논문 2006-43CI-2-2

연성 실시간 태스크들의 스케줄링을 위한 적극적인 슬랙 재활용

(Aggressive Slack Reclamation for Soft Real-Time Task Scheduling)

김 용 석*

(Yong-Seok Kim)

요 약

실시간 태스크들의 스케줄링에 있어서 일반적으로 주어진 태스크 집합에 대하여 최악의 실행시간을 적용하여 시스템의 요구 성능을 결정한다. 멀티미디어 시스템에서와 같이 연성 실시간 태스크들에 대해서는 이보다 낮은 성능의 저가 하드웨어로도 주어진 태스크 집합을 적절히 처리할 수 있게 된다. 태스크의 실행시간은 매 주기별로 가변적인데 실제 실행과정에서 한주기의 작업이 조기에 완료되면 남는 실행시간의 슬랙은 실행시간을 초과하는 태스크들이 공유하여 사용함으로써 전체적으로 태스크들이 마감시간을 초과하는 빈도를 줄일 수 있다. 본 논문에서는 슬랙들을 보다 적극적으로 공유하여 사용하는 알고리즘을 제시하였고 이를 통해 기존의 연구결과들에 비해서 마감시간을 초과하는 빈도를 줄이고 태스크 간의 문맥교환회수도 개선하였다.

Abstract

In scheduling of real-time tasks, the required hardware performance for a given set of tasks is determined based on the worst case execution time. For soft real-time tasks as multimedia applications, a lower performance hardware can service the tasks. Since the execution time of a task can vary in time, we can reclaim the slacks of early completed tasks for those of longer than average execution times. Then, the average ratio of deadline-miss can be lowered. This paper presents an algorithm, Aggressive Slack Reclamation (ASR), that tasks share slacks aggressively. A simulation result shows that ASR enhances the deadline-miss ratio and number of context switches than previous results.

Keywords: 실시간 스케줄링, 연성 실시간 태스크, 슬랙 재활용, 멀티미디어 시스템

I. 서 론

멀티미디어 시스템에 있어서 비디오나 오디오는 데이터 수집, 부호화, 복호화, 및 디스플레이 등이 항상 일정한 시간 간격으로 이루어 져야 한다. 최근에는 하나의 시스템에서 여러 채널의 멀티미디어 컨텐츠를 동시에 처리해야 하는 시스템의 수요가 많아지고 있다. 예를 들어서 여러 곳에 설치된 카메라들로부터 수집된 데이터들을 화면상의 여러 개의 창들에 동시에 표시해야 하는 경우가 이러한 시스템에 해당한다. 멀티미디어 컨텐츠는 매 주기마다 처리해야 하는 작업의 양이 데이터의 내용에 따라 가변적이다. 따라서 특정한 시점에 여러 채널들에 대하여 실행해야 할 작업의 양이 증가하게 되면 일시적으로 마감시간을 초과하는 경우가 발생하게 된다.[1][2]

실시간 태스크 스케줄링에서 주어진 태스크 집합에 대해 스케줄 가능한지에 대한 적합성 (feasibility)을 검사하는 데에는 최악의 실행시간 (WCET: worst case execution time)을 적용한다. 각 태스크들이 WCET에 해당하는 실행시간이 필요하더라도 모두 마감시간을 만족할 수 있는지를 판단하게 된다. 그러나 실제 시스템에 적용하는데 있어서 WCET을 정확하게 예측하는 것은 매우

* 정회원, 강원대학교 전기전자정보통신공학부
(Kangwon National University)
※ 본 논문은 부분적으로 강원대학교 정보통신연구소의 지원을 받았음
접수일자: 2005년9월9일, 수정완료일: 2005년1월16일

어렵다. 최신의 컴퓨터들은 실행하는데 있어서 다양한 불확실한 요소들을 내포하고 있다. DMA, 파이프라이닝, 캐싱, 프리패칭 등의 프로세서 설계기술에 의한 것들과, 인터럽트 처리, 가상 메모리 등의 운영체제 기능에 의한 것들이 대표적인 불확실성의 요인들이다.

WCET을 정확하게 예측한다고 하더라도 이것을 기준으로 적합성을 검사하는 것은 매우 비효율적인 결과를 초래할 수 있다. 태스크들의 실제 실행시간은 상황에 따라 가변적이며 평균적인 실행시간은 WCET보다 훨씬 작을 수 있다. 이러한 경우에 WCET만을 기준으로 적합성 검사를 적용하는 것은 실행 가능한 태스크 집합을 축소시켜야 하거나, 주어진 태스크 집합을 처리하기 위한 하드웨어 성능을 결정하는 과정에서는 보다 고성능의 컴퓨터를 필요로 하게 된다.

연성 실시간 (soft real-time) 응용들 중에는 WCET이 평균적인 실행시간보다 훨씬 긴 경우가 많이 있다. 한 예로서 시스템에서 비디오 프레임을 부호화하거나 복호화할 때 프레임 데이터들에 따라 실행시간이 차이가 난다. 영상 추적 시스템에서는 움직이는 목표물을 추적하는데 있어서 목표물이 예측한 영역에 존재하면 실행시간이 짧지만 그렇지 못한 경우에 최악의 상황에서는 전체 필드를 검색해야하는 상황이 발생하고 따라서 WCET은 평균적인 실행시간에 비해서 훨씬 길게 된다.

특정 태스크 집합을 실행하는데 필요한 하드웨어 규격을 결정하는 데 있어서 WCET 대신에 평균적인 실행시간을 적절히 활용하면 훨씬 저가격의 시스템을 구현할 수 있게 된다. 실제 실행과정에서 평균적인 실행 시간보다 길게 실행되면 일시적인 과부하 현상이 발생할 수 있고 이에 대한 적절한 대응방안이 있어야 한다. 하나의 태스크가 주어진 시간 이상 실행하게 되면 EDF (earliest deadline first) 정책이나 RR (rate monotonic) 정책을 그대로 적용할 경우 도미노 현상에 의해 다른 태스크들도 마감시간을 넘기게 될 수 있다.[3,8] 특정 태스크가 실행시간이 길어지게 되면 이 태스크는 마감시간을 넘기더라도 다른 태스크에는 영향을 미치지 않고 자기 자신에게만 영향이 한정되어야 한다.[1,4]

특정 태스크가 할당시간을 초과했을 때 처리하는 방식으로서는 해당 인스턴스를 강제로 종료시키거나 마감시간을 초과하더라도 우선순위를 낮게 하여 그대로 실행할 수 있다. 앞의 방식은 태스크에 따라서는 해당 인스턴스가 임계영역을 실행중이거나 할 경우처럼 위험한 결과를 초래할 수도 있다. 연성 실시간 시스템에서는 후자의 방법을 적용하는 것이 적절하며, 특히 태스크 인스턴스들이 예상보다 빨리 완료됨으로 인한 여분의 시간을 재활용할 수 있는 장점이 있다.[5,6]

본 논문에서는 멀티미디어 시스템과 같은 연성 실시간 응용들에서 주기적인 태스크들을 실행하는데 있어서 일시적인 과부하가 발생하더라도 마감시간을 놓치는 것은 실행시간이 길어진 태스크들로 한정되도록 하면서, 실제 실행시간이 허용된 시간보다 빨리 완료되는 경우에 남는 시간을 실행시간이 길어지는 태스크들이 활용함으로써 전체적으로 마감시간을 놓치는 비율을 낮출 수 있는 스케줄링 알고리즘을 제안한다. 또한 운영체제의 오버헤드에 해당하는 문맥교환 회수도 기존의 연구결과들에 비해서 개선할 수 있음을 보여준다.

Ⅱ. 관련 연구

멀티미디어 시스템과 같은 연성 실시간 시스템에서는 WCET 대신에 평균적인 실행시간을 적용하여 시스템의 필요 성능을 결정함으로써 저가격의 하드웨어로도 태스크들을 실행할 수 있게 된다. 태스크들이 실제 실행되는 과정에서는 상황에 따라 평균적인 실행시간 이상으로 실행될 수도 있고 이보다 일찍 한주기의 작업을 완료할 수도 있다. 즉, 각 태스크는 실제 실행시간이 때에 따라 할당시간을 초과할 수도 있고 남길 수도 있다.

태스크들의 실제 실행시간이 할당시간을 초과하는 경우에 대한 대응방안으로는 일반적으로 프로세서에 대한 자원예약 (resource reservation) 방식을 사용한다[2]. CBS (constant bandwidth server)[1,4]는 EDF 정책을 기반으로 하면서 초과 소요시간에 대하여 태스크 자신의 이전 또는 이후 주기의 인스턴스들의 남는 시간을 활용하여 처리하는 방식을 제시하였다. 이 방식의 문제점으로는 특정 태스크에 할당된 시간이 실제 실행과정의 평균적인 실행시간 보다 부족하면 이 태스크는 계속 느리게 실행하게 된다. 반대로 특정 태스크에 할당된 시간이 실제 실행에 필요한 시간보다 과도하면 전체 시스템의 성능을 저하시키는 원인이 된다.

이러한 문제점을 해결하기 위한 방안으로서 각 태스크 인스턴스들이 할당된 시간보다 일찍 완료되면 이 슬랙을 다른 태스크들을 실행하는데 활용할 수 있다. GRUB (greedy reclamation of unused bandwidth)[5]에서는 CBS와 마찬가지로 EDF 정책에 따라 태스크들을 스케줄링하지만, 태스크가 자신에게 할당된 용량을 다 사용하지 않고 완료되면 남은 용량을 슬랙으로 등록한다. 하나의 슬랙은 <start, end, utilization>의 값으로 지정된다. 한

주기에 할당된 실행시간에서 태스크 인스턴스가 실제 사용하고 남은 용량이 A일 때, end는 해당 인스턴스의 마감시간으로, start는 마감시간으로부터 한주기의 할당 용량에 대한 A의 비율만큼을 뺀 시점으로, 그리고 utilization은 해당 태스크의 활용율 (utilization = 할당용량/주기)로 설정한다.

GRUB에서 각 태스크 인스턴스는 실제 실행된 시간에 전체 활용율을 나눈 양만큼 자신의 가상시간을 증가시켜 나가는데, 이것은 실행된 시간에 현 시점의 전체 태스크들의 활용율을 곱한 양 만큼만 자신에게 할당된 용량에서 사용하는 효과를 얻는다. 각 슬랙들에 대하여 그 구간 동안은 이 슬랙의 utilization 만큼을 전체 태스크들의 활용율에서 빼 준다. 이것은 곧 각 슬랙 구간에 대하여 그 utilization 만큼을 재활용하는 효과를 가져온다. GRUB는 이렇게 다른 태스크들의 슬랙 부분을 활용함으로써 CBS에 비해 응답시간 면에서 우수하고 또한 문맥교환 회수도 줄일수 있다.

BASH (bandwidth sharing)[6]는 DPE (Dynamic Priority Exchange)[7] 서버와 유사하게 슬랙들을 활용한다. 슬랙들은 <amount, end>의 목록으로 관리한다. Amount는 태스크 인스턴스가 할당된 시간에서 남긴 용량이고 end는 이 인스턴스의 마감시간으로 설정한다. 각 태스크 인스턴스는 자신의 실행시간에 대하여 슬랙에 등록된 용량들을 먼저 활용하는 것으로 처리한다. 즉, 인스턴스의 마감시간보다 end 시간이 이른 슬랙들 중에서 end 값이 가장 이른 것 순으로 실행 시간에서 차감한다. 이 슬랙들로 처리하지 못하고 남는 실행시간 부분만 자신에게 할당된 용량에서 사용한다. 시뮬레이션 결과에 의하면 BASH가 응답시간 면에서 GRUB보다 우수하다. 그러나 문맥교환 회수에 있어서는 본 논문에서 시뮬레이션을 통해 확인한 결과 [5]에서 주장한 것처럼 GRUB이 더 우수함을 보여주고 있다.

CBS, GRUB 및 BASH 모두 스케줄링 과정에서는 용량이 남아있는 실행가능 태스크 인스턴스들 중에서 EDF 정책에 의해 실행할 인스턴스를 선택한다. 각 태스크 인스턴스들이 슬랙들과 자신의 할당용량을 다 활용하고도 작업을 완료하지 못하면 태스크 자신의 다음 주기의 인스턴스를 위한 용량을 활용한다. 이때 마감시간도 다음 주기의 인스턴스의 마감시간으로 변경함으로써 상대적으로 늦게 실행되도록 한다. 이렇게 함으로써 실행시간이 주어진 용량을 초과하는 태스크로 인해서 다른 태스크들이 마감시간을 만족하지 못하게 되는 도미노 현상을 막을 수 있게 된다.

본논문에서는 GRUB과 BASH보다 슬랙들을 보다 적극적으로 사용하는 방안을 제시하고, 그 결과 태스크 인스턴스들이 마감시간을 초과하는 비율이 GRUB과 BASH에 비해 우수하고 문맥교환 면에서도 역시 이들보다 우수함을 보여준다.

II. 적극적인 슬랙의 공유

1. 슬랙의 분할

EDF 정책을 기반으로 스케줄링을 하면 하나의 태스크 인스턴스는 두 부분으로 나누어서 각각을 실행하는 것으로 처리해도 전체 실시간 태스크들에 대하여 마감시간을 만족하는 데에는 영향을 미치지 않는다.[3] 각 부분의 실행시간은 구간 길이의 비율대로 나눈다. 그림 1의 (a)에서 보는바와 같이 5부터 17까지의 시간 구간에 6 만큼의 실행이 필요한 태스크는 (b)와 같이 5부터 13까지의 시간 구간에 4만큼을 실행하는 태스크와 13부터 17까지의 구간에 2만큼을 실행하는 태스크로 분리하여 스케줄링을 적용해도 전체 스케줄링 결과는 동일하게 마감시간을 만족할 것이다.

태스크 인스턴스가 자신의 할당시간을 다 소모하지 않고 일찍 완료된다면 남는 할당시간은 슬랙으로 등록한다. 만약 그림 1에서와 같이 할당시간이 6인 태스크가 4만큼 실행하고 조기에 완료되었다면 이 태스크를 (b)와 같이 분리하여 C_1 부분은 실제 실행한 부분 태스크로, 그리고 C_2 부분은 슬랙으로 남는 부분 태스크로 처리할 수 있다. 태스크의 주기가 P, 매 주기별 할당용량이 C이고, 조기 완료된 인스턴스의 마감시간이 D, 실행하고 남은 용량이 A이면, 슬랙 구간 <start, end>은 다음과 같이 계산할 수 있다.

$$start = D - P * A / C$$
$$end = D$$

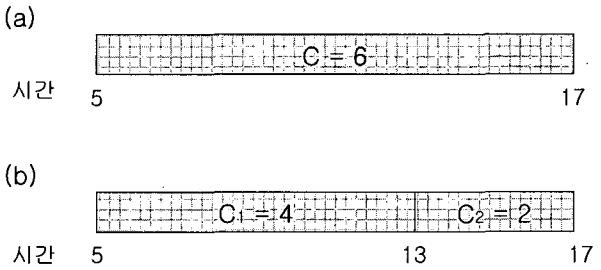


그림 1. 태스크의 분할
Fig. 1. Task Partitioning.

본 논문에서 제안하는 적극적인 슬랙 재활용 (ASR: Aggressive Slack Reclamation)은 각 슬랙을 다시 두 부분으로 분리하여 각각을 독립적인 슬랙처럼 사용한다. BASH에서는 end 시간이 태스크 인스턴스의 마감시간보다 앞인 슬랙들 만을 사용하는데, 만약 이 마감시간보다 뒤까지 걸쳐있는 슬랙들을 이 마감시간을 기준으로 분리한다면 그 전반부는 이 인스턴스가 사용가능하게 된다. 이와 같이 ASR은 슬랙들을 보다 적극적으로 활용함으로써 일시적으로 실행시간이 초과되더라도 마감시간 이내에 완료할 확률을 높여준다.

그림 2와 3은 GRUB, BASH 및 ASR에서 슬랙들을 사용하는 예를 비교하여 보여준다. 그림 2의 (a)는 용량이 3인 슬랙 S_1과 용량이 4인 슬랙 S_2가 남아있고, 실행시간 할당용량 (capacity)이 7만큼 남아있는 태스크 T가 7만큼의 시간동안 실행된 경우를 보여준다. 만약 CBS와 같이 다른 태스크의 슬랙을 전혀 사용하지 않는다면 T의 할당용량이 실행시간인 7만큼 감소하여 0으로 줄어들 것이다. GRUB에서는 (b)와 같이 모든 슬랙들에 대하여 현재시점까지의 부분을 활용하고 모자라는 부분을 자신의 할당용량에서 차감한다. 즉 S_1에서 1만큼, S_2에서 1만큼씩을 활용하고 자신의 할당용량에서 5만큼을 차감하여서 T의 할당용량이 2로 줄어든다. BASH에서는 (c)와 같이 T의 마감시간보다 앞에 존재하는 슬랙들을 사용한다. 즉, 슬랙 S_1만이 그 끝 시점이 T의 마감시간보다 이르므로 이것을 사용하고 나머지 4만큼을 자신의 할당용량에서 차감하여 T의 할당용량이 3으로 줄어든다.

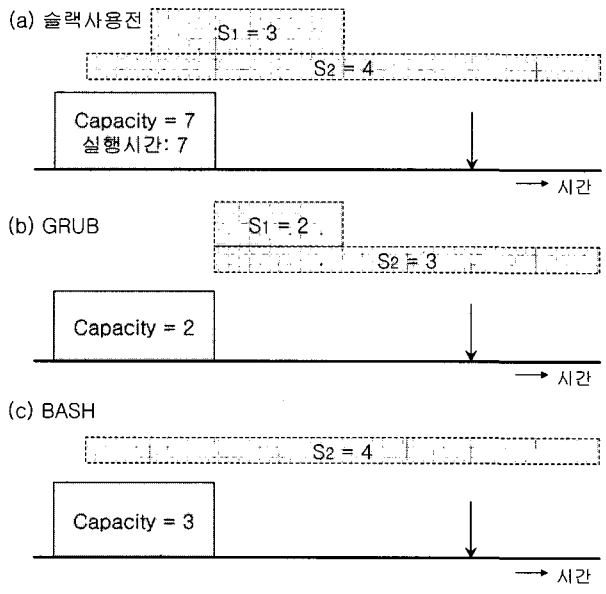


그림 2. GRUB과 BASH의 슬랙 사용
Fig. 2. Slack Reclamation of GRUB and BASH.

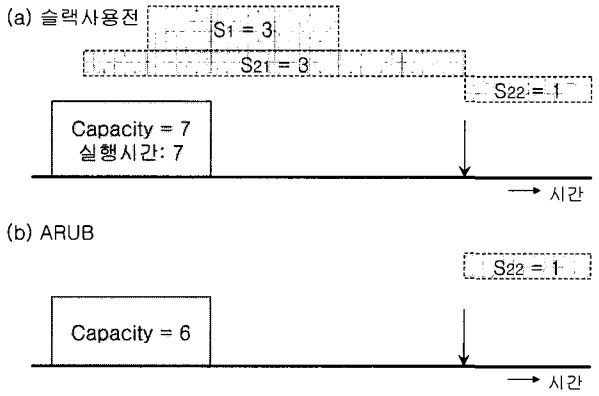


그림 3. ASR의 슬랙 사용
Fig. 3. Slack Reclamation of ASR.

ASR에서는 보다 적극적으로 슬랙들을 사용하는데, 각 슬랙을 사용할 태스크의 마감시간을 기준으로 분할하고 그 전반부의 슬랙도 활용한다. 그림 3의 (a)와 같이 슬랙 S_2를 S_{21}과 S_{22}로 시간 구간의 길이 비율대로 분할하고, (b)에서 보는바와 같이 S_1과 S_{21} 부분을 모두 사용한다. 따라서 실행시간 7중에서 슬랙 S_1과 S_{21}로부터 6만큼을 사용하고 자신의 할당용량에서는 1만큼만 차감하여 할당용량이 6만큼 남게 된다. 즉 T는 실제 실행시간이 정상적인 시간보다 6만큼 늘어나더라도 자신의 마감시간을 만족할 수 있게 된다.

2. 할당시간의 전진 보충

태스크는 실행시간에서 사용가능한 슬랙 부분들의 용량만큼을 제외한 나머지를 자신의 할당용량에서 차감한다. 특정 인스턴스의 실행시간이 정상적인 시간보다 길어져서 할당용량이 0에 도달하게 되면 더 이상 실행하지 못하고 다른 태스크로 전환해야 다른 태스크들에 연속적으로 영향을 미치는 결과를 방지할 수 있다. 이렇게 실행을 완료하지 못한 태스크 인스턴스를 처리하는 단순한 방법으로는 할당용량이 남아있는 다른 태스크들이 없을 때 남는 시간에 실행하는 후면 스케줄링 방식을 사용할 수 있다.[8]

CBS에서는 태스크 별로 자신에게 할당용량 한도 내에서만 실행하도록 제한하므로, 실행시간이 부족할 경우 자신의 다음 한 주기를 위한 할당시간 만큼을 할당용량에 보충한다. 대신에 마감시간을 한주기 뒤로 늘려줌으로써 EDF에 의한 스케줄링 과정에서 실행 우선순위가 낮아지고, 다른 태스크들에 영향을 주지 않으면서 완료할 수 있도록 한다. GRUB과 BASH에서도 이와 동일한 방식을 사용하고 있다.

본 논문에서 제안하는 ASR에서는 무조건 한주기 만

큼의 할당용량을 보충하는 대신에 다른 태스크보다 우선적으로 실행할 수 있는 한도 내에서 할당용량을 보충한다. ASR은 EDF 정책에 따라 스케줄링을 하므로 실행가능 태스크 인스턴스들 중에서 다음 차례의 인스턴스보다 마감시간이 늦어지지 않는 한도 내에서 할당용량을 보충하면 우선적으로 계속 실행할 수 있게 된다. 즉, EDF 정책에 기반한 실행가능 태스크 인스턴스 목록에서 다음에 선택될 태스크 인스턴스의 마감시간을 확인하고 이 구간까지의 할당용량만 보충한다.

예를 들어서 현재 태스크 T의 마감시간이 D, 주기가 P, 한주기의 할당용량이 C이고, 다음에 선택될 태스크 T_2의 마감시간이 $D_2 > D$라면, T의 할당용량과 마감시간은 다음과 같이 계산한다. 만약 다음 주기의 마감시간이 D_2보다 이르다면 GRUB나 BASH와 같이 마감시간을 다음 주기의 마감시간으로 설정한다.

$$a \leftarrow \min (P, D_2 - D)$$
$$D \leftarrow D + a$$
$$Capacity \leftarrow Capacity + a * C / P$$

이렇게 함으로써 T의 마감시간이 여전히 제일 이르므로 보충된 할당용량만큼 계속 실행할 수 있게 된다. 만약 다음 D_2가 D와 같은 경우에는 할당용량이 늘어나지 않으므로 위의 식에서 a를 한주기인 P만큼으로 설정하여 사용한다.

3. ASR 알고리즘

ASR은 기본적으로 EDF 정책을 기반으로 스케줄링을 실시한다. 각 태스크는 매 주기마다 실행할 수 있는 할당용량 한도 내에서 실행함으로써 프로세서를 태스크 별로 일정한 비율이상 사용하지 못하게 막아주고, 태스크 간에 서로 영향을 주지 않도록 해준다.

어떤 태스크 인스턴스가 자신의 할당용량을 다 소모하지 않고 일찍 완료되면 남는 시간은 슬랙으로 기록하였다가 다른 태스크들이 사용할 수 있도록 함으로써, 일시적으로 실행시간이 자신의 할당용량을 초과하더라도 마감시간 이내에 완료할 수 있도록 한다. 태스크는 슬랙들과 자신의 할당용량으로도 실행시간에 부족하면 자신의 다음 주기 부분의 할당용량을 일부분 추가하여 사용하여 자신의 작업을 완료하게 된다.

ASR의 알고리즘은 다음과 같다. 태스크 T의 주기는 P_T, 한주기당 할당시간은 C_T, 활용율은 $U_T = C_T / P_T$ 이다. 태스크의 특정 주기를 위한 인스턴스 J는 도착시간 R_J, 할당용량 $Capacity_J$, 마감시간 D_J를 갖는다. 각 슬랙은 <start, amount, end>를 갖는다. 슬랙들은 end 값이 작은 것이 먼저 오도록 리스트를 유지한다.

(1) 실행가능 상태의 태스크 인스턴스들 중에서 EDF 정책에 따라 마감시간이 가장 이른 것을 선택하여 실행한다. 현재 선택되어 실행중인 태스크를 T, 그 인스턴스를 J라 한다.

(2) J는 $Capacity_J$와 슬랙들의 용량 한도 내에서 실행한다. 현재 실행구간에서 사용한 시간을 used 라고 하면 다음과 같이 슬랙들을 사용하고 모자라는 부분은 $Capacity_J$에서 차감한다.

(2.1) 슬랙들 중에 end 시간이 D_J보다 이른 것들이 있으면 다음과 같이 $Capacity_J$ 대신에 이 슬랙들을 먼저 사용한다.

```
Slack = SlackListHead;
while (used > 0 && Slack != NULL
                        && end_Slack <= D_J) {
  if (amount_Slack <= used) {
    used = used - amount_Slack;
    Remove Slack from the slack list;
  } else {
    start_Slack = start_Slack +
       (end_Slack - start_Slack) * (used / amount_Slack);
    amount_Slack = amount_Slack - used;
    used = 0;
  }
  Slack = the next of Slack;
}
```

(2.2) used 만큼을 채울 수 없으면, 모자라는 부분은 다음과 같이 각 슬랙을 D_J를 기준으로 분할하여 D_J 이전 부분을 사용한다.

```
while (used > 0 && Slack != NULL) {
  if (start_Slack < D_J) {
    a = amount_Slack *
        (D_J - start_Slack) / (end_Slack - start_Slack);
    if (a <= used) {
      used = used - a;
      start_Slack = D_J;
    } else {
      start_Slack = start_Slack +
        (end_Slack - start_Slack) * (used / amount_Slack);
```

```
      amount_Slack = amount_Slack - used;
      used = 0;
    }
  }
  Slack = the next of Slack;
}
```

(2.3) 이것들로 실행시간 used 만큼을 채울 수 없으면 $Capacity_J$에서 모자라는 만큼을 사용한다.

$Capacity_J = Capacity_J - used;$

(3) J가 완료되면 남은 $Capacity_J$에 대하여 다음과 같이 새로운 슬랙 <start, amount, end>를 슬랙 목록에 추가한다.

```
amount = Capacity_J;
start = D_J - P_T * (amount / C_T);
end = D_J;
Add slack <start, amount, end> to the slack list in
        non-decreasing end order;
Capacity_J = 0;
```

그러나, 만약 $Capacity_J$가 부족해서 보충했던 경우라면 슬랙으로 추가하지 않고 $Capacity_J$를 다음 인스턴스로 넘긴다.

(4) J가 실행중에 $Capacity_J$를 다 사용했으나 아직 완료되지 않았으면 다음과 같이 $Capacity_J$를 보충한다.

```
n = the next earliest deadline of ready task
      instances;
a = min(P_T, n - D_J);
if (a == 0)
  a = P_T;
D_J = D_J + a;
Capacity_J = Capacity_J + a * C_T / P_T;
```

(5) 태스크 x의 새로운 인스턴스 j가 실행가능 상태로 되면 $Capacity_j$를 다음과 같이 보충한다.

```
t = the deadline of the previous instance of x;
// 새 인스턴스의 마감시간을 초과하지 않았으면
if (t < R_j + P_x) {
  D_j= R_j + P_x ;
  // 마감시간까지의 용량을 확보
  Capacity_j= Capacity_j + (D_j - t) * C_x / P_x;
} else {
  Refill Capacity_j as Rule (4);
}
```

(6) 실행할 태스크가 없는 구간에 대해서는 그 구간의 길이만큼을 규칙 (2)를 적용하여 슬랙들에서 차감한다.

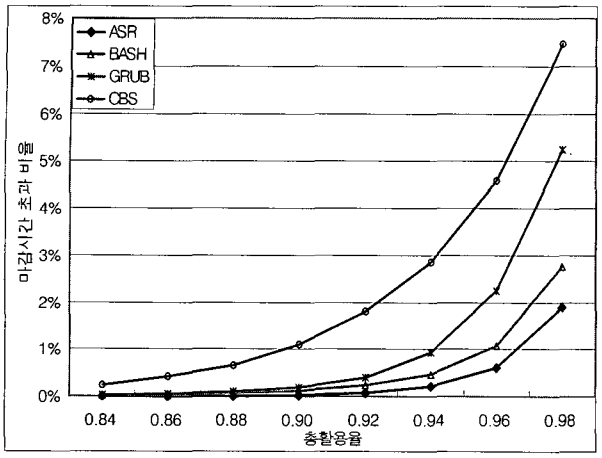
(7) 태스크들의 활용율의 합 U가 1 미만이면 다음과 같이 1-U 만큼의 활용율을 갖는 가상적인 태스크 V가 존재하는 것으로 처리한다. 태스크 V의 모든 인스턴스는 실제 실행과정 없이 바로 완료되고 C_V 양만큼의 슬랙을 추가하는 것으로 처리한다.

```
P_V = the hyper period of all tasks;
C_V = (1 - U) * P_V;
```

III. 성능평가

ASR의 성능을 평가하기 위해서 시뮬레이션을 통하여 기존의 CBS, GRUB, 및 BASH와 비교하였다. 시뮬레이션에 적용한 값으로는 태스크들의 주기를 10부터 500까지의 범위에서 임의로 선택하고, 태스크 별 활용율도 임의로 선택하되 전체 합이 1이하의 지정된 값이 되도록 하였다. 전체 시뮬레이션의 결과는 십만 시간 단위까지 실행하는 작업을 100번 반복하여 평균을 취한 것이다. 실행시간 변동 폭은 태스크 인스턴스들의 실제 실행 시간이 제공된 활용율 대비 변화하는 정도를 나타낸다. 예를 들어서 변동폭이 20%이면 실제 실행시간이 제공된 활용율 대비 80%에서 120% 까지 변화할 수 있음을 의미한다. 만약 주기가 20이고 제공된 활용율이 0.25라면, 변동폭이 20%일 경우 매 주기별 실제 활용율은 0.20에서 0.30 사이의 값이 될 수 있고 이것은 실제 실행시간이 4에서 6까지의 임의의 값을 가질 수 있음을 의미한다.

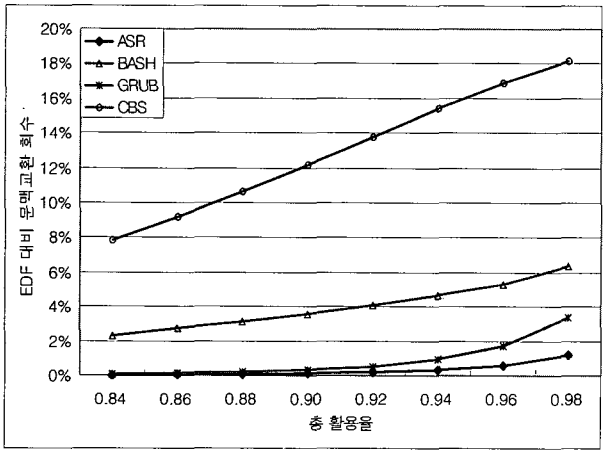
그림 4는 총 활용율의 변화에 따른 마감시간 초과 인스턴스들의 비율을 보여준다. ASR은 비교대상들 중 가장 우수한 성능을 보여주고 있다. 문맥교환회수 면에서도 그림 5에서 보는바와 같이 ASR이 가장 우수한 성능을 보여주고 있다. 그림 5에서 문맥교환회수는 슬랙들의 활용없이 단순히 EDF만을 적용했을 때의 문맥교환회수에 비하여 얼마정도 더 많은지를 비율로 보여준다. BASH와 GRUB를 비교하면 마감시간 초과 비율면에서는 BASH가 우수하지만 문맥교환회수 면에서는 [5]에서 주장했던바와 같이 GRUB이 더 우수하다. 참고로, EDF



태스크개수 10, 실행시간 변동폭 50%

그림 4. 프로세서 활용율에 따른 마감시간 초과율

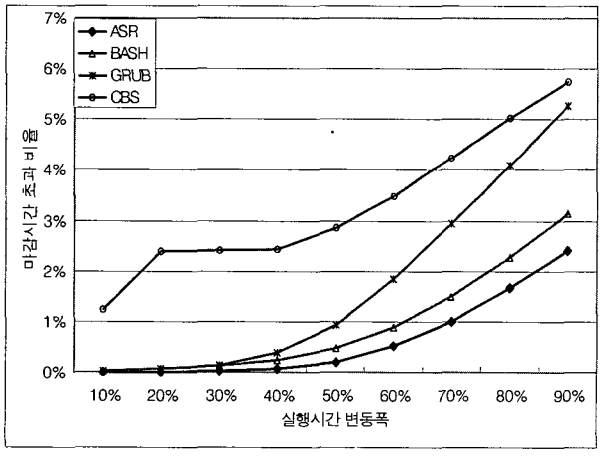
Fig. 4. Deadline Miss Ratio in Processor Utilization.



태스크개수 10, 실행시간 변동폭 50%

그림 5. 프로세서 활용율에 따른 문맥교환회수

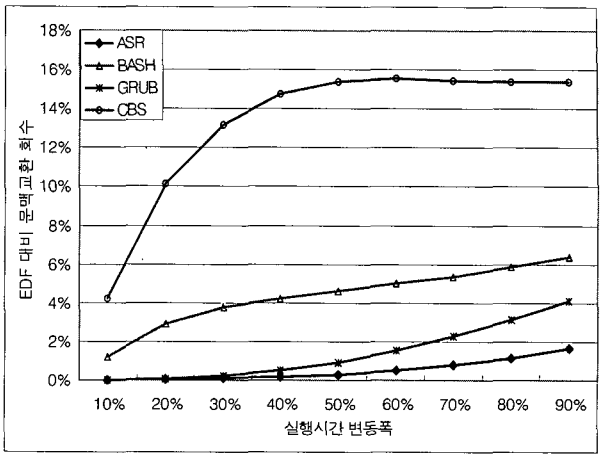
Fig. 5. Number of Context St\witches in Processor Utilization.



태스크개수 10, 총 프로세서 활용율 0.94

그림 6. 실행시간 변동폭에 따른 마감시간 초과율

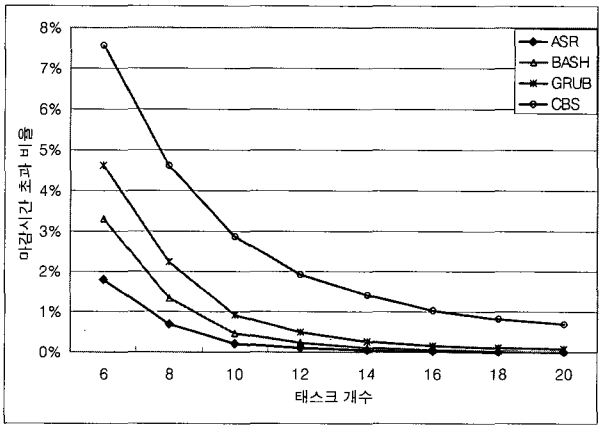
Fig. 6. Deadline Miss Ration in Variance of Computation Time.



태스크개수 10, 총 프로세서 활용율 0.94

그림 7. 실행시간 변동폭에 따른 문맥교환회수

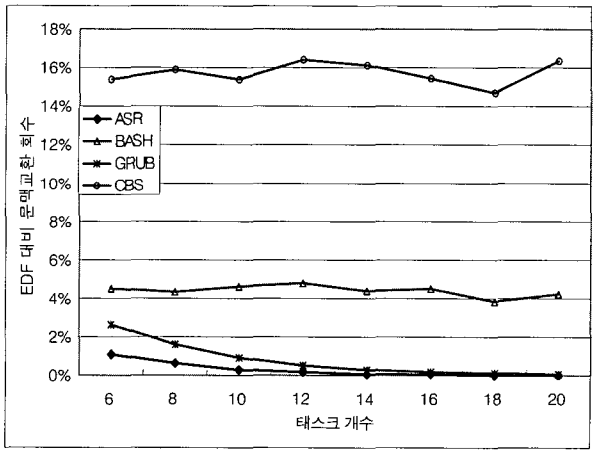
Fig. 7. Number of Context Switches in Variance of Computation Time.



프로세서 활용율 0.94, 실행시간 변동폭 50%

그림 8. 태스크 개수에 따른 마감시간 초과율

Fig. 8. Deadline Miss Ratio in Number of Tasks.



프로세서 활용율 0.94, 실행시간 변동폭 50%

그림 9. 태스크 개수에 따른 문맥교환 회수

Fig. 9. Number of Context Switches in Number of Tasks.

를 그대로 적용하면 문맥교환회수 면에서는 최적이지만 태스크 별로 프로세서 사용에 대한 할당시간을 적용하지 않으므로 하나의 태스크가 실행시간이 길어지면 다른 태스크들도 영향을 받아서 마감시간을 만족하지 못할 수도 있으므로 연성 실시간 태스크들을 스케줄링하는 데 사용하기에는 적절하지 못하다.

그림 6과 7은 태스크 인스턴스들의 실제 실행시간의 변동폭에 따른 마감시간 초과정도와 문맥교환회수의 변화를 보여준다. 전체적으로 실행시간의 변동폭이 증가할수록 마감시간 초과비율도 증가하고 문맥교환회수도 증가한다. ASR은 변동폭에 상관없이 항상 가장 우수한 성능을 보여준다.

그림 8과 9는 태스크 개수의 변화에 따른 마감시간 초과비율과 문맥교환회수는 보여준다. 태스크 개수가 늘어나면 슬랙들을 공유하여 활용하는 효과가 증가하므로 마감시간 초과 비율이 점점 줄어든다. 이 결과에서도 태스크 개수에 상관없이 ASR은 가장 우수한 성능을 보여주고 있다.

V. 결 론

주기적인 실시간 태스크들은 기본적으로 모든 태스크들이 마감시간을 만족하도록 적절히 스케줄링을 해주어야 한다. 그러나 항상 마감시간을 만족하는 것을 보장하기 위해서는 태스크별로 최악의 실행시간을 감안하여 충분한 성능의 하드웨어를 적용하든지, 아니면 적절히 태스크 집합을 줄여야 할 것이다. 멀티미디어 시스템에서와 같이 연성 실시간 태스크들에 대해서는 경우에 따라서는 마감시간을 초과하더라도 심각한 문제가 발생하지는 않는 경우에는 최악의 실행시간 대신에 평균적인 실행시간을 적용함으로써 낮은 성능의 저가 하드웨어로도 주어진 태스크 집합을 적절히 처리할 수 있게 된다.

태스크의 실행시간은 매 주기별로 항상 일정한 것이 아니라 입력 데이터나 여러 상황에 따라서 가변적이다. 따라서 실제 실행시간은 예측한 시간보다 짧아서 조기에 완료될 수도 있다. 이렇게 해서 남는 실행시간의 슬랙은 실행시간을 초과하는 태스크들이 재활용함으로써 전체적으로 마감시간을 초과하는 빈도를 줄일 수 있다.

본 논문에서는 기존의 연구결과들에 비해서 슬랙들을 보다 적극적으로 사용하는 ASR을 제시하였다. 슬랙들을 태스크들 간에 공유하여 활용함으로써 마감시간을 초과하는 빈도를 줄일 수 있는 기존의 연구로서 GRUB과 BASH가 있다. BASH가 GRUB에 비해서 우수한 결과를 보여주고 있는데, 문맥교환 회수에 의한 오버헤드 면에서는 오히려 GRUB가 BASH에 비해서 우수한 결과를 보여준다. 본 논문에서 제안한 ASR은 두 가지 측면에서 모두 BASH와 GRUB에 비해 우수한 결과를 보여주었다.

참 고 문 헌

[1] L. Abeni and G. Buttazo, "Integrating Multimedia Applications in Hard Real-Time Systems," *Proc. IEEE Real-Time Systems Symposium,* Dec. 1998.

[2] C. W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications," *Proc. 1st Int. Conference on Multimedia Computing and Systems,* IEEE, 1994.

[3] C. L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of ACM,* vol. 20, no. 1, pp. 40-61, 1973.

[4] L. Abeni and G. Buttazo, " Resource Reservations in Dynamic Real-Time Systems," *Real-Time Systems,* vol. 27, no. 2, pp. 123-167, Kluwer Academic Publishers, 2004.

[5] G. Lipari and S. Baruah, "Greedy Reclamation of Unused Bandwidth in Constant Bandwidth Servers," *Proc. of IEEE 12th Euromicro Conference on Real-Time Systems,* pp.234-241, June 2000.

[6] M. Caccamo, G. Buttazo, and D. Thomas, Efficient Reclaiming in Reservation-Based Real-Time Systems with Variable Execution Times,", *IEEE Trans. on Computers,* vol. 54, no. 2, pp. 198-213, Feb. 2005.

[7] M. Spurri and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Journal of Real-Time Systems,* 10(2), 1996.

[8] G. Buttazzo, *Hard Real-TimeComputing Systems,* Kluwer Academic Publishers, 1997.

저 자 소 개

김 용 석(정회원)

1984년 서울대학교 해양학과 학사졸업.

1986년 KAIST 전기및전자공학과 석사졸업.

1989년 KAIST 전기및전자공학과 박사 졸업.

1990년～1995년 한국생산기술연구원 및 전자부품 연구원 선임연구원

1995년～현 재 강원대학교 전기전자정보통신 공학부 컴퓨터전공 교수

<주관심분야 : 운영체제, 실시간시스템, 임베디드 시스템>