논문 2007-44TC-7-17

중복구조 실시간 시스템에서의 고장 극복 및 최적 체크포인팅 기법

(Fault Recovery and Optimal Checkpointing Strategy for Dual Modular Redundancy Real-time Systems)

곽 성 우*

(Seong Woo Kwak)

요 약

본 논문에서는 중복 구조 시스템을 이용하여 각 프로세서에서의 출력을 비교하여 효율적으로 고장을 탐지하고, 체크포인팅 기법을 적용하여 과도 고장뿐 아니라 영구적 고장을 극복하기 위한 방법을 제안한다. 매 체크포인터에서는 각 프로세서로 부터의 출력과 과거 체크포인터에 저장된 데이터를 불러와 서로 비교한 후 과거 체크포인터로 회귀할지 태스크의 수행을 계속 수행할지 결정한다. 과도 고장과 영구 고장이 발생할 수 있는 상황에서 제안된 체크포인팅 기법을 탑재한 중복 구조 시스템을 마코프 모델을 이용하여 모델링한다. 마코프 모델로부터 실시간 태스크가 데드라인 이내에서 성공적으로 수행을 끝낼 확률을 계산하고, 이 확률식을 이용하여 중복구조 시스템에 탑재할 체크포인터 구간을 최적화한다. 최적화된 체크포인터 구간은 태스크의 성공적 수행 확률을 최대화 하도록 선정하였다.

Abstract

In this paper, we propose a new checkpointing strategy for dual modular redundancy real-time systems. For every checkpoints the execution results from two processors, and the result saved in the previous checkpoint are compared to detect faults. We devised an operation algorithm in checkpoints to recover from transient faults as well as permanent faults. We also develop a Markov model for the optimization of the proposed checkpointing strategy. The probability of successful task execution within its deadline is derived from the Markov model. The optimal number of checkpoints is the checkpoints which makes the successful probability maximum.

Keywords : Checkpoint, Dual Modular Redundancy, Real-time Task, Fault Recovery

I. 서 론

실시간 제어 시스템의 고장이라 함은 하드웨어 자체의 결함(permanent fault)에 의한 영구적 고장, 외부나 내부의 요인에 의한 일시적인 오동작(transient fault)뿐만 아니라 실시간이라는 시간 조건을 만족하지 않는 것을 포함해야한다. 이와 같은 고장에 대처하기 위해서 대부분의 실시간 제어 시스템은 내고장성(fault tolerance)을 가지도록 설계된다. 최근 하드웨어 기술과 소프트웨어기술의 비약적인 발전으로 인하여 과거에는 불가능한 것으로 여겨졌던 많은 고장 극복 기법들을 프로세서에 탑재 할 수 있게 되었다.

실시간 시스템의 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 기법과 같은 소프트웨어적인 방

* 정회원, 계명대학교 전자공학과
(Dept. of EE, Keimyung University)
※ 본 연구는 2006년도 계명대학교 비사연구기금으로 이루어졌음
접수일자: 2007년4월23일, 수정완료일: 2007년7월2일

법들은 부가적인 하드웨어뿐만 아니라 시간적인 오버헤드(overhead)를 요구하여 실시간 시스템에 사용하는 것이 부적절한 것으로 지금까지 여겨져 왔다. 하지만 최근 컴퓨터의 처리 성능 향상과 OS(Operating System) 기술의 발전으로 체크 포인팅 기법을 실제 실시간 시스템에 적용하는 것이 가능하여졌다. 특히 산업 현장에서 발생하는 시스템 고장들을 분석한 결과에 따르면 소자의 영구적 결함과 같은 하드웨어적인 원인과 함께 외부의 전기적, 기계적 환경변화등과 같은 과도 고장에 의한 것이 다수를 이루고 있는 것으로 보고되고 있다[1]. 따라서 영구적 고장을 극복하기 위한 하드웨어적 중복 구조뿐 아니라, 과도 고장을 극복할 수 있는 체크포인팅 기법과 같은 소프트웨어적 기법이 동시에 탑재되는 것이 효율적이다.

본 논문에서는 중복 구조 시스템을 이용하여 각 프로세서에서의 출력을 비교하여 효율적으로 고장을 탐지하고, 체크포인팅 기법을 적용하여 과도 고장뿐 아니라 영구적 고장을 극복하기 위한 방법을 제안한다. 뿐만 아니라 실시간 태스크가 데드라인 이내에서 수행을 끝낼 확률을 최대화 시키는 최적의 체크포인터 구간을 찾는 방법을 제시한다.

체크 포인팅 기법을 적용한 실시간 시스템을 해석하여 최적의 체크 포인팅 방법을 구하려는 연구는 기존의 다수 연구자들에 의해 연구 되어왔다[2~7]. 본 연구자도 이러한 연구를 수행 한바 있다[8~11]. 실시간 시스템에서 체크포인팅 기법과 관련된 기존의 연구들은 다음과 같다. Geist et. al(1988)는 태스크의 수행 시간을 최소화하는 체크 포인터 삽입 방법을 연구 하였고[4], Shin et. al(1987)은 평균 수행 시간을 최소화 시키는 체크 포인터 삽입 방법을 구하였다[5]. Krishna & Singh(1995)은 지속 시간이 존재하는 고장의 상태를 상정하여, 체크포인터 기법을 탑재한 시스템이 이중화(duplex) 또는 삼중화(triplex) 구조 중 어떤 구성이 더 효율적인지를 해석하였다[2].

하지만 기존의 다른 연구자들에 의한 연구들은 고장 발생시 발생 시점에 고장을 탐지할 수 있다는 가정을 하고 체크포인팅 기법을 연구하였다. 본 연구에서는 고장 탐지를 위해 중복 구조 시스템을 이용하여 두 프로세서에서의 수행 결과를 비교한다. 따라서 고장의 발생 시점에 고장을 바로 탐지하는 것이 아니라 두 프로세서의 수행 결과를 비교하는 체크포인팅 시점에 고장을 탐지하고, 고장 발생시 최근의 체크포인터로 회귀하도록 하여 고장을 극복하는 방식을 취한다. 또한 중복 구조 시스템 중 한쪽 프로세서에 영구 고장이 발생한 경우에도 하나의 프로세서만으로도 고장을 탐지하고 체크포인팅 기법을 운용할 수 있도록 하였다. 제안된 방식으로 체크포인팅 기법을 운용하였을 때 최적의 체크포인팅 구간을 어떻게 선정하여야 하는지를 마코프(Markov) 모델을 이용한 확률 해석 방법을 사용하여 구하였다. 최적의 체크포인터 구간은 실시간 태스크가 데드라인 이내에서 성공적으로 수행을 끝낼 확률을 최대로 하는 값으로 선정하였다.

Ⅱ. 중복구조 체크포인팅 시스템에서의 고장 탐지 및 극복

전형적인 중복 구조 시스템에서는 두 프로세서의 출력을 서로 비교하여 동일하면 고장이 없고, 동일하지 않으면 고장이 발생한 것으로 판단하여 고장을 탐지한다. 따라서 전통적인 중복 구조 시스템에서는 고장을 탐지할 수는 있으나 고장을 극복하기는 어렵다. 본 논문에서는 그림 1.와 같이 중복 구조 시스템에 체크포인팅 기법을 적용하고, 이를 운용하기 위한 새로운 방법을 제안하여 과도 고장뿐 아니라 두 프로세서중 하나에 영구적인 고장이 발생한 경우에도 타당한 출력을 내도록 하고자 한다. 특히 중복 구조 시스템에서 실행되는 태스크는 주어진 데드라인 이내에 반드시 수행을 마쳐야 하는 실시간 태스크로 한정하여 최적의 체크포인터 삽입 구간을 찾는 방법을 제안한다. 본 논문에서는 체크포인팅과 관련하여 다음과 같이 가정한다.

기본가정:

A1. 체크포인터에 저장된 데이터의 비교로 체크포인터 구간에서 발생한 고장을 완벽히 탐지 할 수 있다.

A2. 두 프로세서 간 전송되는 데이터에는 checksum 과 같은 오류 검출 및 보정코드를 사용하여 전송 에러가 없다.

A3. 오류보정 코드 등을 사용하여 체크포인팅 과정에서는 오류가 없다.

다음은 중복구조 시스템에서 제안된 체크포인터 운용 방법이다. 한쪽 프로세서에서 고장이 발생하면 기본적으로 최근의 체크포인터로 회귀하지만, 한쪽 프로세서에서 연속된 두 체크포인터 구간에 고장이 없으면 체크포인터로 회귀하지 않고 다음 구간으로 진행하도록 설계되었다. 여기서 태스크의 상태 변수란 CPU의 레지스터 값 즉 콘텍스트(context)를 의미한다. 따라서 과거

체크포인터에 저장된 콘텍스트 값을 CPU에 로드하면 과거 상태로 회귀할 수 있다.

제안된 체크포인팅 운용 기법

Step 1. 매 체크포인터에서 가장최근의 체크포인터에 저장된 태스크 상태 변수들과 현재 저장할 상태 변수들을 비교한다.

Step 2. 과거 상태 변수들과 현재 저장될 상태 변수들이 서로 일치하는 경우, 이 변수들을 현재 체크포인터에 저장하고 과거 데이터와 일치했다는 플래그(Flag)와 함께 상태 변수들을 다른 쪽 프로세서에 전송한 후 태스크 수행을 계속한다.

Step 3. 과거 상태변수들과 현재 저장될 상태변수들이 일치하지 않은 경우, 현재 변수들을 다른 쪽 프로세서에 전송하고 다른 쪽 프로세서로부터 전송된 상태 변수와 현재 저장될 상태 변수를 비교한다.

Step 4. 다른 프로세서로부터 전송된 상태변수와 현재 저장될 상태변수가 일치하는 경우 이것을 저장하고 태스크 수행을 계속한다.

Step 5. 전송된 상태변수와 현재 저장될 상태변수가 일치하지 않지만 전송된 상태변수에 일치 플래그(Flag)가 있으면 전송된 상태변수를 저장하고 전송된 상태 변수를 CPU에 로드하여 태스크 수행을 계속한다.

Step 6. 전송된 상태변수와 현재 저장될 상태변수가 일치하지 않는 경우 최근의 체크포인터에 저장된 과거의 상태 변수를 CPU에 로드(load)하고 체크포인터로 회귀하여 태스크를 재수행 한다.

태스크에 n개의 체크포인터를 삽입한 경우, n개의 체크포인터 구간을 데드라인 이내에서 고장 없이 수행하면 한주기의 태스크 수행이 끝난다. 태스크 수행 중 고장이 발생하면 태스크의 데드라인 이내에서 사용가능한 여유시간(slack time)을 이용하여 고장이 발생한 체크포인터 구간을 재수행 함으로써 고장을 극복한다. 따라서 여유시간에서 재실행 할 수 있는 체크포인터 구간수는 태스크의 성공적인 수행 확률과 밀접한 관계가 있다. 본 논문에서는 어떻게 체크포인터 구간을 선정하는 것이 데드라인이내에서 태스크의 성공적 수행 확률을 최대로 하는지 찾고자 한다. 수행 시간이 E 인 태스크에 n개의 체크포인터가 삽입된 경우 각 체크포인터 구간은 다음과 같이 구할 수 있다.

$$\Delta = \frac{E}{n} + t_{cp} \tag{1}$$

여기서 t_{cp}는 체크포인팅 오버헤드(overhead)로 다음과 같은 작업을 수행하는데 걸리는 시간이다.

① 가장최근의 체크포인터 저장된 데이터와 현재 저장될 데이터를 비교한다.

② 다른 쪽 프로세서로 현재 저장될 데이터와 비교결과를 전송한다.

③ 다른 쪽 프로세서로부터 전송된 데이터와 현재 데이터를 비교한다.

④ 1번, 3번 데이터 비교에서 최소한 1회 이상 오류가 없을 경우 현재 데이터를 저장한다.

⑤ 데이터 비교에서 모두 오류가 있을 경우 가장 최근의 체크 포인터에 저장된 데이터를 CPU에 로드 한다.

태스크에 체크포인터가 n개 삽입된 후 태스크의 실행 시간 e는 다음과 같다.

$$e = n\Delta = n \cdot \left(\frac{E}{n} + t_{cp}\right) = E + nt_{cp} \tag{2}$$

즉 태스크의 원래 실행 시간에 체크 포인팅을 하는데 걸리는 시간이 더해져 실행 시간이 nt_{cp} 만큼 늘어난다. 태스크의 데드라인을 D 라고 하면 태스크의 여유시간(slack time)은 다음과 같이 구할 수 있다.

$$S = D - e \tag{3}$$

그림 1.은 중복구조 시스템에서 본 논문에서 제시된 방법에 따라 체크포인팅 기법을 운용하는 예를 나타낸다. 태스크는 n개의 체크포인터 구간(블록)으로 구성된다. n개의 체크포인터가 삽입된 태스크의 n개 블록을 각각 $\tau_1, \tau_2, \cdots \tau_n$으로 표시하자. 프로세서-2의 두 번째 시간 슬롯(slot)($\Delta \sim 2\Delta$ 시간)에서 고장이 발생한 경우 두 프로세서는 τ_2를 세 번째 시간 슬롯($2\Delta \sim 3\Delta$)에서 재수행 한다. 시간 $2\Delta \sim 3\Delta$에서 프로세서-1에는 고장이 없고 프로세서-2에서 다시 고장이 발생한다. 프로세서-1에서는 2Δ 시각에 저장된 데이터와 3Δ 시각에 저장될 데이터를 비교한다. 두 구간에서 고장이 없으므로 두 데이터가 동일하고 오류가 없다. 프로세서-1은 프로세서-2로 오류가 없는 완벽한 데이터라는 플래그(flag)와 함께 데이터를 전송한다. 프로세서-2에서 프로세서-1로 전송된 데이터와도 비교한다. 프로세서-1에

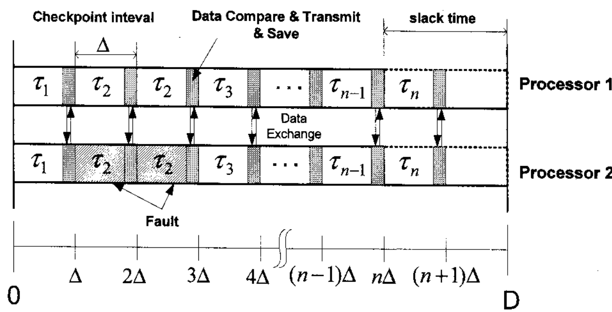


그림 1. 중복구조 체크포인팅 시스템에서의 고장 탐지 및 극복

Fig. 1. Fault detection and recovery in the dual modular redundancy checkpointing system.

서는 두 번의 데이터 비교 중 첫 번째 비교에서 오류가 없으므로 이것을 $3\triangle$ 시각에 저장한다. 프로세서-2는 프로세서-1로부터 전송된 데이터와 자신의 데이터를 비교한다. 또한 $2\triangle$ 에 저장된 데이터와도 비교한다. 시간 슬롯 2와 슬롯 3에서 고장이 발생했기 때문에 두 번의 비교에서 모두 오류가 있음이 발견된다. 데이터 비교에서 오류가 있는 것이 발견되어 다시 최근의 체크포인터로 회귀하여야 하지만, 프로세서-1로 부터 전송된 데이터에 오류가 없다는 플래그가 있으므로 프로세서-1로 부터 전송된 데이터를 저장하고 다음 태스크 블록(τ_3)으로 넘어간다. 따라서 그림 1.에서와 같이 τ_2가 두 번 수행된 후 두 프로세서 모두 τ_3로 진행한다.

모든 태스크 블록에서 오류가 없는 경우 시각 $n\triangle$ 에서 태스크의 수행이 끝나지만 그림 1.와 같이 블록 2에서 오류가 발생하여 τ_2를 재수행 했기 때문에 $(n+1)\triangle$ 시각에서 태스크의 수행이 끝난다. 즉 원래 태스크의 여유시간 중 $\triangle$ 시간만큼을 고장 극복에 사용하였다.

그림 1.에서 프로세서-2에 영구 고장(permanent fault)이 발생한 경우 프로세서-2의 수행이 멈춰 데이터를 전송하지 않거나, 수행이 계속되어 데이터를 전송할 수 있는 경우에도 전송된 데이터에는 오류가 항상 존재한다. 프로세서-1에서는 프로세서-2에 영구고장이 발생한 이후부터 프로세서-2로부터 데이터가 전송되지 않거나, 전송된 데이터와 프로세서-1의 데이터를 비교하면 오류가 검출 된다. 따라서 제안된 체크포인팅 기법에 따라 체크포인팅 기법을 운영하면 프로세서-1 에서는 프로세서-2에 영구고장이 발생한 이후부터 태스크 블록들을 항상 2번씩 수행한다. 여유시간만 충분하다면 프로세서-2에서 영구고장이 발생한 경우에도 각 태스크 블록들을 2번씩 수행함으로써 태스크의 수행을 데드라인 이내에서 완전히 끝낼 수 있다. 즉 영구 고장을 극복할 수 있다.

Ⅲ. 중복 구조시스템 모델링

3.1 과도 고장만 있는 경우

과도 고장이 고장 발생율 λ를 가진 Poisson 분포에 따라 발생한다고 할 때, t 시간내에 n개의 고장이 발생할 확률은 다음과 같다.

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

따라서 $\triangle$ 시간 구간에 고장이 없을 확률은

$$p = \alpha_0(\lambda, \triangle) = e^{-\lambda\triangle} \tag{5}$$

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

$$q = \sum_{n=1}^{\infty} \frac{(\lambda\triangle)^n}{n!} e^{-\lambda\triangle} = 1 - e^{-\lambda\triangle} \tag{6}$$

본 논문에서는 고장의 발생과 관련하여 다음과 같이 가정한다.

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

l을 태스크의 데드라인 이내에 들어 갈 수 있는 체크포인터 수로 정의하면 다음과 같이 구할 수 있다.

$$l = \left\lfloor \frac{D}{\triangle} \right\rfloor \tag{7}$$

l개의 체크포인터 구간 중 n개의 구간에서 고장이 없으면 태스크는 성공적으로 수행이 끝난다.

과도 고장이 있는 경우에 체크포인팅 기법을 탑재한 중복 구조 시스템을 마코프 체인(Markov Chain)으로 모델링하기 위해 그림 1.에서와 같은 n 블록으로 이루어진 태스크의 수행을 그림 2.와 같이 $2n+1$ 개의 상태(state)로 나타내자. 각각의 태스크 블록을 2개의 상태로 나타내고 마지막 n번째 블록 이후를 1개의 상태로 나타낸다. 하나의 상태에서 다음 상태로의 천이(transition)는 각 블록의 끝(즉 매 체크포인터의 끝)에서 일어나도록 한다. 그림 2.는 과도고장만 존재하는 경우의 마코프 모델을 나타낸다. 각각의 상태는 두 개의 인덱스(index) (i,j)로 구성된다. 첫 번째 인덱스 "i"는

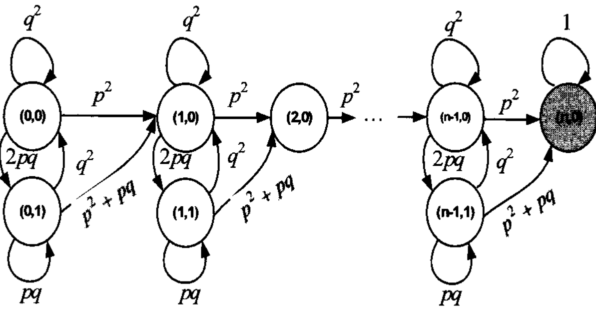


그림 2. 과도 고장하의 중복 구조 체크포인팅 시스템의 마코프 모델

Fig. 2. Markov model of the dual modular redundancy checkpointing system under transient faults.

고장 없이 성공적으로 수행된 태스크 블록수를 나타내며, 두 번째 인덱스 "j"는 두 프로세서에서 태스크 블록의 수행 상태를 나타낸다. 예를 들어 $(1,j)$는 성공한 태스크 블록수가 1개(τ_1이 성공적으로 수행됨)라는 것을 나타내고, $(n,0)$는 성공한 블록수가 n개 ($\tau_1, \tau_2, \cdots \tau_n$이 성공적으로 수행됨)라는 것을 나타낸다. 따라서 $(n,0)$ 상태는 모든 태스크 블록이 수행된 상태를 나타낸다.

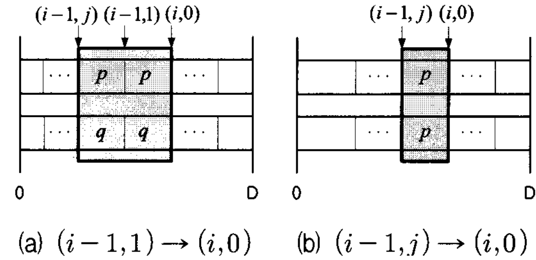
태스크 블록의 수행 상태를 나타내는 "j"는 2개로 나눈다. 다음은 상태 $(i,0)$, $(i,1)$에 대한 정의이다.

상태 $(i,0)$: $(i+1)$번째 블록이 처음 수행되는 상태, 또는 두 프로세서 모두에 고장이 발생하여 $(i+1)$번째 블록이 재수행 돼야 하는 상태.

상태 $(i,1)$: 한쪽 프로세서에서는 고장이 발생하고 다른 쪽 프로세서에서는 고장이 발생하지 않아서 $(i+1)$번째 블록을 다음 시간 슬롯에서도 계속 수행해야 되는 상태.

그림 3.와 그림 4.는 상태 $(i,0)$로의 천이를 그림으로 보여준다. 그림 3.은 i번째 태스크 블록이 성공적으로 수행되어 $(i+1)$번째 블록이 처음 수행되는 상태를 나타내고, 그림 4.는 두 프로세서에 고장이 발생하여 $(i+1)$번째 블록을 다시 수행해야 하는 상태를 나타낸다.

그림 5.는 상태 $(i,1)$로의 천이를 그림으로 보여준다. 그림 3. (a)에서와 같이 한쪽 프로세서에서 연속으로 두 시간 슬롯에 고장이 없는 경우는 제외하고, 한쪽 프로세서에서는 고장이 발생하고 다른 쪽 프로세서에서는 고장이 발생하지 않아서 $(i+1)$번째 블록을 다음 시간 슬롯에서도 계속 수행해야 되는 상태가 $(i,1)$이다. 즉 상태 $(i,1)$는 각 프로세서에서 연속으로 고장이 없는



(a) $(i-1,1) \rightarrow (i,0)$ (b) $(i-1,j) \rightarrow (i,0)$

그림 3. $(i+1)$번째 태스크 블록이 처음 수행될 상태 ($(i-1,j) \rightarrow (i,0)$ 로의 천이)

Fig. 3. State Transition of $(i-1,j) \rightarrow (i,0)$.

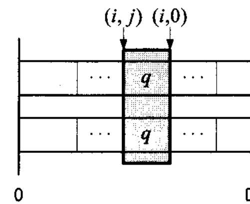
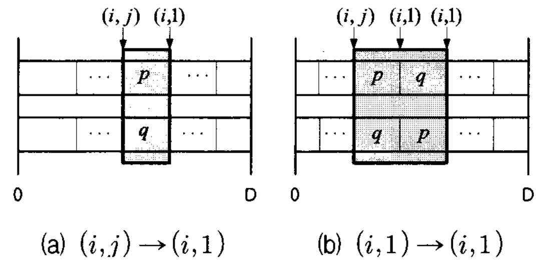


그림 4. 고장에 의해 $(i+1)$번째 태스크 블록이 재수행 되어야 하는 상태($(i,j) \rightarrow (i,0)$ 로의 천이)

Fig. 4. State Transition of $(i,j) \rightarrow (i,0)$



(a) $(i,j) \rightarrow (i,1)$ (b) $(i,1) \rightarrow (i,1)$

그림 5. $(i+1)$번째 블록을 다음 시간 슬롯에서도 계속 수행해야 되는 상태. ($(i,j) \rightarrow (i,1)$ 로의 천이)

Fig. 5. State Transition of $(i,j) \rightarrow (i,1)$.

시간 슬롯이 존재하는 경우를 제외하고 한쪽은 고장이 있고 다른 쪽은 고장이 없는 경우를 나타낸다.

초기 상태가 $(0,0)$일 때 그림 2.의 마코프 모델에서 l 스텝 후 상태 $(n,0)$에 있을 확률을 구하면 태스크가 데드라인 이내에서 성공적으로 수행을 마칠 확률이 된다. 마코프 모델의 1스텝은 1개의 시간 슬롯을 의미하고, 식 (7)에 의해 데드라인 내에 l개의 시간 슬롯이 존재하며 이들 슬롯에서 n개의 태스크 블록이 성공적으로 수행되면 태스크의 수행이 완료되기 때문이다.

l 스텝 후 각 상태에 있을 확률을 구하기 위해 다음과 같이 정의하자.

$c_{ij}(k)$: k 번째 스텝에서 상태 (i,j)에 있을 확률

$C_i(k)$: k 번째 스텝에서 각 상태에 있을 확률을 나타

내는 칼럼 벡터(column vector)

즉, $C_i(k) = \begin{bmatrix} c_{i0}(k) \\ c_{i1}(k) \end{bmatrix}$

$2n+1$ 상태를 가진 마코프 체인의 k 스텝 후의 상태를 구하기 위해서는 $(2n+1)\times(2n+1)$크기의 상태 천이 행렬(state transition matrix)를 구한 후, 상태 천이 방정식을 풀어야 된다. 하지만 그림 2.의 마코프 모델을 살펴보면 각 상태 (i,j)에서 천이할 수 있는 다음 상태는 $(i,j)\rightarrow(i+1,0)$, $(i,j)\rightarrow(i,j')$ 또는 $(i,j)\rightarrow(i,j)$ 뿐이다. 즉 인덱스 i가 1만큼 증가되는 상태($i+1$)로 천이하던지, 인덱스 i는 그대로이고 j만 바뀌는 상태로 천이하거나 현재 상태를 그대로 유지한다. 따라서 k 스텝에서 각 상태에 있을 확률은 다음의 식으로 구할 수 있다.

$$
\begin{aligned}
C_0(k) &= M_p \cdot C_0(k-1) \\
C_1(k) &= M_p \cdot C_1(k-1) + M_n \cdot C_0(k-1) \\
&\vdots \\
C_i(k) &= M_p \cdot C_i(k-1) + M_n \cdot C_{i-1}(k-1) \\
&\vdots \\
C_{n-1}(k) &= M_p \cdot C_{n-1}(k-1) + M_n \cdot C_{n-2}(k-1) \\
C_n(k) &= C_n(k-1) + M_n \cdot C_{n-1}(k-1)
\end{aligned}
\quad (8)
$$

여기서,

$$M_p = \begin{bmatrix} q^2 & q^2 \\ 2pq & pq \end{bmatrix},\ M_n = \begin{bmatrix} p^2 & p^2+pq \\ 0 & 0 \end{bmatrix} \quad (9)$$

식 (8)의 k 스텝에서 각 상태에 있을 확률($C_i(k)$)은 두 항의 합으로 이루어져 있다. 인덱스 "i"를 그대로 유지하는 $(i,j)\rightarrow(i,j')$ 또는 $(i,j)\rightarrow(i,j)$ 로 천이하는 항($M_p \cdot C_i(k-1)$)과 인덱스 "i"가 "1" 증가되는 $(i-1,j)\rightarrow(i,j')$로 천이 하는 항($M_n \cdot C_{i-1}(k-1)$)의 합이다. M_p 는 상태 (i,j)에서 인덱스 i가 동일하면서 다음 상태로 천이하는 확률 천이 행렬이다. 이것은 현재 상태에서 j 만 바뀌는 상태 또는 현재 상태를 그대로 유지하는 천이를 나타내는 것으로, $(i,j)\rightarrow(i,j')$ 또는 $(i,j)\rightarrow(i,j)$로의 천이를 나타내는 행렬이다. M_n은 인덱스 i가 1 증가하면서 다음 상태로 천이하는 확률 천이 행렬이다. 즉 $(i-1,j)\rightarrow(i,j')$로 천이할 때의 상태 천이 행렬이다. 그림 2.로부터 M_p와 M_n는 식 (9)와 같이 구할 수 있다.

l 스텝 후에 $(n,0)$에 있을 확률은 그림 1.의 체크포인팅 기법을 탑재한 중복 구조 시스템이 제안된 방법으로 운용됐을 때 태스크를 데드라인 이내에 성공적으로 수행할 확률이 된다. 이것은 식 (8)을 풀어서 구할 수 있다. 따라서 식 (8)로 부터 l 스텝 후에 $(n,0)$에 있을 확률을 구한 후 이것을 최대로 하는 체크포인터 구간을 구하면 그림 1.의 중복 구조 체크포인팅 시스템을 최적화하는 것이 된다.

3.2 과도 고장과 영구 고장이 동시에 존재하는 경우

영구 고장은 과도 고장과는 달리 프로세서의 일정 부분에 영구적인 손상이 발생한 것으로 고장 발생 후 계속 존속하며 하드웨어를 교체하거나 영구적 고장이 발생한 부분을 고립시키는 방법으로 극복한다. 즉 본 논문에서 사용하는 과도 고장 극복 방법인 체크포인터 삽입과 회귀(rollback)로는 영구고장을 극복할 수 없다. 하지만 그림 1.의 중복 구조 시스템에서 한쪽 프로세서에서 영구적 고장이 발생한 경우에도 제안된 방법으로 체크포인팅 기법을 운용하면 여유 시간이 충분한 경우 태스크의 성공적 실행을 보장할 수 있다. 즉 제안된 체크포인팅 운영 방식은 과도 고장뿐 아니라 영구고장이 발생한 경우에도 적용할 수 있다. 영구적 고장이 존재하는 경우를 고려하기 위해 $(i,2)$, $(i,3)$의 두 가지 상태를 다음과 같이 정의한다.

상태 $(i,2)$: 한쪽 프로세서에서 영구적 고장이 발생하고 태스크의 $(i+1)$번째 블록이 처음 수행되는 상태, 또는 한쪽은 영구적 고장이 다른 쪽 프로세서에 과도 고장이 발생하여 $(i+1)$번째 블록을 재수행 해야 되는 상태.

상태 $(i,3)$: 한쪽 프로세서에서 영구적 고장이 발생하였지만 다른 쪽 프로세서에서는 과도 고장이 발생하지 않아서 $(i+1)$번째 블록을 다음 시간 슬롯에서도 계속 수행해야 되는 상태. 연속된 2 시간 슬롯 모두에서 고장이 없으면 $(i+1,1)$로 천이함.

여기서 각 상태의 첫 번째 인덱스 i 는 과도 고장 때의 정의와 같이 성공적으로 수행된 태스크 블록 수를 나타낸다.

그림 6.은 상태 $(i,j)\rightarrow(i,2)$로의 천이를 보여준다. 그림 6.은 영구적 고장이 두 프로세서 중 한 곳에 발생하고, 영구적 고장이 발생한 시간 슬롯에 다른 쪽 프로세서에서는 과도 고장이 발생하여$(i,2)$로 천이하는 경우이다. 그림 7.은 한 프로세서에서는 영구 고장이 발생하였지만 다른 쪽 프로세서에서 연속된 2개의 시간 슬롯에 고장이 없어 $(i+1)$번째 블록이 처음 수행되는 상태를 나타낸다. 연속된 두 시간 슬롯에 고장이 없기

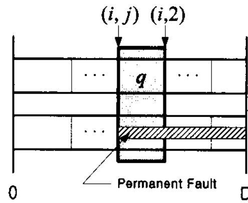
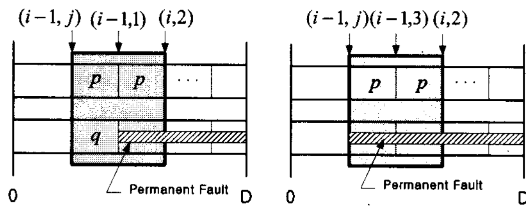


그림 6. 영구적 고장이 발생한 경우 $(i,j)\rightarrow(i,2)$로의 상태천이

Fig. 6. State Transition of $(i,j)\rightarrow(i,2)$.



(a) $(i-1,1)\rightarrow(i,2)$ (b) $(i-1,3)\rightarrow(i,2)$

그림 7. 영구적 고장이 발생한 경우 $(i-1,j)\rightarrow(i,2)$로의 상태천이

Fig. 7. State Transition of $(i-1,j)\rightarrow(i,2)$.

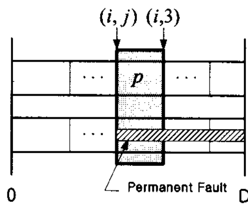


그림 8. 영구적 고장이 발생한 경우 $(i,j)\rightarrow(i,3)$로의 상태 천이

Fig. 8. State Transition of $(i,j)\rightarrow(i,3)$.

때문에 태스크의 i번째 블록이 성공적으로 수행되어 "i"가 1 증가되고, "j"는 영구 고장이 발생하여 1→2 (또는 3→2) 로 천이하는 경우이다.

그림 8.은 상태 $(i,3)$로의 천이를 보여준다. 영구적 고장이 두 프로세서 중 한 곳에 발생하고, 영구적 고장이 발생한 시간 슬롯에 다른 쪽 프로세서에서는 과도 고장이 없는 상태를 나타낸다. 이때 연속된 2 시간 슬롯 모두에서 고장이 없으면 그림 7. (b)에서와 같이 $(i+1,1)$로 천이한다.

그림 9.는 영구적 고장이 고려된 중복 구조 시스템의 마코프 모델이다. 영구적 고장 발생을 나타내기 위해 3.1절에서 과도 고장만 있는 경우에 정의한 상태 (i,j)의 두 번째 인덱스 "j"에 영구 고장 발생 후의 태스크 블록의 수행 상태를 나타내는 두 가지 상태 $((i,2),(i,3))$가 더 추가 되었다. 영구적 고장은 한번 발생하면 계속 존속하므로 상태 $(i,2)$나 $(i,3)$로 한번 천이한 후에는 다시 $(i,0)$나 $(i,1)$으로 천이 할 수 없다.

초기 상태가 (0,0)일 때 그림 9.의 마코프 모델에서 l 스텝 후 상태 $(n,0)$ 또는 $(n,2)$에 있을 확률을 구하면 태스크가 데드라인 이내에서 성공적으로 수행을 마칠 확률이 된다. k 번째 스텝에서 각 상태에 있을 확률을 나타내는 칼럼 벡터 $C_i(k)$는 다음과 같다.

$$C_i(k)=\begin{bmatrix} c_{i0}(k) \\ c_{i1}(k) \\ c_{i2}(k) \\ c_{i3}(k) \end{bmatrix} \tag{10}$$

영구 고장이 존재하는 경우에도 상태 천이 방정식 식 (8)은 그대로 사용할 수 있다. 단 상태 천이 확률 행렬은 다음과 같다.

$$M_p=\begin{bmatrix} \alpha^2q^2 & \alpha^2q^2 & 0 & 0 \\ 2\alpha^2pq & \alpha^2pq & 0 & 0 \\ 2\alpha\gamma q & 2\alpha\gamma q & \alpha q & \alpha q \\ 2\alpha\gamma p & \alpha\gamma p & \alpha p & 0 \end{bmatrix},$$

$$M_n=\begin{bmatrix} \alpha^2p^2 & \alpha^2(p^2+pq) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \alpha\gamma p & 0 & \alpha p \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{11}$$

여기서 α 는 Δ시간 구간에서 영구고장이 발생하지 않을 확률이고, γ 는 영구 고장이 발생할 확률이다. 영구 고장도 과도 고장과 같이 발생율 λ_p를 가지는 Poisson 분포에 따라 발생한다고 가정한다. α 와 γ 는 다음과 같다.

$$\alpha=e^{-\lambda_p\Delta},\ \gamma=1-e^{-\lambda_p\Delta} \tag{12}$$

3.3. 최적 체크포인터 수

그림 1.의 중복 구조 시스템에서 최적의 체크포인터 구간은 실시간 태스크가 데드라인 이내에서 수행될 확률을 최대로 하는 체크포인터 수를 찾으면 구할 수 있다. 과도고장만 있는 경우에는 그림 2.의 마코프 모델을 이용하고, 과도 고장과 영구고장이 동시에 있는 경우에는 그림 9.의 모델을 이용하여 식 (8)을 풀어서 구한다. 태스크에 n개의 체크포인터가 삽입되었을 때 데드라인 이내에서 성공적으로 수행될 확률(P)은 다음과 같다.

과도고장만 있는 경우: $P_n=c_{n0}(l)$,

초기조건: $C_0(0)=[1\ 0]^T$,

$C_1(0)=C_2(0)\cdots C_n(0)=[0\ 0]^T$

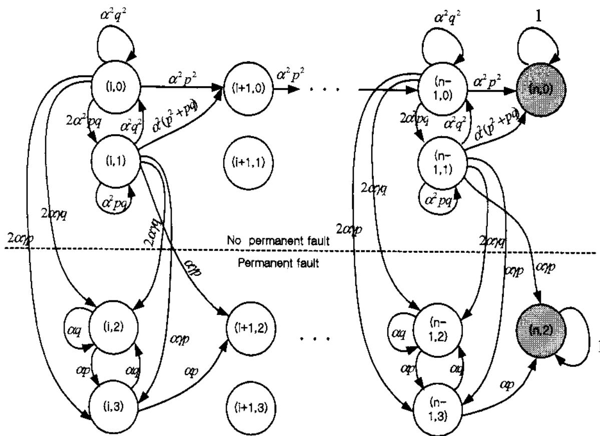


그림 9. 영구 고장과 과도 고장이 있는 경우의 중복 구조 체크포인팅 시스템 마코프 모델

Fig. 9. Markov model of the dual modular redundancy checkpointing system under transient and permanent faults.

과도고장과 영구고장이 있는 경우: $P_n = c_{n0}(l) + c_{n2}(l)$

초기조건: $C_0(0) = [1\ 0\ 0\ 0]^T$,

$C_1(0) = C_2(0) = \cdots C_n(0) = [0\ 0\ 0\ 0]^T$

여기서 l은 식 (7)에 의해 구해진다. 태스크에 삽입할 수 있는 최대 체크 포인터 수는 다음식과 같이 유도할 수 있다.

$$\bar{n} = \left\lfloor \frac{D-E}{t_{cp}} \right\rfloor \tag{12}$$

따라서 최적 체크포인터 수는 다음의 식 (13)과 같이 P_n을 최대로 하는 n^* 이다.

$$n^* = \underset{n}{Max}\ \{P_n\},\quad 1 \le n \le \bar{n} \tag{13}$$

Ⅳ. 시뮬레이션 결과

몇 가지 예제에 대하여 본 논문에서 제시된 방법으로 중복구조 시스템을 운용했을 때 최적의 체크 포인터 수를 구하였다. 그림 10.과 그림 11.은 과도 고장만 존재하고 영구 고장이 없는 경우에 데드라인 이내에서 태스크가 성공적으로 수행을 끝낼 확률 변화를 체크포인터 수에 따라 나타낸 것이다. 그림 10.은 데드라인 $D=1$, 태스크의 실행시간 $E=0.6$, 고장 발생률 $\lambda=0.1$, 체크포인터 오버헤드 $t_{oh}=0.02$ 인 경우 체크포인터 수에 따른 성공 확률 변화를 나타낸다. 태스크의 성공적 수행 확률을 최대화 시키는 최적 체크포인터 수는

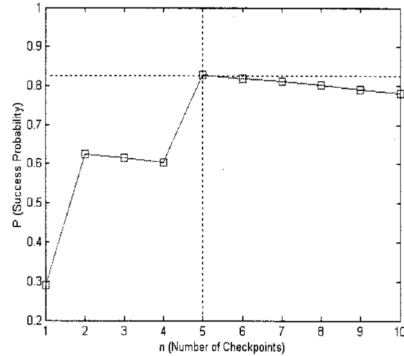


그림 10. 체크포인터 수에 따른 태스크의 성공적 수행 확률 변화 ($D=1,\ E=0.6,\ \lambda=1,\ t_{oh}=0.02$)

Fig. 10. Probability of successful execution vs. number of checkpoints. ($D=1, E=0.6, \lambda=1, t_{oh}=0.02$)

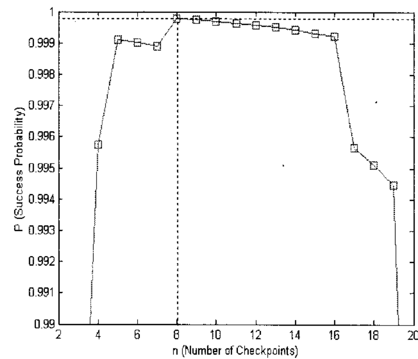


그림 11. 체크포인터 수에 따른 태스크의 성공적 수행 확률 변화 ($D=1, E=0.4,\ \lambda=5,\ t_{oh}=0.02$)

Fig. 11. Probability of successful execution vs. number of checkpoints. ($D=1, E=0.4, \lambda=5, t_{oh}=0.02$)

$n^*=5$ 이다. 태스크의 실행시간이 $E=0.4$인 경우의 성공적 확률은 그림 11.과 같다. 이 경우 $n^*=8$ 일 때 최적의 체크포인터 수가 된다. 체크포인터 수가 많아지면 체크포인터 오버헤드가 늘어남에 따라 수행시간의 증가를 가져온다. 따라서 체크포인터 수가 너무 많으면 도리어 성공적 수행 확률이 줄어든다는 것을 볼 수 있다.

그림 12.와 그림 13.은 과도 고장과 영구 고장이 동시에 존재하는 경우에 데드라인 이내에서 태스크가 성공적으로 수행을 끝낼 확률 변화를 나타낸다. 그림 12.는 데드라인 $D=1$, 태스크의 실행시간 $E=0.6$, 과도 고장 발생률 $\lambda_t=1$, 영구 고장 발생율 $\lambda_p=0.1$, 체크포

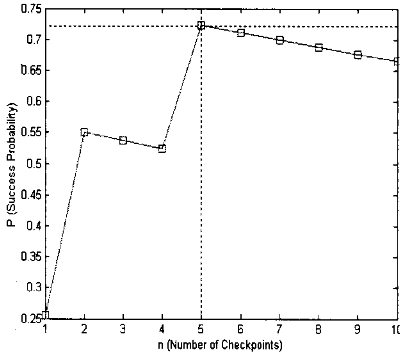


그림 12. 체크포인터 수에 따른 태스크의 성공적 수행 확률 변화 ($D=1, E=0.6, \lambda_t=1, \lambda_p=0.1, t_{oh}=0.02$)

Fig. 12. Probability of successful task execution vs. number of checkpoints ($D=1, E=0.6, \lambda_t=1, \lambda_p=0.1, t_{oh}=0.02$)

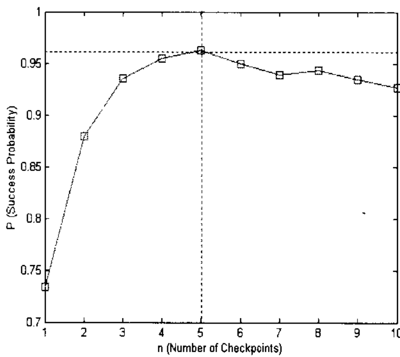


그림 13. 체크포인터 수에 따른 태스크의 성공적 수행 확률 변화 ($D=1$, $E=0.4$, $\lambda_t=1$, $\lambda_p=0.1$, $t_{oh}=0.02$)

Fig. 13. Probability of successful task execution vs. number of checkpoints ($D=1$, $E=0.4$, $\lambda_t=1$, $\lambda_p=0.1$, $t_{oh}=0.02$)

인터 오버헤드 $t_{oh}=0.02$ 인 경우 체크포인터 수에 따른 성공 확률 변화를 나타낸다. 그림 13.은 태스크의 실행시간이 $E=0.4$인 경우이다. $E=0.6$ 인 경우 $n^*=5$, $E=0.4$인 경우 $n^*=5$ 일 때 최적의 체크포인터 수가 된다.

그림 14.는 그림 13.의 예에서 영구적 고장 발생율이 매우 작은 $\lambda_p=0.001$인 경우를 나타낸 것이다. 영구적 고장이 없는 경우(영구적 발생률이 "0")인 그림 11.과 비교해보면 영구적 발생률이 매우 작아지면 영구적 고장이 없는 경우와 최적체크포인터 수가 같음을 볼 수 있다.

그림 10.~그림 14.에서 볼 수 있듯이 체크포인터 수

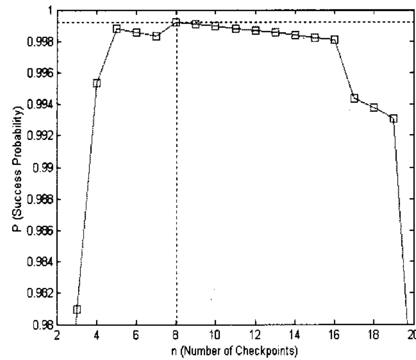


그림 14. 체크포인터 수에 따른 태스크의 성공적 수행 확률 변화 ($D=1$, $E=0.4$, $\lambda_t=1$, $\lambda_p=0.001$, $t_{oh}=0.02$)

Fig. 14. Probability of successful task execution vs. number of checkpoints ($D=1$, $E=0.4$, $\lambda_t=1$, $\lambda_p=0.001$, $t_{oh}=0.02$)

에 따른 태스크의 성공적 확률 변화는 Convex 함수가 아니라 톱니 형태의 계단 함수임을 알 수 있다. 즉 로컬 미니멈(local minimum)이 존재하므로 최적의 체크포인터 수의 위치가 어디인지를 알기 위해서는 식(13)에 의해 주어진 모든 영역($1 \le n \le \overline{n}$)에 대한 확률을 계산하고 비교하여야 됨을 알 수 있다.

V. 결 론

본 연구에서는 중복 구조를 이용하여 고장을 탐지하고, 체크포인터를 삽입하여 과도 고장 뿐 아니라 영구 고장에도 대응할 수 있는 방법을 제안하였다. 제안된 기법에 대하여 마코프 모델을 사용한 확률 해석 기법을 적용하여 제시된 방식을 최적화 하도록 설계하였다. 최적화를 위해 태스크의 성공적 수행 확률을 계산하고 이를 바탕으로 체크포인터 구간을 선정하였다. 과도 고장만 존재하는 경우와 영구고장과 과도고장이 동시에 존재하는 경우에 대하여마코프 모델을 유도하고 이를 최적 체크포인터 구간 선정에 이용하였다. 유도된 마코프 모델의 특징을 살펴보면 현재 상태에서 인덱스가 하나 증가된 다음 상태로의 천이만 존재하고 다른 상태로의 천이는 존재하지 않는 다는 것을 이용하여 빠르게 최종 확률을 계산할 수 있는 수식을 유도 하였다. 몇 가지 예제에 대한 시뮬레이션 결과로부터 본 논문에서 제안한 방식이 체크포인팅 기법을 탑재한 중복 구조 시스템을 최적화 하는데 사용할 수 있음을 살펴보았다. 체크포인팅 시점에 고장을 탐지할 수 있는 특성 때문에 체크포

인터 수에 따른 태스크의 성공적 확률은 Convex 함수가 아니라 톱니 형태의 계단 함수로 표시됨을 알 수 있었다.

참 고 문 헌

[1] 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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

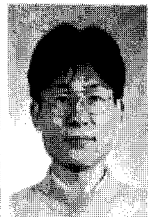
[9] Seong Woo, Kwak, "Reliability Analysis and Design of Real-time Fault Tolerant Control Systems under Transient Faults", Ph.D thesis, KAIST, 2000.

[10] 곽성우, 하드데드라인을 가지는 다중 실시간 주기적 태스크에서의 체크포인팅 기법, 전기학회논문지-D, 제53권 제8호, pp. 594-601, 2004년 8월

[11] 곽성우, 유관호, TMR 실시간 제어시스템의 내고장성 기법 및 신뢰도 해석, 제어.자동화시스템공학논문지, vol.10, no.8, pp.748-754, 2004년 8월

[12] 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.

저 자 소 개



곽 성 우(정회원)

1993년 한국과학기술원 전기및전자공학과 학사 졸업.

1995년 한국과학기술원 전기및전자공학과 석사 졸업.

2000년 한국과학기술원 전기및전자공학과 박사 졸업.

2000년 9월~2003년 2월 한국과학기술원 인공위성연구센터, 선임연구원, 연구교수

2003년 3월~현재 계명대학교 전자공학과 전임강사, 조교수

<주관심분야 : 실시간 제어 시스템, 임베디드 시스템, 고장 대응 및 극복, 위성 탑재 컴퓨터>