인터 기법을 탑재한 시스템이 이중화(duplex) 또는 삼중화(triplex) 구조 중 어떤 구성이 더 효율적인지를 해석하였다. 반면, Mosse [14]는 데드라인이 서로 다른 비주기적인 실시간 태스크들이 EDF(Earliest Deadline First) 알고리즘에 의해 스케줄링(scheduling) 되는 상황에서 여러 개의 과도 고장을 극복하기 위한 EDF 스케줄링 가능성(feasibility-check)을 연구하였다. Ghosh[13]는 서로 다른 데드라인을 가진 실시간 태스크들이 존재하는 경우, 태스크 수행 중 고장이 발생 하였을 때 체크 포인터를 삽입하는 것이 아니라 해당 태스크를 재수행하여 고장을 극복하는 방법을 사용하여 전체 태스크들의 스케줄링이 가능한지를 판별하였다.

논문 저자도 이러한 연구를 수행 한바 있다[9,10,16]. 하지만 기존 연구의 대부분은 단일 태스크 또는 동일한 주기(또는 데드라인)를 가지는 여러 개의 태스크가 존재하는 시스템에 대하여 체크 포인팅 기법을 적용한 것이었다. 서로 다른 주기(또는 데드라인)를 가지는 태스크들이 존재하는 경우, 체크 포인팅 기법을 적용하려면 매우 복잡한 수학적인 해석이 필요하다. 이러한 어려움 때문에 이에 대한 연구는 현재까지 미미한 실정이다. 본 논문에서는 이 문제를 해결해 보고자 한다.

† 교신저자, 正會員 : 啓明大 工大 電子工學科 助敎授 · 工博
E-mail : ksw@kmu.ac.kr
* 正 會 員 : 啓明大 工大 電子工學科 副敎授 · 工博
接受日字 : 2007年 3月 27日
最終完了 : 2007年 5月 15日

연구자의 기존 연구에서 주기가 다른 태스크들이 존재하는 경우에, 근사적인 Fail 확률을 계산하여 체크 포인터를 삽입하는 연구를 진행하였다[16]. 이 연구에서는 태스크의 최대 주기이내에서 고장이 최대 1개 발생하는 근사적인 경우로 한정하였다. 대신 고장의 지속시간을 고려하여 매우 복잡한 확률식이 필요하였다. 본 논문에서는 기존의 연구를 확대하여 고장의 발생에 제한 조건이 없도록 하고, 고장의 지속시간이 없는 경우에 최적의 체크 포인터 삽입 방법을 연구하였다. 최적의 체크 포인터를 삽입하기 위한 척도로써 모든 실시간 태스크들이 자신의 주기이내에서 성공적으로 수행될 확률을 계산한다. 이 확률 식을 기반으로 태스크들의 성공적 수행 확률을 최대화 하는 최적의 체크 포인터 구간을 찾는다.

2. RM 스케줄링된 실시간 태스크

RM(Rate Monotonic) 스케줄링은 가장 일반적인 실시간 스케줄링 방법 중의 하나이다. RM 스케줄링에서는 태스크의 우선순위가 데드라인(deadline)에 반비례 한다. 따라서 가장 짧은 데드라인을 가진 태스크의 우선순위가 가장 높고, 가장 긴 데드라인을 가진 태스크의 우선순위가 가장 낮다. 본 논문에서 다루는 실시간 태스크들은 다음과 같은 특징을 가지는 것으로 가정한다.

기본가정:

A1. 태스크들은 Rate Monotonic(RM) 알고리즘에 의해 스케줄링 된다.

A2. 각 태스크의 주기는 가장 짧은 태스크 주기의 2의 승수 주기중의 하나이다. 즉 $D_i \in \{T, 2T, 2^2T, \cdots 2^MT\}$, T: 주기가 가장 짧은 태스크의 주기, D_i: 태스크 i의 주기, i: 자연수.

A3. 각 태스크의 데드라인(deadline)은 태스크의 실행 주기(period)와 같다.

위의 가정에 따라 각 태스크의 실행 주기는 그 태스크의 데드라인과 같으며, 실행 주기는 최소 주기의 2의 승수배로 이루어져 있다. 그림 1은 가정 A1. ~ A3.를 만족하는 세 태스크(τ_1, τ_2, τ_3)를 스케줄링 한 예이다. 태스크 τ_1은 주기와 데드라인이 T, 태스크 τ_2는 2T, 태스크 τ_3는 4T이다. 가장 우선순위가 높은 τ_1이 매 주기 T 마다 실행되고, 다음 우선순위인 τ_2가 2T 마다 태스크 τ_1의 수행이 끝난 후 실행된다. 우선순위가 제일 낮은 τ_3는 τ_1 과 τ_2 의 수행이 끝난 후 나머지 시간에 실행된다. 4T이후의 스케줄링 패턴은 아래의 0~4T 까지의 형태가 계속 반복된다. 즉 A1.~A3. 의 가정을 만족하는 태스크들의 스케줄링의 경우 최대 주기내의 스케줄링 패턴이 그 이후에 계속 반복된다. 따라서 최대 주기 이내의 스케줄링만을 고려하여 고장 극복 알고리즘을 설계할 수 있다. 만약 각 태스크가 자신의 주기(또는 데드라인)이내에 수행을 끝낼 수 없다면 시스템은 Fail 한다. 본 논문에서는 실시간 태스크의 고장 극복을 위해 각 태스크마다 체크 포인터를 삽입하고, 고장이 발생한 경우 가장 최근의 체크 포인터로 회귀하여 재 수행함으로써 고장을 극복하는 체크 포인팅 기법을 적용한다. 체크 포인팅 기법은 태스크가 자신의 데드라인 이내에서 가질 수 있는 여유 시간을 이용하여 고장 극복에 사용하는 방식이다. 하지만 그림 1과 같이 다수의 태스크가 동시에 스케줄링 되는 경우 여유 시간을 관리하기가 쉽지 않으며, 이들을 해석하여 최적의 방식을 찾는 것은 매우 어렵다.

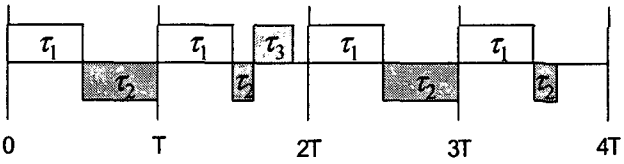

그림 1 RM 알고리즘을 이용한 태스크 스케줄링
Fig. 1 Task Scheduling using RM Algorithm

3. 고장 발생 및 극복

실시간 시스템이 직면하는 고장들 중 많은 부분이 과도 고장에 기인하는 것으로 보고되고 있다. 이와 같은 과도 고장이 고장 발생율 λ를 가진 Poisson 분포에 따라 발생한다고 할 때, t 시간내에 n개의 고장이 발생할 확률은 다음과 같다.

$$\alpha_{n(\lambda,t)} = \frac{(\lambda t)^n}{n!} e^{-\lambda t} \tag{1}$$

따라서 Δ 시간 구간에 고장이 없을 확률은

$$p = \alpha_0(\lambda, \Delta) = e^{-\lambda\Delta} \tag{2}$$

이며, Δ 시간 구간에 1개 이상의 고장이 발생할 확률은 다음과 같이 얻을 수 있다.

$$q = \sum_{n=1}^{\infty} \frac{(\lambda\Delta)^n}{n!} e^{-\lambda\Delta} = 1 - e^{-\lambda\Delta} \tag{3}$$

과도 고장이 식 (1)의 Poisson 분포에 따라 발생하는 경우, 실시간 태스크에 체크포인터를 삽입하여 고장을 극복할 수 있다. 체크 포인팅 시에는 현재 수행하고 있는 태스크의 모든 상태를 저장하고, 이들 상태에 오류가 있는지 검증(고장 탐지)하는 2가지 과정을 수행한다. 그림 2는 실시간 태스크에 고장이 발생한 경우, 체크포인팅 시 발생한 고장을 탐지하고, 고장이 발생한 구간을 재수행 함으로써 고장을 극복하는 과정을 나타낸다. 재수행 과정은 가장 최근의 체크 포인터로 회귀(roll back)하여 체크 포인터에 저장된 태스크 상태를 불러와 그 시점부터 태스크의 수행을 재개한다. 이렇게 함으로써 고장이 발생한 구간에 대한 고장극복을 할 수 있다. 본 논문에서는 고장의 발생과 탐지에 관련하여 다음과 같이 가정 한다.

A4. 고장은 고장 발생율 λ를 가진 Poisson 분포에 따라 발생한다.

A5. 고장에 대한 탐지는 체크 포인터 삽입 시 고장 탐지 알고리즘에 의해 수행되며, 발생한 모든 고장은 탐지 된다.

고장극복을 위해 고장이 발생한 구간을 재수행하는 것은 원래 태스크의 수행시간 연장을 가져온다. 태스크의 연장된 시간은 데드라인 이내이어야 하며, 다른 태스크의 수행을 보

장 할 수 있어야 한다. 즉 각 태스크가 데드라인 이내에서 가질 수 있는 여유 시간을 이용하여 태스크의 일정 부분을 재수행 함으로써 고장을 극복한다. 따라서 체크 포인터가 삽입되는 구간을 어떻게 결정하느냐에 따라 재수행 구간의 길이가 결정되고, 고장 극복 확율이 달라진다. 체크 포인터 수를 늘이면 재수행 구간은 줄어들지만 체크 포인팅을 위한 부담(overhead)은 늘어난다.

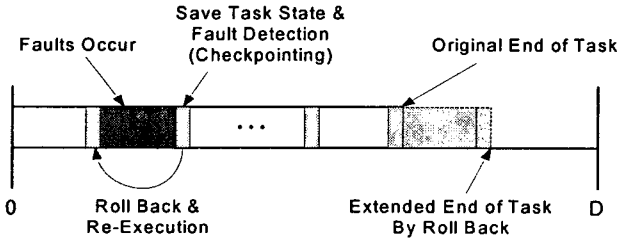

그림 2 체크포인터 삽입을 통한 고장 극복
Fig. 2 Fault Recovery using Checkpoint Placement

4. 최적 체크포인터 구간

4.1 태스크 실행 형태

실시간 태스크들이 A1. ~A5. 의 가정을 만족하고, 전체 m개의 태스크가 있다고 하자. 또한 i 번째 태스크는 τ_i 로 나타내고, τ_i의 데드라인(또는 실행주기)를 D_i 로 표시한다. τ_i의 인덱스 (index) i 는 태스크의 주기가 짧은 것부터 차례로 번호를 부여한다. 태스크 우선순위는 RM 알고리즘에 따라 주기가 짧을수록 우선순위가 높으므로 τ_1 이 우선순위가 가장 높고, τ_m이 우선순위가 가장 낮다. 따라서 D_1은 주기가 가장 짧은 태스크의 주기이고, D_m은 주기가 가장 긴 태스크의 주기가 된다. 즉 $D_1 \le D_2 \le D_3 \cdots \le D_m$.

고장극복을 위해 태스크 τ_i에 삽입된 체크 포인터 수를 n_i라고 하자. 또한 체크 포인터 오버헤드(overhead)는 t_{cp} , 체크포인터가 삽입되기 전의 태스크 τ_i의 원래 실행 시간은 E_i, τ_i의 체크포인터 삽입 구간을 Δ_i로 표시하자. 체크 포인터 오버헤드(overhead) t_{cp}는 태스크의 상태를 저장하는데 걸리는 시간과 고장 탐지알고리즘을 수행하는데 걸리는 시간이다. τ_i에 삽입된 체크 포인터 수가 n_i 이므로 τ_i의 체크포인터 구간 Δ_i는 다음과 같이 구할 수 있다.

$$\Delta_i = \frac{E_i}{n_i} + t_{cp} \tag{4}$$

따라서 τ_i에 체크포인터가 n_i개 삽입된 후 τ_i의 실행 시간 e_i 는 다음과 같다.

$$e_i = n_i \Delta_i = n_i \cdot \left(\frac{E_i}{n_i} + t_{cp}\right) = E_i + n_i t_{cp} \tag{5}$$

즉 τ_i의 원래 실행 시간에 체크 포인팅을 하는데 걸리는 시간이 더해져, 실행 시간이 $n_i t_{cp}$ 만큼 늘어난다.

체크 포인터를 이용한 고장 극복은 각 태스크가 자신의 데드라인 이내에 사용할 수 있는 여유 시간을 이용하여 고장이 발생한 부분을 재수행 함으로서 이루어진다. 따라서 몇 개의 체크 포인터를 삽입해야 하는가는 고장 발생율 뿐만 아니라 각 태스크의 여유시간과 밀접한 관계가 있다. 본 논문에서는 각 태스크마다 서로 다른 수의 체크 포인터를 삽입하여 고장 극복에 이용한다. 이것은 태스크 마다 각기 서로 다른 여유시간을 가지고 있으므로 태스크 마다 서로 다른 체크 포인터 수를 가져야 최적의 방법이 되기 때문이다. 최적의 체크 포인터 수를 구하기 위하여 본 논문에서는 먼저 고장이 발생한 상황에서 모든 태스크들이 자신의 데드라인 이내에 수행을 끝낼 확률식을 구하고, 다음으로 이 확률을 최대화 하는 체크 포인터 수를 각 태스크 별로 찾는 방식으로 진행한다.

고장이 발생하지 않은 상황에서 각 태스크가 가질 수 있는 최대 여유 시간은 다음과 같이 구할 수 있다.

$$S_{M_i} = D_i - \left(\frac{D_i}{D_1}\right)e_1 - \left(\frac{D_i}{D_2}\right)e_2 \cdots - \left(\frac{D_i}{D_i}\right)e_i \tag{6}$$

여기서 S_{M_i}는 τ_i 를 포함하여 τ_i와 우선 순위가 같거나 높은 모든 태스크가 스케줄링 되었을 때, τ_i 가 자신의 실행 주기 내에서 사용할 수 있는 최대 여유 시간이다. 식 (6)의 S_{M_i}는 τ_i보다 우선순위가 낮은 태스크의 수행은 고려하지 않고, 오로지 τ_i 와 τ_i보다 우선순위가 높은 태스크의 수행을 보장하는 경우의 최대 여유 시간이다. 그림 3과 그림 4는 주기가 T, 2T 인 두 태스크가 스케줄링 되었을 때 S_{M_i}를 나타내는 예이다. 그림 3은 τ_1보다 우선순위가 낮은 τ_2의 수행을 고려하지 않았을 때 τ_1이 가질 수 있는 최대 여유시간을 나타내며, 그림 4는 τ_2보다 우선순위가 높은 τ_1을 스케줄링하고 τ_2가 가질 수 있는 최대 여유시간을 τ_2의 수행 시간에 따라 2가지 경우로 나누어 나타낸 것이다.

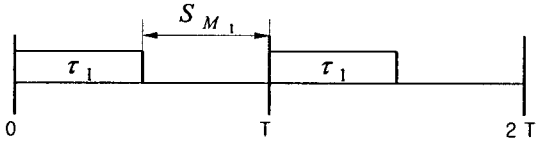

그림 3 우선순위가 가장 높은 τ_1의 최대여유시간(S_{M_1})
Fig. 3 Maximum slack time for task τ_1

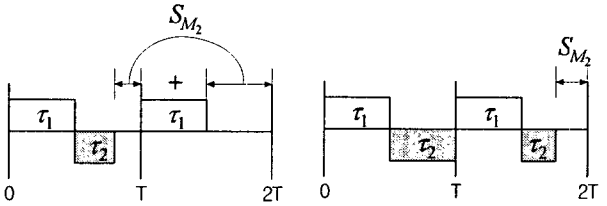

그림 4 τ_2의 최대 여유 시간(S_{M_2})
Fig. 4 Maximum slack time for task τ_2

그림 5는 τ_i의 한주기 내에서의 실행 형태를 나타낸 것이다. D_i 주기 내에서 τ_i 보다 우선순위가 높은 $\tau_1 \sim \tau_{i-1}$이 먼저 수행된 후 τ_i 가 실행된다. 또한 τ_i 의 실행 도중 τ_i 보다 우선순위가 높은 태스크의 실행을 위해 τ_i은 선점(preemption) 될 수 있다. 그림 5에서 사선 박스로 표시된 부분이 다른 태스크에 의해 선점된 부분을 나타낸다. 고장이 Poisson 분포에 따라 발생하고 고장의 지속시간이 없는 경우에는 고장의 발생 측면에서 보았을 때 그림 5의 실행 형태는 그림 6과 같이 우선순위가 높은 태스크에 의해 선점(preemption)된 부분을 한 곳에 모아 처음에 수행 되도록 하고, τ_i의 수행 중 선점된 부분을 제거한 후 τ_i만을 따로 수행하는 것으로 생각 할 수 있다. 이것은 Poisson 분포의 특성상 과거의 고장 발생 기억과 상관없이 앞으로의 고장 발생 확률이 결정된다는 것에 기인한다. 이러한 특성 때문에 태스크 τ_i가 자신의 주기 이내에서 성공적으로 수행될 확률은 그림 6의 태스크 수행 형태에서 Δ_i로 이루어진 체크포인터 구간이 n_i개 성공적으로 수행될 확률과 같다. 고장이 발생하여 l_{ik} 만큼의 구간이 손상을 입은 경우 $l_{ik}\Delta_i$ 만큼 수행시간이 늘어난다. 이때 늘어난 수행 시간은 τ_i가 가질 수 있는 최대 여유시간 (S_{M_i}) 이내 이어야 한다.

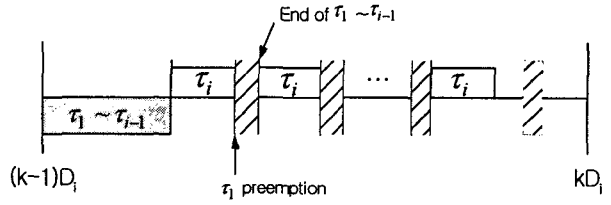

그림 5 τ_i의 한주기 내에서의 실행 형태

Fig. 5 Execution pattern of τ_i in one period

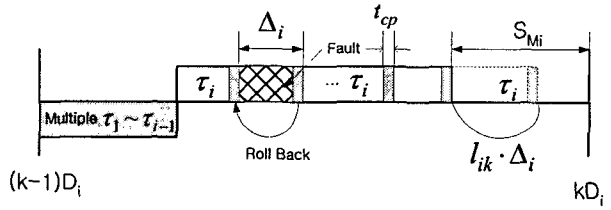

그림 6 타 태스크의 선점(preemption)을 한꺼번에 고려한 τ_i의 한주기 실행 형태

Fig. 6 Execution pattern of τ_i in one period considering preemption of other tasks at a time

4.2 태스크의 성공적 실행 확률

v_i를 태스크의 최대 주기(D_m)내에 있는 τ_i의 주기 수로 정의하면 v_i는 다음과 같이 구할 수 있다.

$$v_i = \frac{D_m}{D_i} \tag{7}$$

따라서 D_m 내에는 v_i개의 τ_i이 실행되는 주기가 존재한다. D_m이 가장 긴 태스크의 주기이므로 가정 A2.에 의해 v_i는 항상 정수 값이 된다.

$L_i = [l_{i1}, l_{i2}, \cdots l_{iv_i}]$를 τ_i의 각 주기에서 고장에 의해 재수행된 체크포인터 구간 수를 나타내는 벡터로 정의한다. 예를 들어 $L_1 = [2, 1, \cdots, 0]$인 경우 τ_1이 첫째 주기에서 재수행된 구간이 2개이고, 두 번째 주기에서는 1개, 그리고 마지막 주기에서는 재수행된 체크 포인터 구간 수가 없다는 것을 나타낸다.

$\psi_i(L_i)$를 τ_i의 각 주기에서 재수행 체크포인터 구간 수를 나타내는 벡터가 L_i와 같을 때 τ_i이 최대 주기(D_m)이내에서 수행을 끝낼 확률로 정의하면, $\psi_i(L_i)$ 는 다음과 같이 구할 수 있다.

$$\psi_i(L_i) = \left({}_{n_i+l_{i1}-1}C_{l_{i1}}\, p_i^{n_i} \cdot q_i^{l_{i1}}\right) \cdot \left({}_{n_i+l_{i2}-1}C_{l_{i2}}\, p_i^{n_i} \cdot q_i^{l_{i2}}\right) \cdots \left({}_{n_i+l_{iv_i}-1}C_{l_{iv_i}}\, p_i^{n_i} \cdot q_i^{l_{iv_i}}\right) \tag{8}$$

여기서 $p_i = e^{-\lambda\Delta_i}$는 Δ_i 구간내에서 고장이 없을 확률이고, $q_i = 1 - e^{-\lambda\Delta_i}$ 는 Δ_i 구간내에서 고장이 발생할 확률을 나타낸다. $\psi_i(L_i)$ 는 v_i개의 확률값의 곱으로 구성되어 있다. 첫 번째 확률값 $\left({}_{n_i+l_{i1}-1}C_{l_i}\, p_i^{n_i} \cdot q_i^{l_{i1}}\right)$은 τ_i의 첫 번째 주기에서 성공적으로 수행을 끝낼 확률이다. 이것은 n_i개의 체크 포인터 구간에서 고장이 없을 확률($p_i^{n_i}$)과 l_{i1} 개 구간에서 고장에 의해 손상을 입을 확률($q_i^{l_i}$)을 곱한 것이다. τ_i의 수행이 n_i개의 체크 포인터 구간으로 구성되므로 n_i개의 체크 포인터 구간에서 고장이 없어야 하며, τ_i의 첫 번째 주기에서 고장에 의해 재수행된 구간 수를 나타내는 L_i의 첫 번째 원소가 l_{i1} 이기 때문에 l_{i1}개의 구간에서는 고장이 있어야 한다. 또한 전체 $n_i + l_{i1}$ 구간 중 고장 있는 체크포인터 구간이 l_{i1}개 존재하면서 n_i개의 고장 없는 체크포인터 구간이 서로 배열되는 경우의 수를 구해야 한다. 이때 마지막 체크포인터 구간인 $n_i + l_{i1}$번째 구간은 항상 고장이 없이 끝나야 한다. 태스크 수행 중 마지막 체크포인터 구간은 고장이 없어야 태스크의 수행이 완료되기 때문이다. 따라서 마지막 구간을 제외한 $n_i + l_{i1} - 1$ 구간 중 l_{i1}개의 고장이 있는 체크포인터 구간이 배치되는 경우의 수는 ${}_{n_i+l_{i1}-1}C_{l_i}$이므로 $\left({}_{n_i+l_{i1}-1}C_{l_i}\, p_i^{n_i} \cdot q_i^{l_{i1}}\right)$은 재수행 체크포인터 구간 수가 l_{i1} 개 이면서 첫 번째 주기에서 τ_i이 성공적으로 수행을 마칠 확률이 된다. 두 번째 이후 확률식도 위와 같은 원리로 만들어 진다. 따라서 식(8)의 v_i개 확률값의 곱은 τ_i이 D_m이내의 각 주기에서 재수행 벡터 L_i를 가지면서 성공적으로 수행을 끝낼 확률이 된다.

다음으로 Γ_r 을 $L_1, L_2, \cdots L_r$ 의 재수행 형태를 가진 태스크 $\tau_1, \tau_2 \cdots \tau_r$이 최대 주기 D_m이내의 모든 주기에서 RM 알고리즘에 의해 스케줄링이 가능하면 "1"의 값을 가지고, 스케줄링이 불가능 하면 "0" 값을 가지도록 정의 한다.

$$\Gamma_r = \begin{cases} 0 & \text{if } \tau_1 \cdots \tau_r \text{ are unschedulable} \\ 1 & \text{if } \tau_1 \cdots \tau_r \text{ are schedulable} \end{cases} \quad (9)$$

또한 γ_r을 최대 주기 D_m내에 있는 τ_r의 각 주기에서 여유 시간이 존재하는지를 체크하는 변수로 정의하고, 모든 주기에서 여유시간이 존재하면 값이 "1", 여유시간이 존재하지 않은 주기가 하나라도 존재하면 값이 "0"이 되도록 하자. γ_r은 다음의 식으로 표현할 수 있다.

$$\gamma_r = \delta(s_r^1) \bullet \delta(s_r^2) \cdots \delta(s_r^{v_r}) \quad (10)$$

$$\delta(x) = \begin{cases} 1 & , \ x \geq 0 \\ 0 & , \ x < 0 \end{cases} \quad (11)$$

여기서 s_r^k은 $\tau_1, \tau_2 \cdots \tau_r$의 재실행 벡터가 $L_1, L_2, \cdots L_r$일 때, τ_r의 k 번째 주기에서의 여유시간을 나타낸다. 따라서 식 (10)의 $s_r^1, s_r^1 \cdots s_r^{v_r}$은 τ_r의 첫 번째 주기, 두 번째 주기, ... D_m내의 마지막 주기에서의 여유 시간을 나타낸다. γ_r의 값이 "1"이라는 것은 τ_r의 모든 주기에서 여유시간이 존재한다는 의미이며, 이것은 τ_r이 스케줄링이 가능하다는 의미이다. γ_r의 값이 "0"일 때는 τ_r의 여유시간이 존재하지 않는 주기가 적어도 하나 존재한다는 의미로, 이것은 τ_r이 스케줄링이 불가능 하다는 것을 나타낸다.

s_r^k은 τ_r의 k 번째 주기에서의 여유시간을 나타내는 변수로 다음과 같이 구할 수 있다.

$$s_r^k = S_{M_r} - \sum_{j \in \Omega_r^k(1)} l_{1j} \Delta_1 - \sum_{j \in \Omega_r^k(2)} l_{2j} \Delta_2 \cdots - \sum_{j \in \Omega_r^k(r)} l_{rj} \Delta_r \quad (12)$$

여기서 $\Omega_r^k(i)$은 τ_i의 재수행 벡터 $L_i = \left[l_{i1}, \ l_{i2}, \ \cdots \ l_{iv_i} \right]$의 요소 중, τ_r의 k 번째 주기내에 속하는 l_{ij}의 인덱스(index) j 로 이루어진 집합이다.

$$\Omega_r^k(i) = \left\{ j \mid (k-1) \bullet D_r < D_i \bullet j \leq k \bullet D_r, \right\} \quad (13)$$
$$j : \text{자연수}$$

Γ_r은 $\tau_1, \tau_2 \cdots \tau_r$로 이루어진 r개의 태스크의 스케줄링 가능성을 나타낸다. 이것은 $\tau_1, \tau_2 \cdots \tau_{r-1}$로 이루어진 r-1개의 태스크의 스케줄링의 가능성을 판별한 후, r번째 태스크 τ_r의 스케줄링 가능성을 나타내는 값 γ_r을 곱하여 얻을 수 있다. 즉 Γ_r와 Γ_{r-1}사이에는 다음의 관계식이 성립함을 알 수 있다.

$$\Gamma_r = \Gamma_{r-1} \bullet \gamma_r \quad (14)$$

고장이 없는 경우를 상정하고 τ_i보다 우선순위가 낮은 태스크의 스케줄링을 고려하지 않았을 때 τ_i이 가질 수 있는 최대 여유 시간이 S_{M_i}이므로, τ_i의 체크포인터 구간이 Δ_i일 때 최대 여유 시간내에서 재실행 할 수 있는 체크 포인터 구간 수($\bar{l}_i$)는 다음과 같이 구할 수 있다.

$$\bar{l}_i = \left\lfloor \frac{S_{M_i}}{\Delta_i} \right\rfloor \quad (15)$$

즉 $\bar{l}_i$ 는 τ_i보다 우선순위가 낮은 태스크의 스케줄링을 고려하지 않았을 때 τ_i의 고장 극복에 사용할 수 있는 최대 체크 포인터 구간 수를 나타낸다. 따라서 태스크 τ_i의 재실행 구간 수를 나타내는 벡터 L_i의 각 원소(l_{ij})가 가질 수 있는 최대값은 $\bar{l}_i$가 된다. 재실행 벡터 L_i의 각 원소가 가질 수 있는 모든 경우의 수를 나타내기 위해 다음과 같이 정의 하자.

$$\sum_{L_i=0}^{\bar{l}_i} \equiv \sum_{l_{i1}=0}^{\bar{l}_i} \sum_{l_{i2}=0}^{\bar{l}_i} \cdots \sum_{l_{iv_i}=0}^{\bar{l}_i} \quad (16)$$

식 (8), 식(14)와 식(16)을 이용하여 최대 주기(D_m)내의 $\tau_1, \tau_2 \cdots \tau_m$로 이루어진 모든 태스크들이 자신의 주기 이내에서 성공적으로 수행될 확률(P)은 다음과 같이 유도할 수 있다.

$$P = \sum_{L_1=0}^{\bar{l}_1} \sum_{L_2=0}^{\bar{l}_2} \cdots \sum_{L_m=0}^{\bar{l}_m} \psi_1(L_1) \bullet \psi_2(L_2) \bullet \cdots \psi_m(L_m) \bullet \Gamma_m \quad (17)$$

$$\Gamma_1 = 1 \quad (18)$$

확률 P는 각각의 태스크들이 자신의 데드라인 이내에서 성공적으로 수행될 확률의 곱으로 이루어져 있다. 즉 $\tau_1, \tau_2 \cdots \tau_m$의 태스크들이 D_m내에서 성공적으로 수행될 확을 $\psi_1(L_1), \psi_2(L_2), \cdots, \psi_m(L_m)$의 곱으로 되어 있다. 이때 각각의 태스크는 자신에게 주어진 여유시간내에서 고장이 발생한 구간을 재수행 함으로써 고장 극복을 할 수 있다. 이와 같은 각각의 태스크에서 재수행이 일어나는 모든 경우를 고려해야 전체 태스크들이 성공적으로 수행될 확률이 구해진다. 각 태스크가 D_m 주기 내에서 사용 할 수 있는 최대 여유시간은 식(15)와 같이 주어지므로 τ_i의 각 주기에서 재실행 할 수 있는 모든 경우의 수는 식(16)과 같이 나타낼 수 있다. 따라서 $\tau_1, \tau_2 \cdots \tau_m$의 모든 태스크에서 일어 날 수 있는 모든 재실행할 수 있는 경우를 나타내면 $\sum_{L_1=0}^{\bar{l}_1} \sum_{L_2=0}^{\bar{l}_2} \cdots \sum_{L_n=0}^{\bar{l}_m}$ 이 된다. 하지만 $\sum_{L_1=0}^{\bar{l}_1} \sum_{L_2=0}^{\bar{l}_2} \cdots \sum_{L_m=0}^{\bar{l}_m}$가 나타내는 경우는 태스크 $\tau_1, \tau_2 \cdots \tau_m$의 스케줄링이 가능한지를 고려하지 않고 유도된 식이다. $\sum_{L_1=0}^{\bar{l}_1} \sum_{L_2=0}^{\bar{l}_2} \cdots \sum_{L_m=0}^{\bar{l}_m}$가 나타내는 경우 중 태스크의 스케줄링이 가능한 것만을 찾아내기 위하여 Γ_m이 사용된다. Γ_m은 m개의 태스크 모두가 스케줄링 가능할 때 값이 "1"이며, 스케줄링이 안되는 태스크가 존재하면 그 값이 "0"가 된다. 따라서 Γ_m을 사용하여 $\sum_{L_1=0}^{\bar{l}_1} \sum_{L_2=0}^{\bar{l}_2} \cdots \sum_{L_m=0}^{\bar{l}_m} \psi_1(L_1) \bullet \psi_2(L_2) \cdots \psi_m(L_m)$로 주어진 확률 값 중 스케줄링이 가능한 것만을 골라 낼 수 있다. 이렇

게 유도된 식이 식(17)이다. Γ_m은 식(14)에 의해 계산이 가능하다. 또한 우선순위가 가장 높은 태스크 τ_1의 경우 재실행 경우의 수가 $\sum_{L_1=0}^{\bar{l}_1}$ 이므로 τ_1의 스케줄링 가능성을 나타내는 값 Γ_1은 식 (18)과 같이 "1"이 된다.

4.3 최적 체크포인터 구간

최적 체크포인터구간의 선정은 태스크들이 성공적으로 수행될 확률(P) 값을 최대화 시키는 각 태스크에서의 체크 포인터 수를 구하면 된다. 본 논문에서는 각각의 태스크 별로 고유한 체크 포인터 구간을 가지도록 체크 포인터를 삽입한다. 각 태스크별로 삽입 할 수 있는 최대 체크 포인터 수 내의 모든 가능한 체크 포인터를 삽입하였을 때 확률 P를 계산한 후, 확률 P값을 최대로 하는 체크 포인터수가 최적의 체크 포인터 수가된다. 최적 체크포인터 수를 태스크의 수행시간으로 나눈 값이 최적체크포인터 구간이 된다.

각 태스크에 삽입할 수 있는 최대 체크 포인터 수는 식(19)와 같이 유도 할 수 있다.

$$\overline{n_i} = \left\lfloor \frac{D_i - E_i}{t_{cp}} \right\rfloor \tag{19}$$

아래 식(20)과 같이 각 태스크의 체크 포인터 수를 변화시키면서($1 \le n_i \le \overline{n_i}$) 확률 P 값이 최대가 되는 체크 포인터 수를 구하면 이것이 각 태스크에서의 최적 체크 포인터 수($n_1^*, n_2^* \cdots n_i^*$) 가된다.

$$(n_1^*, n_2^* \cdots n_i^*) = \underset{n_1,\cdots,n_m}{Max}\{P\}, \quad 1 \le n_i \le \overline{n_i} \tag{20}$$

따라서 각 태스크에서의 최적 체크포인터 구간은 다음과 같이 구할 수 있다.

$$\Delta_i^* = \frac{E_i}{n_i^*} + t_{cp} \tag{21}$$

5. 시뮬레이션 결과

몇 가지 간단한 예제에 대하여 본 논문에서 유도된 수식을 이용하여 최적의 체크 포인터 수를 구하였다. 태스크의 실행 시간이 $E_1 = 0.4, E_2 = 0.7$ 이고, 주기가 $D_1 = 1, D_2 = 2$인 두 태스크가 스케줄링 되는 경우를 생각하자. 고장 발생율 $\lambda = 0.1$, 체크포인터 오버헤드 $t_{cp} = 0.02$로 가정한다. 그림 7은 태스크 τ_1과 태스크 τ_2에 삽입된 체크포인터 수에 따른 최대 주기 $D_2 = 2$ 내에서 두 태스크의 성공적 수행 확률을 나타낸 것이다. 성공적으로 수행될 확률이 최대가 되는 지점은 $n_1^* = 3, n_2^* = 3$ 인 경우이다. 따라서 태스크 τ_1에는 3개의 체크 포인터를 삽입하고, 태스크 τ_2에도 3개의 체크 포인터를 삽입하는 것이 최적이 된다. 그림 8은 $E_1 = 0.3, E_2 = 0.5$, $D_1 = 1, D_2 = 2$, $\lambda = 0.5$, $t_{cp} = 0.02$ 인 경우의 성공적 수행 확률을 나타낸다. 이 경우 최적 체크 포인터 수는$n_1^* = 4, n_2^* = 6$ 이다.

표 1은 세 태스크가 스케줄링 되는 경우의 결과이다. 각 태스크의 실행 시간이 $E_1 = 0.6, E_2 = 0.5, E_3 = 0.3$ 이고, 주기는 $D_1 = 1, D_2 = 2, D_3 = 4$ 이며, $\lambda = 0.1$, $t_{cp} = 0.02$ 이다. 이 경우 확률 P= 0.75068로 최대로 하는 최적 체크 포인터 수가 $n_1^* = 1, n_2^* = 3, n_3^* = 1$ 임을 보여준다. 체크 포인터 수가 1개인 경우는 태스크의 수행 끝 시점에 체크 포인터를 1개 삽입한다는 것을 의미한다. 표 1에서 확률 값이 "0"인 지점은 스케줄링이 불가능해지는 경우이다.

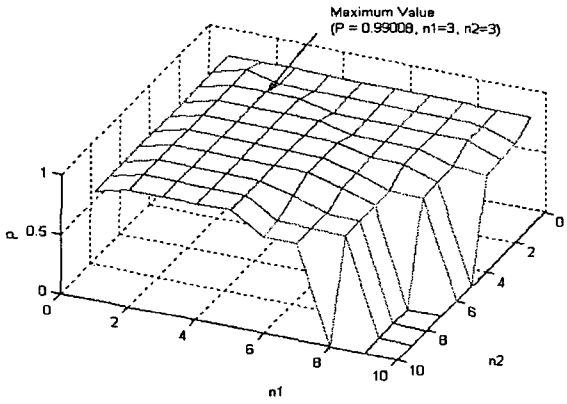

그림 7 체크포인터 수에 따른 태스크의 성공적 수행 확률
Fig. 7 Success probability vs. number of checkpoints
($E_1 = 0.4, E_2 = 0.7, D_1 = 1, D_2 = 2, \lambda = 0.1, t_{cp} = 0.02$)

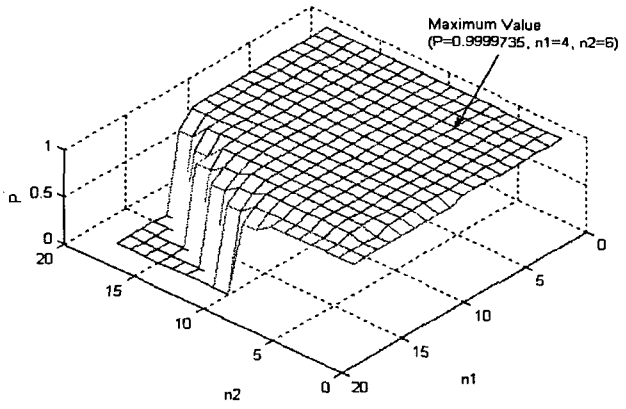

그림 8 체크포인터 수에 따른 태스크의 성공적 수행 확률
Fig. 8 Success probability vs. number of checkpoints
($E_1 = 0.3, E_2 = 0.5, D_1 = 1, D_2 = 2, \lambda = 0.5, t_{cp} = 0.02$)

표 3 체크포인터 수에 따른 태스크의 성공적 수행 확률
Table 3 Success probability vs. number of checkpoints
($E_1 = 0.6, E_2 = 0.5, E_3 = 0.3, D_1 = 1, D_2 = 2, D_3 = 4, \lambda = 0.1,\ t_{cp} = 0.02$)

(n1,n2, n3)	P	(n1,n2, n3)	P	(n1,n2, n3)	P	(n1,n2, n3)	P
(1,1,1)	0.68113	(2,1,1)	0.6757	(3,1,1)	0.67032	(4,1,1)	0
(1,1,2)	0.67977	(2,1,2)	0.67435	(3,1,2)	0	(4,1,2)	0
(1,1,3)	0.70269	(2,1,3)	0.67301	(3,1,3)	0	(4,1,3)	0
(1,1,4)	0.70266	(2,1,4)	0.67166	(3,1,4)	0	(4,1,4)	0
(1,2,1)	0.67841	(2,2,1)	0.67301	(3,2,1)	0	(4,2,1)	0
(1,2,2)	0.67706	(2,2,2)	0.67166	(3,2,2)	0	(4,2,2)	0
(1,2,3)	0.6757	(2,2,3)	0	(3,2,3)	0	(4,2,3)	0
(1,2,4)	0.67435	(2,2,4)	0	(3,2,4)	0	(4,2,4)	0
(1,3,1)	0.75068	(2,3,1)	0	(3,3,1)	0	(4,3,1)	0
(1,3,2)	0.74918	(2,3,2)	0	(3,3,2)	0	(4,3,2)	0
(1,3,3)	0.74768	(2,3,3)	0	(3,3,3)	0	(4,3,3)	0
(1,3,4)	0.74619	(2,3,4)	0	(3,3,4)	0	(4,3,4)	0
(1,4,1)	0.75051	(2,4,1)	0	(3,4,1)	0	(4,4,1)	0
(1,4,2)	0.74901	(2,4,2)	0	(3,4,2)	0	(4,4,2)	0
(1,4,3)	0.74752	(2,4,3)	0	(3,4,3)	0	(4,4,3)	0
(1,4,4)	0	(2,4,4)	0	(3,4,4)	0	(4,4,4)	0

6. 결 론

서로 다른 주기를 가지는 여러 실시간 태스크가 스케줄링 되어 실행되는 시스템에서 체크 포인터를 삽입하는 방식에 대한 연구는 지금까지 거의 없었다. 이것은 여러 태스크들이 스케줄링 됨으로써 발생하는 복잡성 때문에 이 문제를 해석적 방법으로 접근하기가 쉽지 않았기 때문이다. 본 논문에서는 실시간 태스크들이 RM(Rate Monotonic) 알고리즘에 의해 스케줄링 되고, 태스크의 주기가 데드라인과 동일하다는 가정하에 최적의 체크 포인터 삽입 방법을 연구하였다. 또한 수학적 복잡성을 피하기 위해 각 태스크의 주기는 최소 주기의 2의 승수 배로 이루어진 경우로 한정하였다. Poisson 분포를 가지는 과도 고장이 발생하는 상황에서, 모든 실시간 태스크들이 자신의 주기이내에서 성공적으로 수행될 확률식을 유도 하였다. 이 확률 식은 각 태스크들이 가질 수 있는 최대 여유 시간, 태스크들에서 고장 극복에 사용되는 시간, 고장 상황에서 모든 태스크들의 스케줄링 가능성을 체크하는 식, 고장 상황에서 각 태스크가 자신의 주기 내에서 수행을 성공적으로 마칠 확률식들로 부터 유도되었다. 이 확률 식을 최대로 하는 값이 각 태스크에서 사용되는 최적의 체크 포인터 구간이 된다. 몇 가지 예들에 대한 시뮬레이션 결과로부터 본 논문에서 제시한 방법의 유용성을 살펴보았다.

감사의 글

이 논문은 2006년도 정부재원(교육인적자원부 학술연구조성사업비)으로 한국학술진흥재단의 지원을 받아 연구되었음(KRF-2006-003-D00179)

참 고 문 헌

[1] C. M. Krishna and A. D. Singh, "Optimal configuration of redundant real-time systems in the face of correlated failure," IEEE Trans. on Reliability, vol. 44, pp. 587-594. Dec.1995.

[2] Seong Woo Kwak and Byung Kook Kim, "Task Scheduling Strategies for Reliable TMR Controllers using Task Grouping and Assignment", IEEE Trans. on Reliability, vol. 49, no.4, pp. 355-362, Dec. 2000.

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

[4] D. P. Siewiorek and R. S. Swarz, Reliable Computer Systems, Digital Press, 1992.

[5] Avi Ziv and Jehoshua Bruck, "An on-line algorithm for checkpoint placement," IEEE Trans. on Computers, vol. 46, pp. 976-984, Sep. 1997.

[6] R. Geist, R. Reynolds, and J. Westall, "Selection of a checkpoint interval in a critical-task environment," IEEE Trans. on Reliability, vol. 37, pp. 395-400, Oct. 1988.

[7] Kang G. Shin, Tein-Hsiang Lin, and Yann-Hang Lee, "Optimal checkpointing of real-time tasks," IEEE Trans. on Computers, vol. C-36, pp. 1328-1341, Nov. 1987.

[8] C. M. Krishna and A. D. Singh, "Reliability of checkpointed real-time systems using time redundancy," IEEE Trans. on Reliability, vol. 42, pp. 427-435, Sep. 1993.

[9] Seong Woo Kwak, Byung Jae Choi and Byung Kook Kim, "Optimal Checkpointing Strategy for Real-Time Control Systems under Faults with Exponential Duration", IEEE Trans. on Reliability, vol.50, no.3, pp. 293-301, Sep. 2001.

[10] Seong Woo, Kwak, "Reliability Analysis and Design of Real-time Fault Tolerant Control Systems under Transient Faults", Ph.D thesis, KAIST, 2000.

[11] John W. Young, "A first order approximation to the optimal checkpoint intervals," Comm. of the ACM, vol. 17, pp.530-531, Nov. 1974.

[12] Hagbae Kim and Kang G. Shin, "Modeling of externally-induced/common-cause faults in fault-tolerant systems," IEEE/AIAA Digital Avionics System Conference, pp. 402-407, Oct. 1994.

[13] Sunondo Ghosh,Rami G. Melhem, Daniel Mosse," Fault-Tolerance Through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems", IEEE Trans. on Parallel Disgrib. Syst, Vol.8, No. 3 pp. 272-284, 1997.

[14] Frank Liberato, Rami Melhem, Daniel Mosse, "Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems", vol49, No.9, IEEE Trans. on Computers, pp.906-914, sep. 2000.

[15] H. Kim and K. G. Shin, "Design and Analysis of an Optimal Instruction Retry Policy for TMR Controller Computers", IEEE Trans. on Computers, vol 45, pp. 1217-1225, Nov. 1996.
[16] 곽성우, 하드데드라인을 가지는 다중 실시간 주기적 태스크에서의 체크포인팅 기법, 전기학회논문지-D, 제 53권 제8호, pp. 594-601, 2004. 8

저 자 소 개

곽 성 우 (郭 成 祐)

1970년 3월 104일생. 1993년 한국과학기술원 전기및전자공학과 졸업(학사) 1995년 동 대학원 전기및전자공학과 졸업(석사). 2000년 동 대학원 전기및전자공학과 졸업(공박). 2000년~2002년 인공위성연구센터 선임연구원, 연구교수. 2003년~현재 계명대 전자공학과 전임강사, 조교수
Tel : 053-580-5926
Fax : 053-580-5165
E-mail : ksw@kmu.ac.kr

정 용 주 (鄭 容 朱)

1965년 5월 24일생. 1988년 서울대학교 전자공학과 졸업(학사) 1990년 한국과학기술원 전기및전자공학과 졸업(석사). 1995년 동 대학원 전기및전자공학과 졸업(공박). 1995년~1998년 LG전자 선임연구원. 1999년~현재 계명대 전자공학과 전임강사, 조교수, 부교수
Tel : 053 - 580 - 5925
Fax : 053 - 580-5165
E-mail : yjjung@kmu.ac.kr