고신뢰 실시간 시스템을 위한 체크포인팅 프레임워크

(A Checkpointing Framework for Dependable Real-Time Systems)

이 효 순 † 신 현 식 ††
(Hyosoon Lee) (Heonshik Shin)

요 약 본 논문은 고신뢰 실시간 시스템에 체크포인팅을 적용할 수 있도록 실시간성과 신뢰성을 모두 고려하는 체크포인팅 프레임워크를 제공한다. 실시간 태스크의 시간 예측성은 할당된 체크포인트의 수와 태스크가 실행 중에 감내해야하는 고장의 수를 기반으로 태스크의 최악 실행 시간(WCET:Worst Case Execution Time)을 산출함으로써 보장된다. 태스크가 실행 중에 극복해야하는 고장의 수는 태스크의 신뢰성 요구조건을 기반으로 산출됨으로써 태스크의 신뢰성이 보장되도록 한다. 이렇게 얻어진 태스크들의 WCET와 태스크가 극복해야 하는 고장의 수를 이용하여, 각 태스크의 스케쥴 가능성을 보장하기 위해 요구되는 최소의 체크포인트 수를 유도하는 알고리즘을 제안한다. 본 논문에서 제안하는 프레임워크는 체크포인팅의 시간 중복량를 기반으로 하므로, 다른 시간 중복 기법에 대해서도 확장이 용이하다.

키워드 : 결함 허용성, 체크포인팅, 실시간 시스템, 스케쥴 가능성

Abstract We provide a checkpointing framework reflecting both the timeliness and the dependability in order to make checkpointing applicable to dependable real-time systems. The predictability of real-time tasks with checkpointing is guaranteed by the worst case execution time (WCET) based on the allocated number of checkpoints and the permissible number of failures. The permissible number of failures is derived from fault tolerance requirements, thus guaranteeing the dependability of tasks. Using the WCET and the permissible number of failures of tasks, we develop an algorithm that determines the minimum number of checkpoints allocated to each task in order to guarantee the schedulability of a task set. Since the framework is based on the amount of time redundancy caused by checkpointing, it can be extended to other time redundancy techniques.

Key words : Fault tolerance, Checkpointing, Real-time systems, Schedulability

1. 서 론

결함 허용성은 실시간 분산 시스템 설계에 있어서 가장 중요한 요구사항 중 하나이다[1]. 일반적으로 신뢰성이 요구되는 시스템의 경우, 영구적인 고장(permanent failure)은 다중의 하드웨어·소프트웨어 자원을 통해서만이 극복될 수 있는 반면에, 일시적 고장(transient failure)은 시간 중복(time redundancy)만으로도 극복할 수 있다. 시간 중복 기법은 하나의 모듈이 수행을 완료하기 전에 고장이 발생하는 경우, 상태를 저장해 두었던 이전 위치에서 해당 모듈을 재수행하거나 다른 백업용 버전을 이용하여 수행을 대신하는 기법이다. 태스크에 할당되어야 할 시간 중복의 양은 태스크가 실행을 완료하기 위하여 그 실행 중에 극복해야하는 고장의 수에 따라서 달라진다.

실시간 시스템에서의 신뢰성을 보장하기 위한 방법들은 신뢰성 요구조건 뿐만 아니라 시간 제약 조건들을 고려해야 한다. 실시간 태스크의 최악 실행 시간(WCET: Worst Case Execution Time)은 스케쥴 가능성을 보장하는데 널리 이용되는 중요한 척도이다. 태스크가 실행되는 도중에 고장이 발생하게 되면 고장의 검출과 극복 등을 수행하기 위한 별도의 시간이 추가로 소요되므로, 태스크의 WCET 산출과정에는 주어진 계산을 수행하는데 필요한 순수 계산 시간외에 결함을 검

† 비 회 원 : 서울대학교 컴퓨터공학부
fanta@cselab.snu.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
shinhs@cselab.snu.ac.kr
논문접수 : 2001년 5월 7일
심사완료 : 2002년 2월 18일

출하고 복구하는데 필요한 오버헤드가 포함되도록 하여야 한다.

무작위로 결함이 발생할 수 있는 시스템 환경 하에서 결함 회피 기능이 포함되어 있지 않은 태스크의 평균 실행 시간(expected execution time)은 발생한 고장의 수에 대해서 지수분포를 따라서 증가한다는 사실은 이미 잘 알려져 있다[2]. 이는 고장이 발생했을 때, 프로그램이 처음부터 다시 재시작해야 하기 때문이다. 그러나, 체크포인팅을 이용하면 고장이 발생했을 경우에 가장 최근에 저장된 상태에서부터 실행을 재개하기 때문에 태스크의 평균 실행 시간이 발생한 고장의 수에 대해서 선형적으로 증가한다. 체크포인팅은 응용의 신뢰도를 높이기 위하여 널리 사용되는 기법임에도 불구하고 실시간 시스템에서는 실행시간에 요구되는 오버헤드로 인하여 재시작(restart)기법이 널리 사용되어 왔다[1, 3]. 그러나, 현재의 기술 수준은 체크포인팅을 매우 적은 오버헤드를 가지고도 수행할 수 있을 정도로 발전하여, 실시간 시스템에서 체크포인팅을 적용할 수 있게 되었다. 예를 들어, Bowen과 Pradhan[4]은 프로세서 기반 체크포인팅 기법과 메모리 기반 체크포인팅 기법을 제안하였는데, 이 기법들은 체크포인팅이 무시할 만한 정도의 오버헤드로 수행될 수 있도록 한다.

체크포인팅은 하드웨어나 소프트웨어의 중복 기법이나 재시작(restart) 기법과 비교할 때 매우 경제적인 고장 감내 기법이다. 그러나, 체크포인트를 설정하는 데에는 비용이 수반되므로, 체크포인트를 자주 설정하면 할수록 고장 발생 후에 태스크를 재수행하는 데 필요한 오버헤드는 감소하게 되지만 체크포인트 설정에 소요되는 시간은 증가하게 된다. 따라서, 신뢰성 요구조건하에서 성능을 최적화 하는 체크포인트의 수가 존재하게 된다. 본 논문에서는 실시간 시스템에 체크포인팅을 적용할 수 있도록 하는 프레임워크를 제공한다. 우선, 체크포인팅을 채택하는 태스크들의 WCET를 산출하는 방법을 제공함으로써 실시간 태스크의 시간 예측성을 보장한다. 이 때, 태스크의 WCET는 할당된 체크포인트의 수와 태스크가 실행을 성공적으로 완료하기 위하여 그 실행 중에 극복해야하는 고장의 수를 기반으로 계산된다. 태스크가 실행 중에 극복해야하는 고장의 수는 태스크의 신뢰성 요구조건을 기반으로 산출됨으로써 태스크의 신뢰성이 보장되도록 한다. 마지막으로, 각 태스크의 스케줄 가능성을 보장하기 위해서 필요한 최소의 체크포인트 수를 유도하는 알고리즘을 제공한다.

본 논문은 다음과 같이 구성된다. 2장에서는 신뢰도를 고려하는 실시간 스케줄링 정책들과 대표적인 체크포인트 배치 정책들을 소개하고, 3장에서는 본 논문에서 사용되는 모델과 가정을 설명한다. 4장에서는 체크포인트의 수와 극복해야 하는 고장의 수를 기반으로 태스크의 WCET를 산출하는 방법을 설명하고, 5장에서는 신뢰성 요구조건을 만족시키기 위해서 태스크가 실행 중 극복해야 하는 고장의 수를 산출하는 방법을 제시한다. 6장에서는 4장과 5장에서 제시한 방법을 바탕으로 태스크의 스케줄 가능성을 고려하여 체크포인트를 배치하는 알고리즘을 제시하고, 7장에서는 본 논문의 주요 성과를 요약한다.

2. 관련 연구

시간 중복을 적용하여 고장을 감내하기 위한 실시간 스케줄링 정책들이 연구되어 왔다. Pandya와 Malek[5]은 RMS(Rate Monotonic Scheduling)를 사용하여 하나의 고장만을 감내하는 주기적인 태스크들의 스케줄 가능성을 분석하였다. 이들은 일시적인 고장을 대상으로 하였으며, 고장 감내 기법으로는 재시작 기법을 채택하였다. 이들은 시스템에 고장이 발생하면 실행이 완료되지 않은 모든 태스크들이 재수행된다고 할 때, 태스크 집합의 CPU 이용률(utilization factor)이 50%를 넘지 않으면 모든 태스크들이 종료시한을 넘기지 않고 수행 가능함을 증명하였다. Bertossi와 Mancini[1]는 경성 실시간 시스템을 위한 선점 가능한(preemptive) 스케줄링 알고리즘을 제안하였는데, 고장을 극복하기 위하여 주기적으로 태스크의 상태를 안정적인 저장장치에 저장하는 모델을 채택하였다. 두 가지의 프로세서 집합에 고장 감내성이 보장되지 않는 스케줄을 중복하거나, 주기적으로 각 태스크의 상태를 백업 프로세서에 저장함으로써 결함 허용성을 지원하였다. Ghosh et al.[6]은 연속된 고장 사이의 최소 시간 간격이 존재한다고 간주하고, 태스크가 종료시한 전에 재수행될 수 있을 만큼의 여유시간을 스케줄에 미리 예약해 놓는 동적 프로그래밍 해법을 제시하였다. 그러나, 이런 연구들은 프로그램의 재시작 기법만을 대상으로 하거나, 하나의 고장 혹은 하나의 신뢰성 요구조건만을 대상으로 함으로써 다양한 고장 발생 상황과 고장 감내 기법들에 대한 고려가 부족하였다.

성능을 최적화하기 위해 제안된 체크포인팅 정책으로는 equidistant와 equicost가 대표적이다. Equidistant 정책은 고장 발생 분포에 상관없이 모든 체크포인트 주기의 크기를 동일하게 하는 정책으로서, 임의로(random) 고장이 발생할 수 있는 환경에서 태스크의 평균 실행 시간을 최소화하고 시스템의 가용률을 최대

화한다. 이 정책은 매우 간단하기 때문에 시스템 설계자들에 의해서 많이 이용되어왔다[2]. Equicost 정책은 평균 재수행 비용이 체크포인팅 비용과 같게 되는 지점에서 체크포인트를 설정하는 정책이다. 이 정책은 고장 발생이 Weibull 분포를 따르는 경우, equidistant 정책보다 시스템 가용률을 더 높일 수 있다[8].

전방형 체크포인팅(roll-forward checkpointing) 기법은 고장을 극복할 때 자원의 중복을 이용하여 이전 체크포인트 상태로 복귀하는 것을 회피하기 위해 제안되었다[10, 11]. 이 기법은 하드웨어 자원의 중복을 요구하므로, 비용이나 전력 소모 그리고 부피 측면에서 중복도를 고려하여야 한다. 그러나, 만약 제공되는 자원의 중복도가 대처할 수 없을 만큼의 고장이 발생하거나 설령 자원의 여분이 남아있을 경우에도 최악의 시나리오로 고장이 발생하는 경우에는 이전 체크포인트 상태로 복귀하여 재수행하는 것이 불가피하다. 이런 경우, 태스크의 WCET는 단일 노드에서 체크포인팅을 구동하는 경우와 동일하게 된다.

현재까지 체크포인팅을 대상으로 하는 대부분의 연구들은 태스크의 평균 실행 시간을 최소화하는 것을 목표로 확률적인 분석을 행하여 왔다. 그러나, 실시간 시스템의 경우는 시간 제약을 만족하여야 하는 예측가능성이 중요시된다. 실시간 시스템에서의 체크포인팅을 분석하는 연구들은 간단한 모델 가정 하에서 체크포인팅을 채택하는 태스크의 WCET를 분석하였지만, 성능 최적화를 위한 체크포인트 배치 정책에 대한 연구가 부족하고 신뢰성 요구조건을 통합하여 실시간 시스템에 적용시키는 체계적인 방법론에 대한 제안은 이루어지지 않았다[12, 13, 14]. 본 논문에서는 체크포인팅 기법이 실시간 태스크에 미치는 시간적인 영향을 정량화하고, 실시간성과 신뢰성 요구조건을 동시에 만족시키면서 체크포인팅을 실시간 시스템에 적용 가능하도록 하는 프레임워크를 제시한다.

3. 모델과 가정

일반적으로 고장은 영속적, 일시적 또는 간헐적(intermittent)으로 분류된다. 관련 연구들[3, 15, 16]에서는 일시적인 오류가 대부분의 시스템 고장의 원인이 되고 있음을 보여주고 있기 때문에, 본 논문에서는 일시적 결함과 간헐적 결함만을 대상으로 한다.

이제까지 시스템의 고장 형태를 표현하기 위한 모델들이 많이 제안되어 왔다[16, 17]. 그 중 가장 널리 사용되고 있는 것은 고장들이 지수 분포나 Weibull 분포 혹은 이들 분포들의 변형을 따른다고 모델링하는 것이다. 본 논문에서는 고장 발생이 지수 분포를 따른다고 가정하며, 고장 발생률을 λ로 표현한다.

고장은 태스크의 수행 도중 언제라도 발생가능하며, 시스템은 적절한 결함 검출 기능을 탑재하고 있다고 가정한다. 결함 검출로 인하여 추가로 요구되는 오버헤드는 없다고 가정한다. 일반적으로 체크포인팅 수행 과정에서 수용성 검사를 수행하는 점에서 볼 때, 이 가정은 적절하다고 판단된다.

실제 시스템에서는 타임스탬프 값을 사용하는 명령어 블록과 같이 체크포인트를 설정할 수 없는 실행 구간이 존재하지만, 여기서는 이상적인 체크포인팅 정책을 도출하기 위하여 태스크 τ_i가 실행되는 동안 언제라도 체크포인트가 설정될 수 있으며, 체크포인트 설정과 이전 상태로의 복귀는 각각 C_i와 R_i 만큼의 시간이 소요된다고 가정한다.

모든 태스크는 경성 종료시한을 가지는 실시간 태스크라고 간주된다. 통상적으로 실시간 시스템에서 WCET로 일컬어지는 태스크 τ_i의 순수 계산 요구 시간(productive processing time) w_i는 반복적인 테스트나 시간 분석 도구에 의하여 실행 전에 미리 알려져 있다고 가정한다.

4. 체크포인팅을 이용하는 태스크의 WCET의 산출

이 장에서는 일정한 횟수의 고장 발생을 고려한 실시간 태스크의 WCET를 정량화하는 방법을 설명한다. $\alpha_i(n, k_i)$를 태스크 τ_i가 실행 시간 동안 최대 n개의 체크포인트 설정에 소요되는 시간과 최대 k_i 개의 고장 발생시 이를 극복하기 위하여 요구되는 시간을 합한 값이라고 하고, $A_i(n, k_i)$를 이 값의 최대치라고 하자. 태스크의 실행가능성을 보장하기 위한 태스크 τ_i의 최악 실행 시간 $W_i(n, k_i)$은 다음과 같이 표현될 수 있다.

$$W_i(n, k_i) = w_i + A_i(n, k_i) \quad (1)$$

이 시간 요구량은 최대 고장 수 k_i 보다 많은 고장이 발생하지 않는 한 유효하다.

태스크 τ_i가 n개의 체크포인트를 설정하면서 실행하는 중에 k_i개의 고장이 발생하였을 때, n번째 체크포인트가 설정된 이후에 발생한 고장의 수를 b라고 하고 j번째 고장이 발생한 체크포인트 간격을 P_j라고 하자. 태스크는 n개의 체크포인트를 설정하면서, j번째 고장

이 마지막 체크포인트 구간에서 발생한다면 태스크의 실행 시간에 추가되는 시간은 최대 P_j+R_i가 되고, 그 외의 체크포인트 구간에서 발생한 경우는 태스크의 실행 시간에 최대 $P_j+C_i+R_i$만큼 추가된다. 따라서 태스크의 WCET는 다음과 같다.

$$W_i(n,k_i)=w_i+k_i\cdot R_i+(k_i+n-b)\cdot C_i+\sum_{j=1}^{k_i}P_j. \quad (2)$$

만약, T_j를 j번째 체크포인트 간격이라고 하면, 식 (2)에서 $-b\cdot C_i+\sum_{j=1}^{k_i}P_j$는 $\{T_1, T_2, ..., T_n, T_{n+1}-C_i\}$에서 반복적으로 k_i개를 선택하여 그 값을 합한 것과 같으므로, $\{I_1, I_2, ..., I_n, I_{n+1}\}$를 $\{T_1, T_2, ..., T_n, T_{n+1}-C_i\}$의 내림차순 정렬이라고 할 때, WCET는 다음과 같다.

$$W_i(n,k_i)=w_i+n\cdot C_i+k_i\cdot(C_i+R_i+I_1) \quad (3)$$

식 (3)의 값은 $I_1=I_2=...=I_{n+1}$일 때 최소가 된다. 이러한 체크포인트 배치는 태스크의 실행 종료로 인하여 마지막 $n+1$ 번째의 체크포인트는 설정되지 않는 점을 제외하고는 equidistant 정책과 동일하다. 따라서, $T_1=T_2=...=T_n=\frac{w_i-C_i}{n+1}$이고, $T_{n+1}=\frac{w_i-C_i}{n+1}+C_i$ 일 때 식 (3)의 값은 다음과 같이 표현된다.

$$W_i(n,k_i)=w_i+n\cdot C_i+k_i\cdot\left(C_i+R_i+\frac{w_i-C_i}{n+1}\right) \quad (4)$$

이제, 식 (4)에서 계산되는 WCET 값을 최소화하는 체크포인트의 개수 n^*을 유도할 수 있다. 체크포인트의 개수 n은 0보다 크거나 같은 정수이고 $w_i-C_i>0$이므로, $\frac{\partial^2 W_i(n,k_i)}{\partial n^2}=2\cdot k_i\cdot\frac{w_i-C_i}{(n+1)^3}>0$. 따라서, $\frac{\partial W_i(n,k_i)}{\partial n}=0$을 통하여 WCET값을 최소로 하는 n을 구할 수 있다.

$$\frac{\partial W_i(n,k_i)}{\partial n}=C_i-k_i\cdot\frac{w_i-C_i}{(n+1)^2}=0$$
$$n=\sqrt{\frac{k_i\cdot(w_i-C_i)}{C_i}}-1$$

그런데, n은 정수이므로, $\left\lceil\sqrt{\frac{k_i\cdot(w_i-C_i)}{C_i}}-1\right\rceil$과 $\left\lfloor\sqrt{\frac{k_i\cdot(w_i-C_i)}{C_i}}-1\right\rfloor$ 중에서 식 (4)의 값을 더 작게 하는 값이 n^*가 된다. 따라서, $m=\frac{k_i\cdot(w_i-C_i)}{C_i}$라고 할 때, 최적의 체크포인트 개수 n^*는 다음과 같다.

$$n^*=\begin{cases}\lceil\sqrt{m}\rceil-1, & \text{if } \lceil\sqrt{m}\rceil\cdot\lfloor\sqrt{m}\rfloor\geq m>1\\ \lfloor\sqrt{m}\rfloor-1, & \text{if } m>\lceil\sqrt{m}\rceil\cdot\lfloor\sqrt{m}\rfloor>1\\ 0, & \text{otherwise}\end{cases}$$

그러나, 식 (3)에서 유도되는 WCET는 k_i개 이하의 고장이 발생할 경우에만 유효하다. 실시간 시스템에서 중요시되는 시간의 예측성은 k_i번째의 고장 발생까지만 지켜지기 때문에, 그 이상에 대해서는 체크포인트를 계속 진행한다 할지라도 실시간성을 보장할 수 없다. 따라서 실시간 태스크의 시간 예측성과 신뢰성을 모두 만족시키기 위해서는 태스크가 가지는 신뢰성 요구조건에 만족시키기 위하여 그 실행 중에 극복할 수 있어야 하는 고장의 수 k_i를 유도하는 것이 매우 중요한데, 5장에서 그 방법을 다루도록 한다. 우리는 이전 연구에서 k_i개의 고장이 발생할 때까지만 체크포인팅을 수행하는 모델을 제안하고 최적의 체크포인트 배치 정책을 제안하였다[18].

5. 신뢰성 요구조건을 기반으로 한 고장 발생 수의 산출

태스크 τ_i가 실행을 성공적으로 완료하기 위해서 그 실행 중에 극복해야 하는 고장의 수 k_i는 태스크가 가지는 신뢰성 요구조건에 의해 산출되어야 한다. 일반적으로 태스크의 신뢰성 요구조건은 신뢰도 목표치를 산정하거나 최소 고장 발생 시간 간격을 가정하는 형태로 표현되고 있다.

5.1 신뢰도 목표치 모델

시스템 설계자가 태스크 τ_i의 신뢰도 목표치를 ρ_i로 설정했을 경우, 태스크가 극복해야 하는 고장의 수 k_i는 다음과 같이 유도할 수 있다.

$$k_i=MIN\{j|\forall j\ R(W_i(n,j))>\rho_i\} \quad (5)$$

식(5)에서 함수 $R(t)$는 시간 t까지의 신뢰도를 나타낸다. $R(W_i(n,k_i))$의 값은 $W_i(n,k_i)$ 동안 최대 k_i 개의 고장이 발생할 수 있는 확률로서 다단계(multi-stage) Erlang 분포를 통하여 다음과 같이 계산할 수 있다.

$$R(W_i(n,k_i))=\sum_{j=0}^{k_i}\frac{(\lambda\cdot W_i(n,k_i))^j}{j!}\cdot e^{-\lambda\cdot W_i(n,k_i)}$$

5.2 최소 고장 발생 시간 간격 모델

이 모델은 두 개의 연속된 고장 발생 사이에는 최소한 T_F만큼의 시간 간격이 존재한다고 가정한다. 태스크의 WCET가 유한한 값으로 한정되기 위해서는 실행 중 연속된 T_F 시간 동안 적어도 하나의 체크포인트가 설정되어야만 한다. 따라서, T_F가 다음 조건을 만족하는 경우에만 실시간성을 보장할 수 있다.

$$T_F>R_i+MAX_{1\leq j\leq n}\{T_j+C_i, T_{n+1}\} \quad (6)$$

n개의 체크포인트를 가지는 태스크 τ_i가 실행을 성공적으로 완료하기 위하여 극복해야 하는 고장의 수는 다음과 같이 재귀적으로 산출할 수 있다.

$$k_i(0) = \left\lceil \frac{w_i}{T_F} \right\rceil$$
$$k_i(j) = \left\lceil W_i(n, k_i\frac{(j-1))}{T_F} \right\rceil \quad (6)$$
$$k_i = \lim_{j\to\infty} k_i(j)$$

그러나, 식(7)을 통한 방법은 $W_i(n, k_i)$의 값을 계산함에 있어서 연속된 고장 사이의 최소 시간 간격 T_F를 반영하지 않고 있기 때문에 태스크의 WCET 값을 과대하게 산출하는 단점이 있다. 이제부터 T_F를 반영하도록 체크포인팅 정책을 수정함으로써 k_i의 계산을 개선하는 방법을 설명한다. Lemma 1에서 이 고장 모델에서의 최악 고장 발생 시나리오가 가지는 특성을 설명하고, k_i의 값을 줄이는 방법을 Theorem 1에서 제안한다.

Lemma 1 *i번째 고장이 발생한 시간을 f_i라고 할 때, 최악의 고장 발생 시나리오는 다음을 만족시킨다.*

P1: 태스크의 실행 동안 적어도 하나의 고장이 발생한다.

P2: 모든 i에 대해서 i번째 고장은 $[f_{i-1}+T_F, f_{i-1}+2\cdot T_F)$ 시간 구간 안에서 발생한다. 이 때, $f_0 = -T_F$.

P3: 모든 고장은 체크포인트가 설정되기 직전이나 태스크가 종료하기 직전에 발생한다.

증명. 어떠한 고장이라도 고장을 극복하는데 필요한 오버헤드가 존재하므로 태스크의 실행시간을 증가시키는 효과를 가진다. 따라서 P1은 참이다.

P2의 증명 : 모델의 정의에 의해 어떠한 T_F 시간 구간 안에 복수개의 고장이 발생할 수 없으며, $f_i \geq f_{i-1}+T_F$가 성립된다. 만약, T_F가 태스크의 WCET 보다 크다면, P1에 의하여 하나의 고장만이 발생한다. 그러나, T_F가 태스크의 WCET 보다 작고, f_1이 T_F보다 늦다면, 고장이 발생하지 않은 $[0, f_1 - T_F)$ 시간 구간이 존재하게 된다. 그러나, 이 시간 범위동안 하나의 고장이 더 발생한다면 태스크의 실행시간은 더 커지게 된다. 따라서, 최악의 고장 발생 시나리오에서는 f_1이 $[0, T_F)$ 구간에서 발생하여야 한다.

$i>1$에 대해서, $f_i \geq f_{i-1}+2\cdot T_F$라고 가정하자. 그러면, 고장이 발생하지 않은 $[f_{i-1}+T_F, f_i - T_F)$ 시간 구간이 존재하게 된다. 이 구간 동안에 하나의 고장이 더 발생한다면 태스크의 실행시간은 증가하게 되므로 가정은 모순이 된다. 따라서, $f_i < f_{i-1}+2\cdot T_F$가 성립한다.

P3의 증명 : 마지막 체크포인트 설정 이후에 고장이 발생하는 경우, 그 고장이 태스크의 종료 직전에 발생할 때가 그렇지 않은 경우보다 태스크가 재수행되는 시간이 더 크게 된다. 마찬가지로, 고장이 두 개의 연속된 체크포인트 설정 사이에서 발생하는 경우에는 두 번째 체크포인트가 설정 완료되기 직전에 고장이 발생할 때가 체크포인트 설정 전 시간 구간동안에 발생하는 경우보다 태스크가 재수행되는 시간이 더 크게 된다. □

식 (6)에서 보인 바와 같이 임의의 T_F 시간 구간 안에는 적어도 하나의 체크포인트가 설정되어있다. j번째 고장이 발생된 시간을 f_j라고 일컫고, $f_0 = -T_F$라고 할 때, Lemma 1을 통해서 j번째 고장이 $[f_{j-1}+T_F, f_{j-1}+2\cdot T_F)$ 시간 구간 내에서 체크포인트 설정 직전이나 태스크 종료 직전에 발생함을 알 수 있다. 따라서 모든 j에 대해서 $[f_j, f_j+T_F-C_i)$ 시간 구간 중에 단지 하나의 체크포인트를 $f_j+T_F-C_i$에서 설정할 때, 해당 구간에서 처리되는 순수 계산 시간이 최대가 된다. 이렇게 배치되는 체크포인팅 정책을 T_F-*정책*이라고 부르자.

Theorem 1 *두 개의 연속된 고장 발생 시간이 적어도 T_F만큼 떨어지는 경우, T_F-정책을 따르는 체크포인팅을 이용하는 태스크가 실행을 성공적으로 완료하기 위해서 극복해야 하는 고장의 최대 수는 다음과 같다.*

$$k_i = \begin{cases} \left\lceil \frac{w_i}{T_F - R_i - C_i} \right\rceil - 1, \\ \quad \text{if } 0 < w_i - \left\lfloor \frac{w_i}{T_F - R_i - C_i} \right\rfloor \cdot (T_F - R_i - C_i) \leq C_i \\ \left\lceil \frac{w_i}{T_F - R_i - C_i} \right\rceil \quad otherwise \end{cases} \quad (8)$$

증명. 태스크 τ_i가 T_F-정책을 채택할 때, j번째 고장 발생 후 T_F 시간동안 처리되는 순수 계산 시간은 $T_F - R_i - C_i$와 j번째 고장 발생 시점에서의 잔여 순수 계산 시간 중 작은 값이 되며, 이 값은 다른 구간에서의 체크포인트 배치와는 무관하다. □

그림 1는 $w_i = 100, C_i = 10$, 그리고 $R_i = 7$일 때, k_i가 T_F에 따라서 어떻게 변화하는가를 보여주고 있다. T_F가 순수 계산 시간 w_i의 60% 이상이 될 때, 태스크가

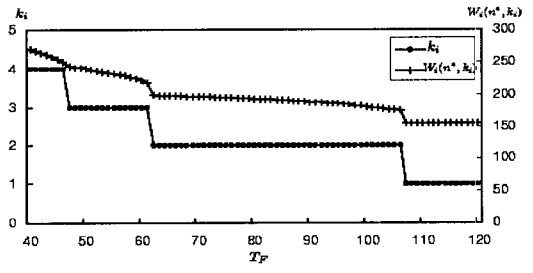

그림 1 T_F에 따른 변화

극복해야 하는 고장의 수가 2개 이하가 됨을 확인할 수 있다.

6. 스케줄 가능성을 고려하는 체크포인트 할당 정책

체크포인팅을 수행하는 동안 시스템이나 응용은 그 상태 값들이 모두 저장될 때까지 기능이 중단된다. 따라서, 시스템의 가용성을 최대화하기 위해서는 설정되는 체크포인트의 수가 최소가 되도록 운용하여야 한다. 이 경우, 태스크의 WCET를 최소화하기보다는 시간 제약 조건을 만족시키는 최소의 체크포인트 수를 찾아야 한다. 이 장에서는 태스크 집합의 스케줄 가능성이 만족되도록 각 태스크에 할당되어야 하는 체크포인트의 최소 수를 찾아내는 알고리즘 *MinCkpt*를 표 1과 같이 제안한다.

표 1 알고리즘 *MinCkpt*

```
알고리즘 MinCkpt
입력 : 새로 도착한 태스크 τ_new
출력 : 스케줄 가능 여부
{
/* 모든 태스크는 우선 순위에 따라서 정렬되어 있다.
   즉, τ_i의 우선 순위는 τ_{i+1}의 우선 순위보다 높다. */
/* num_i는 태스크 τ_i에 할당된 체크포인트의 수를 저장하고
   있는 변수 */
      num_new←0
  k_new← Get_ki_by_TF() 혹은 Get_ki_by_ρ_i()
  for ∀i≥new, τ_i에 대해서 do {
     while τ_i가 스케줄 가능하지 않으면 do {
        τ_g←Get_ τ_i_by_Gain()
        if τ_g = nil then return unschedulable
        num_g←num_g+1
        τ_g의 정보를 갱신
     } /* END OF WHILE LOOP */
  } /* END OF FOR LOOP */
  return schdulable
} /* END OF FUNCTION */
```

알고리즘 *MinCkpt*는 우선 새로 도착한 태스크 τ_{new}의 신뢰성 요구조건에 따라서 실행 중에 극복해야 하는 고장의 수 k_{new}를 계산해낸다. 이 값은 5장에서 설명된 방법에 따라 작성된 표 2의 알고리즘들을 통하여 유도된다. k_{new}의 값과 0으로 초기화되어 있는 num_{new}를 이용하면 태스크 τ_{new}의 WCET를 계산할 수 있으므로,

표 2 알고리즘 Get_k_i_by_ρ_i와 Get_k_i_by_T_F

```
알고리즘 Get_ki_by_ρ_i
입력 : τ_i , ρ_i
출력 : k_i
{
/* num_i는 태스크 τ_i에 할당된 체크포인트의 수를 저장하고
있는 변수 */
  k_i←0
  while R(W_i(num_i, k_i)) < ρ_i do k_i←k_i+1
  return k_i
}
알고리즘 Get_ki_by_TF
입력: τ_i, T_F
출력: k_i
{
  val← w_i / (T_F − R_i − C_i)
  if w_i − C_i ≤ ⌊ val ⌋ · (T_F − R_i − C_i)
     then k_i← ⌈ val ⌉ − 1
  else k_i← ⌈ val ⌉
  return k_i
}
```

τ_{new}의 스케줄 가능성을 검사할 수 있다. 스케줄 가능성 검사를 통하여 태스크들이 모두 스케줄 가능할 경우, 알고리즘은 체크포인트 배치를 마치고 종료한다.

그러나, 태스크 집합이 스케줄 불가능하다고 판단될 경우, 알고리즘 *MinCkpt*는 각 태스크들이 모두 스케줄 가능해 질 때까지 하나의 새로운 체크포인트를 추가 할당해 나간다. 하나의 체크포인트를 더 할당할 경우 스케줄 가능하지 않은 태스크 τ_i의 스케줄 가능성에 가장 크게 공헌을 할 수 있는 태스크를 탐색한다. 탐색 알고리즘 *Get_ τ_i_by_Gain*은 표 3에서 제시하고 있다. 알고리즘 *Get_ τ_i_by_Gain*은 스케줄 불가능한 태스크의 응답시간을 단축시키는 양으로 공헌 정도를 정량화하고, 그 값이 가장 큰 태스크를 리턴한다. 알고리즘 *Get_ τ_i_by_Gain*에 의하여 호출되는 알고리즘 *Get_Interference_by_ τ_j*은 상위 우선 순위 태스크가 하위 우선 순위 태스크에 미치는 간섭시간(interference time)을 계산한다. 이 알고리즘은 스케줄링 정책에 따라서 달라지므로 여기서는 자세히 다루지 않겠다. 가장 공헌이 큰 태스크에 하나의 체크포인트를 추가 할당함으로써 τ_i의 응답시간을 단축시킬 수 있으나, 그 결과로 신뢰성 요구조건을 만족시키기 위한 고장의 수 k_i도 역시 달라지게 되므로, 태스크

표 3 알고리즘 Get_ τ_i_by_Gain

```
알고리즘 Get_ τi_by_Gain
입력 : 종료시한 초과되는 태스크 τi
출력 : 태스크 τi의 종료시간을 가장 많이 단축시킬 수 있는
태스크
{
 ∀j, gainj←0
 for j=1 to i do {
   if j=i then result1←Wi(numi, ki)
   else {
        /* τj가 τi에 미치는 간섭시간을 계산 */
         result1←Get_Interference_of_ τj()
         numj와 kj의 값을 push
   }  /* END OF IF */
   numj←numj+1
   τj의 정보를 갱신
   if j=i then   result2←Wj(numj, kj)
   else {
        /* τj가 τi에 미치는 간섭시간을 계산 */
        result2←Get_Interference_of_ τj()
        numj와 kj의 값을 pop
   }  /* END OF IF */
   gainj←result1 − result2
 }  /* END OF FOR LOOP */
 gainmax←MAX{∀j, gainj}
 if gainmax≤0 then return nil
 return {τj|gainj=gainmax}
}  /* END OF FUNCTION */
```

의 정보를 갱신하여야 한다. 만약 태스크 τ_i가 하나의 체크포인트를 추가 할당했음에도 스케줄 가능해지지 않을 경우에는 τ_i보다 우선 순위가 높은 태스크들에 최적의 체크포인트 수 n^* 만큼의 체크포인트가 모두 할당될 때까지 앞의 과정을 반복한다. 우선 순위 높은 태스크들에게 최적의 체크포인트 수만큼을 모두 할당했음에도 스케줄 불가능할 경우, 알고리즘 *Get_ τ_i_by_Gain*()은 nil 값을 리턴하게 되므로 알고리즘 *MinCkpt*는 종료된다.

6.1 알고리즘 MinCkpt의 실행 예

이 장에서는 알고리즘 MinCkpt이 어떻게 동작하는지를 예를 통해서 보인다. 태스크 집합은 표 4에서 보이는 것과 같이 4개의 태스크로 구성되어 있다고 하자. 태스크들은 RMS 정책으로 스케줄링되며, 모두 동시에 도착하였다고 하자. 각 태스크의 스케줄 가능성은 실시간 시스템에서 널리 이용되는 대표적인 검사 방법인 Lehoczky[19]의 CT(Completion Time) 검사를 사용한다. 이 태스크 집합이 고장이 발생하지 않는 시스템 환경에서 실행된다면 CPU 이용률이 50%가 되므로 모두 스케줄 가능하다. 그러나, 태스크 집합은 하드웨어와 소프트웨어의 불완전성과 외부 요인 등으로 인해서 고장 발생률이 0.00159인 조건에서 실행되고, 각 태스크는 95% 이상의 확률로 정상적인 동작이 보장되어야 하는 요구조건을 가진다고 가정하자. 만약, 태스크들이 고장 감내 능력을 가지고 있지 않다면, 표 4에서 보이는 것과 같이 각 태스크들은 신뢰도 목표치 95%보다 작은 신뢰도를 가지게 된다.

표 4 태스크 집합

우선 순위	주기(=종료시한)	w_i	C_i	R_i	$R(w_i)$
1	500	100	10	7	0.870228
2	1000	170	16	13	0.789544
3	3000	200	20	16	0.757297
4	4000	250	30	20	0.706452
고장이 발생하지 않는 환경에서의 CPU 이용률 : 50%					

표 5 재시작 기법을 적용한 경우

우선 순위	k_i	$W_i(num_i, k_i)$	주기	$R(W_i(num_i, k_i))$
1	1	207	500	0.980564
2	3	719	1000	0.951874
3	3	848	3000	0.952923
4	5	1600	4000	0.952175
CPU 이용률 : 182%				

표 6 알고리즘 MinCkpt을 적용한 경우

우선 순위	k_i	num_i	$W_i(num_i, k_i)$	종료 시한	$R(W_i(num_i, k_i))$
1	1	3	157	500	0.956374
2	2	4	328.8	1000	0.973518
3	2	3	472	3000	0.959430
4	2	3	503.3	4000	0.952534
CPU 이용률 : 93%					

태스크 집합의 신뢰성 요구조건을 만족시키기 위하여 고장 감내 기법으로 재시작 기법을 적용하는 경우에는, 표 5에서 보이는 것과 같이 각 태스크는 1에서 5개의 고장을 극복해야 한다. 그 결과로 태스크 집합의 CPU 이용률은 100%를 초과하게 되어 스케줄이 불가능하게 된다. 즉, 재시작 기법을 사용하는 경우에는 태스크 집합의 신뢰성과 스케줄 가능성을 동시에 만족시킬 수는 없다.

알고리즘 *MinCkpt*를 통하여 태스크에 체크포인팅을 적용하는 경우에는, 재시작 기법과 비교해 볼 때 태스크의 WCET를 상당히 감소시킬 수 있으므로, 신뢰성과 스케쥴 가능성을 동시에 만족시킬 수 있는 기회가 증가하게 된다. 각 태스크에 할당된 체크포인트들이 이전 연구 [18]에서 제안된 체크포인팅 정책에 따라서 배치된다고 할 때, 알고리즘 *MinCkpt*가 실행된 결과로 각 태스크가 가지게 되는 특성은 표 6과 같다. 표 6에서 살펴볼 수 있듯이 각 태스크는 실행 중에 3에서 4개의 체크포인트를 설정함으로써 93%의 CPU 이용률로 최대 1에서 2개의 고장을 감내할 수 있으므로 신뢰성과 스케쥴 가능성을 동시에 보장할 수 있게 되었다. 표 7은 알고리즘 *MinCkpt*의 실행 단계별로 태스크들의 특성이 변화되는 과정을 보여준다.

7. 결론

실시간 시스템에서의 신뢰성을 보장하기 위한 방법들은 신뢰성 요구조건을 만족시키는 동시에 실시간성을 고려해야 한다. 우리가 아는 범위내에서 체크포인팅을 사용하는 태스크의 시간 예측성과 신뢰성 요구조건을 통합하여 실시간 시스템에 적용할 수 있도록 하는 체계적인 방법론은 제안된 바 없다. 본 논문에서는 체크포인팅 기법을 실시간 시스템에 적용할 수 있도록 하는 프레임워크를 제공하였다. 먼저, 실시간성을 보장하기 위하여 체크포인팅을 채택하는 태스크의 WCET를 산출하는 방법을 제공하였다. 실시간 태스크의 WCET는 스케쥴 가능성을 보장하는데 공통적으로 이용되는 중요한 척도이다. 그리고, 신뢰도 목표치와 최소 고장 발생 시

표 7 알고리즘 MinCkpt의 실행 예

우선순위	k_i	num_i	$W_i(num_i, k_i)$	종료시한	응답 시간	$gain_1$	$gain_2$	$gain_3$	$gain_4$
태스크 τ_1 도착: 스케쥴 가능									
1	1	0	207	500	207				
태스크 τ_2 도착: 스케쥴 불가능									
2	3	0	719	1000	1340	80	321		
태스크 τ_2에 체크포인트를 하나 추가 할당: 스케쥴 가능									
2	2	1	398	1000	812				
태스크 τ_3 도착: 스케쥴 불가능									
3	4	0	1712	3000	5936	240	106	592	
태스크 τ_3에 체크포인트를 하나 추가 할당: 스케쥴 가능									
3	2	1	472	3000	2908				
태스크 τ_4 도착: 스케쥴 불가능									
4	5	0	1600	4000	53944	320	141.3	40	840
태스크 τ_4에 체크포인트를 하나 추가 할당: 스케쥴 불가능									
4	3	1	760	4000	26932	320	141.3	40	80
태스크 τ_1에 체크포인트를 하나 추가 할당: 스케쥴 불가능									
1	1	1	167	500	167				
4	3	1	760	4000	8764	80	141.3	40	80
태스크 τ_2에 체크포인트를 하나 추가 할당: 스케쥴 불가능									
2	2	2	362.7	1000	696.7				
4	3	1	760	4000	5884	80	135.3	40	80
태스크 τ_2에 체크포인트를 하나 추가 할당: 스케쥴 불가능									
2	2	3	328.8	1000	495.8				
4	3	1	760	4000	5681	80	-1.2	40	80
태스크 τ_1에 체크포인트를 하나 추가 할당: 스케쥴 불가능									
1	1	2	157	500	157				
4	3	1	760	4000	4918.2	20	-1.2	40	80
태스크 τ_4에 체크포인트를 하나 추가 할당: 스케쥴 불가능									
4	3	2	680	4000	4838.2	20	-1.2	40	176.6
태스크 τ_4에 체크포인트를 하나 추가 할당: 스케쥴 가능									
4	2	3	503.3	4000	2903.8				

간 간격 제약을 신뢰성 요구조건으로 다루며, 각 모델로 표현되는 요구조건을 만족시키기 위하여 태스크가 실행되는 동안에 극복해야 하는 고장의 수를 산출하는 방법을 제시하였다. 이를 바탕으로 태스크의 스케줄 가능성을 고려하여 체크포인트를 할당하는 알고리즘을 제안하였고, 예를 통해 그 동작을 설명하였다.

본 논문에서 제시하는 프레임워크는 신뢰성 요구조건을 반영한 태스크의 WCET에 기반을 둔다. 따라서, 기존에 제안된 태스크의 WCET를 기반으로 하는 실시간 스케줄링 정책들에 어려움없이 적용할 수 있다. 그리고, 다른 시간 중복 기법을 채택하는 태스크의 경우에는 신뢰성 요구조건을 반영하여 WCET를 산출하는 방법이 응용될 수 있다.

참 고 문 헌

[1] A. Bertossi and L. Mancini, "Scheduling algorithms for fault-tolerance in hard real-time systems," Journal of Real-Time Systems, vol. 7, pp. 229-245, Nov. 1994.

[2] V. Nicola, "Checkpointing and the modeling of program execution time," in Software Fault Tolerance (M. Lyu, ed.), ch. 7, pp. 167-188, Chichester: John Wiley & Sons, 1995.

[3] S. Ghosh, Guaranteeing Fault Tolerance Through Scheduling in Real-Time Systems. PhD thesis, University of Pittsburgh, 1996.

[4] N. Bowen and D. Pradhan, "Processor and memory based checkpoint and rollback recovery," IEEE Computer, vol. 26, pp. 22-29, Feb. 1993.

[5] M. Pandya and M. Malek, "Minimum achievable utilization for fault-tolerant processing of periodic tasks," IEEE Trans. Computers, vol. 47, pp. 1102-1112, Oct. 1998.

[6] S. Ghosh, R. Melhem, D. Mosse, and J. Sarma, "Enhancing real-time schedules to tolerate transient faults," in Proceedings of Real-Time Systems Symposium, 1995.

[7] K. Shin, T. Lin, and Y. Lee, "Optimal checkpointing of real-time tasks," IEEE Trans. Computers, vol. 36, pp. 1328-1341, Nov. 1987.

[8] A. Tantawi and M. Ruschitzka, "Performance analysis of checkpointing strategies," ACM Trans. Computer Systems, vol. 2, pp. 123-144, May 1984.

[9] J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in Proceedings of USENIX Winter 1995 Technical Conference, pp. 213-223, 1995.

[10] D. Pradhan and N. H. Vaidya, "Roll-forward checkpointing scheme: A novel fault-tolerant architecture," IEEE Trans. Computers, vol. 43, pp. 1163-1174, Oct. 1994.

[11] J. Long, W. Fuchs, and J. Abraham, "A forward recovery strategy using checkpointing in parallel systems," in Proceedings of International Conference on Parallel Processing, pp. 272-275, 1990.

[12] A. Burns, R. Davis, and S. Punnekkat, "Feasibility analysis of fault-tolerant real-time task sets," in Euromicro Workshop on Real-Time Systems, pp. 29-33, 1996.

[13] S. Punnekkat, Schedulability Analysis for Fault Tolerant Real-Time Systems. PhD thesis, University of York, 1997.

[14] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," Journal of Real-Time Systems, Jan. 2001.

[15] R. Iyer, D. Rossetti, and M. Hsueh, "Measurement and modeling of computer reliability as affected by system activity," ACM Trans. Computer Systems, vol. 4, pp. 214-237, Aug. 1986.

[16] X. Castillo, S. McConnel, and D. Siewiorek, "Derivation and calibration of a transient error reliability model," IEEE Trans. Computers, vol. 31, pp. 658-671, July 1982.

[17] D. Pradhan, Fault-Tolerant Computer System Design. Prentice-Hall, 1995.

[18] H. Lee, H. Shin, N. Chang, "Checkpoint placement for fault-tolerant real-time systems," in Preprints of IFAC Workshop on Distributed Computer Control Systems, pp. 61-66, 2000.

[19] J. Lehoczky, L. Shar, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in Proceedings of Real-Time Systems Symposium, pp. 166-171, 1989.

이 효 순

1994년 서울대학교 컴퓨터공학과 학사
1996년 서울대학교 컴퓨터공학과 석사
2002년 서울대학교 전기 컴퓨터공학부 박사 현재 삼성전자 정보통신 총괄 무선사업부 책임연구원
연구 관심 분야는 실시간 운영체제, 결함허용 기법, 시스템의 실시간성 분석

신 현 식

1973년 서울대학교 응용물리학과 공학사. 1980년 미국 텍사스 대학교 의공학과 공학석사. 1985년 미국 텍사스 대학교 전기·컴퓨터공학과 공학박사. 1886년 ~ 현재 서울대학교 컴퓨터공학부 교수. 관심분야는 실시간 계산, 분산 시스템, 입출력 처리