멀티미디어 응용을 위한 온-라인 스케줄링 알고리즘

심재홍°
조선대학교 인터넷소프트웨어공학부

An On-Line Scheduling Algorithm for Multimedia Applications

Jae-Hong Shim
Dept. of Internet Software Engineering, Chosun Univ.
E-mail : jhshim@chosun.ac.kr

요 약

본 연구에서는 시스템 과부하로 인해 충분한 실행시간을 가지지 못한 태스크의 중간 결과도 수용 가능한 범위 내에서 인정하는 멀티미디어 응용을 위한 동적 태스크 스케줄링 알고리즘을 제안하며, 스케줄링 알고리즘의 시간 복잡도보다는 실행 오버헤드를 줄이는데 초점을 둔다. 시뮬레이션 결과 태스크 도착율과 스케줄링 윈도우 크기에 따라 스케줄링 성능이 달라 진다는 것을 확인하고, 적절한 스케줄링 윈도우 크기를 결정할 수 있는 방안에 대해 논의한다.

1. 서론

대부분의 멀티미디어 응용을 구성하는 단위 태스크는 만기(deadline), 실행시간, 도착시간, 주기 등의 시간적 제약을 가지며, 태스크 스케줄러는 범용 스케줄러보다는 실시간 시스템에서 사용되는 스케줄링 알고리즘을 사용한다 [1, 2]. 멀티미디어 응용을 위한 태스크들은 그들의 만기 전에 모든 실행을 마쳐야 하지만, 시스템에 과부하가 걸렸을 때는 각 태스크에게 할당되는 프로세서 시간을 조금씩 줄여 스케줄링 한다. 예를 들어, 비디오 이미지를 압축하거나 해제하는 응용 [1], 멀티미디어 응용 [2], 정보 수집 및 추출, 멀티미디어 데이터베이스 질의 처리, 멀티미디어 통신을 위한 스위칭 시스템 등과 같은 응용에서는 시스템에 부하가 없을 경우 각 태스크에게 프로세서 시간을 많이 할애하여 그 만큼 태스크의 출력 결과의 품질(quality)을 높게 하지만, 과부하가 걸릴 경우 비록 태스크 실행 결과의 품질은 조금 떨어지더라도 각 태스크에게 할당되는 시간을 상대적으로 작게 한다.

이러한 응용에서는 태스크의 실행을 끝까지 완료하지 못하고 부분적으로 실행한 결과 (경성 실시간 시스템에서는 쓸모 없는 결과로 간주 함) 역시 나름대로의 의미(가치)를 가진다. 부분적으로 실행됨으로써 생성된 중간 결과는 최소한 우리에게 그 태스크가 무엇을 하려고 했는지에 대한 대략적인 정보를 제공한다. 만약 실행되지 못한 태스크의 나머지 부분도 모두 실행되었다면 보다 정밀한 결과를 생성했을 것이다.

이러한 태스크들로 구성된 멀티미디어 응용들을 지원하는 스케줄링 기술은 멀티미디어 시스템 내의 모든 태스크들의 만기를 준수하면서 정상적인 시스템 부하에는 최상(고수준)의 태스크 실행 결과(품질)를 보장하지만, 시스템 과부하 시에는 각 태스크의 만기를 넘기지 않은 채 수용 가능한 근사치의 실행 결과를 생산하여 시스템의 급격한 성능 감소를 막을 수 있는 이점을 제공한다.

위와 같은 특성을 가진 대표적인 태스크 모델로 IRIS(Increasing Reward with Increasing Service) 태스크 모델을 들 수 있다 [3-6]. IRIS 태스크들은 그들의 실행 결과로 가치(reward)를 생성한다. 이는 만기 전까지 그 태스크에게 주어진 프로세서 시간 양(실행시간)의 함수의 값으로 표현

되며, 이때의 함수를 가치함수(reward function)라 한다. IRIS 모델에서는 일반적인 실시간 시스템과는 달리 태스크의 실행시간이 사전에 결정되는 것이 아니라, 스케줄러에 의해 결정된다. 스케줄러는 스케줄링 목적에 따라 모든 태스크들의 만기를 넘기지 않는 범위 내에서 각 태스크에게 할당할 실행시간을 결정해야 한다. 따라서 태스크들은 만기 전까지 스케줄러가 할당해 준 시간만큼 실행될 수 있다.

본 연구에서는 멀티미디어 응용을 구성하는 온-라인 IRIS 태스크들의 총 가치(모든 태스크들의 가치의 합)를 최대화 시키면서 스케줄링 알고리즘의 실행 오버헤드(run-time overhead)를 줄이는데 초점을 둔다. 즉, 모든 태스크들을 대상으로 스케줄링하는 기존의 연구 방법과는 달리 본 연구에서는 스케줄링 당시의 가치 증가율이 가장 높은 몇 개의 태스크들을 대상으로 스케줄링하는 방안을 제시한다.

2. IRIS 태스크 스케줄링 알고리즘

IRIS 모델과 관련한 연구는 주로 모든 태스크들의 총 가치(모든 태스크들의 가치의 합)를 최대화 시키는 알고리즘을 찾는데 주력했다. 참고 문헌 [3, 4]에서 저자들은 IRIS 태스크들을 온-라인으로 스케줄링하는 세가지 정책을 설계하고 이들을 평가했으며, 또한 더 이상의 태스크가 도착하지 않고 동일한 도착시간을 가지는 임의의 태스크 집합이 주어졌을 때 이들의 총 가치를 최대화하는 정적 스케줄링 문제에 대한 최적의 해법을 구하는 알고리즘을 제안했다. 동일한 알고리즘 복잡도를 가지면서 최적의 해법을 구하는 또 다른 방법이 참고문헌 [5]에 의해 제시되었다. 기존의 연구들은 총 가치를 최대화 시키는 알고리즘의 개발 외에도 알고리즘의 복잡도를 줄이기 위한 연구도 함께 진행되었다. 참고문헌 [3]에서 제시된 알고리즘은 가치 함수가 비감소 오목 함수일 경우 $O(n^2)$, 지수 함수일 경우 $O(n \log n)$의 복잡도를 가지며, [5]에서도 동일한 복잡도를 가진다. 여기서, n은 태스크의 개수를 의미한다.

최근에 알고리즘의 복잡도를 줄이기 위한 또 다른 알고리즘이 참고문헌 [6]에서 제시되었으며, 이는 $O(cn)$의 복잡도를 가진다. 여기서 c는 작은 정수형 상수이다. 이 알고리즘은 크게 두 가지 아이디어를 기반으로 한다. 첫째, 온-라인 태스크들의 총 가치를 최대화 시키는 문제는 가치함수 도함수들의 최대값들 중에서 최소값을 찾는 문제를 해결함으로써 풀 수 있다는 것이다. 둘째는 새로운 태스크가 도착하기 전까지 이전에 스케줄된 태스크들 중 소수만이 실제 실행되고 나머지는 새로이 도착한 태스크와 함께 다시 스케줄링을 해야 한다는 사실에 주목하고, 한번 스케줄링할 때 모든 태스크를 스케줄링하는 것이 아니라, 최대 도함수 값들의 최소 값과 동일한 도함수 값을 가지는 태스크만 스케줄링하는 것이다. 그러나 [6]에서 제시한 알고리즘이 비록 $O(cn)$의 복잡도를 가진다고 해서 실행-오버헤드(run-time overhead)가 작은 것을 의미하는 것은 아니다. 즉, 이 알고리즘은 최대 도함수 값들의 최소 값을 구하기 위해 bisection 방법을 이용하며, 매번 모든 태스크들에 대해 가치함수 도함수의 역함수를 이용하여 각 태스크에게 할당할 서비스 시간을 계산하고, 모든 태스크에게 할당한 서비스 시간이 EDF(earliest deadline first) 알고리즘에 의해 스케줄링 가능한지를 체크 한다. 이는 비록 $O(cn)$의 알고리즘 복잡도를 가지지만 많은 태스크 개수를 가진 시스템에 적용하기에는 여전히 높은 실행 오버헤드를 가진다는 것을 의미한다.

3. 스케줄링 윈도우 크기를 가진 동적 스케줄링 알고리즘

본 절에서는 IRIS 모델을 따르는 태스크들의 총 가치는 수용할 만한 범위 내에서 약간 감소하지만, 실행 오버헤드가 현저히 감소하는 동적 스케줄링 알고리즘을 제안한다.

단일 프로세서 상에서 실행되는 선점 가능한 태스크들의 집합이 있다고 가정하자. 태스크 집합 내의 각 태스크 t_i, $i = 1, 2, .., n$,는 도착시간 r_i, 만기 τ_i, 가치함수 $f_i(x)$를 가진다. 태스크는 임의의 시간에 시스템에 도착할 수 있고 만기가 지난 후에는 시스템에서 삭제된다. 가치함수 $f_i(x)$는 스케줄러에 의해 태스크 t_i에 할당된 프로세서 시간(서비스 시간)의 양(x)의 함수로 정의된다. 즉, 태스크 t_i가 만기 전까지 총 x시간 동안 서비스를 받았다면 $f_i(x)$의 가치(reward)를 가진다. 가치함수는 일반적으로 비감소 볼록한 모양(non-decreasing concave)을 가진다. 가치함수는 미분 가능하고, 따라서 $f_i(x)$의 도함수를 $g_i(x)$로 표현하며 $g_i(x) \equiv df_i(x)/dx$라 정의한다. 도함수 $g_i(x)$는 가치함수 $f_i(x)$가 비감소 볼록형이므로 비증가 오목형(non-increasing convex) 함수이다. 즉, 태스크의 실행시간이 증가함에 따라 가치는 여전히 증가하지만, 증가율은 점점 감소하다가 동일해짐을 의미한다. 또한 도함수 $g_i(x)$의 역함수를 $g_i^{-1}(x)$로 표현한다.

모든 태스크들의 가치들의 합(총 가치)을 최대화하기 위한 가장 간단한 방법은 매 순간 가치 증가율 $g_i(0)$가 가장 높은 태스크에게 프로세서를 할당하는 것이다. 그러나 이 방법은 스케줄링 오버헤드가 너무 크므로 구현 불가능하다. 다른 방법은 [6]에서 제시한 방법으로 보다 높은 도함수 값(가치 증가율)을 가지는 태스크들에게 보다 많은 서비스 시간을 할당하는 방법이다. 이 방법은 다음과 같은 정리(theorem)를 기반으로 하고 있다.

[정리]

$\phi(\tau_0)$가 각 스케줄링 시점 τ_0에서의 가치함수 도함수들의 최대값들 중에서 최소값이라 가정하고, 태스크 집합 $\mathcal{N} = \{ t_i \mid g_i(0) \geq \phi(\tau_0) \} = \{ t_1, t_2, \ldots, t_l \}$ $(l \leq n)$ 내의 태스크들은 만기 순서로 정렬 (즉, $\tau_i < \tau_{i+1}$) 되어 있다고 가정하자. 이 경우, $y_i = g_i^{-1}(\phi(\tau_0))$ 라 할 때 $\sum_{i=1}^{k} y_i = \tau_k - \tau_0$ 인 그러한 k $(1 \leq k \leq l)$가 반드시 존재한다. 또한 이 때 구해진 각 y_i $(1 \leq i \leq k)$는 시점 τ_0에서 임의의 최적 알고리즘에 의해 구해진 상응하는 값과 동일하다.

따라서 [6]에서는 위 정리를 기반으로 상위 단계의 스케줄링 알고리즘에서는 $\phi(\tau_0)$를 구하고, $\sum_{i=1}^{k} y_i = \tau_k - \tau_0$ 인 k개의 태스크를 결정한 후, 이들에게 y_i의 실행시간을 할당해 준다. 즉, k개의 태스크들만 스케줄링한다. 하위 단계 스케줄링 알고리즘으로 EDF를 사용하며, 이는 만기 순으로 태스크들에게 프로세서를 할당하되 상위 단계에서 할당한 실행시간 y_i만큼만 실행한다.

스케줄링된 태스크들의 실행 도중 새로운 태스크가 도착하면 상위 단계의 알고리즘이 다시 실행되어 재 스케줄링 한다. 만약 상위 알고리즘에 의해 스케줄링된 태스크들이 하위 단계의 EDF에 의해 모두 실행될 때까지 새로운 태스크가 도착하지 않으면 상위 알고리즘이 다시 실행되어 앞 전에 스케줄링 되지 못했던 태스크들을 대상으로 새로이 스케줄링 한다. 이 경우는 이 알고리즘의 오버헤드로 간주된다. 즉, [3]에서는 매 스케줄링 시점(새로운 태스크가 도착하는 시점)에서 모든 태스크들을 스케줄링하지만, [6]에서는 τ_0에서 $\phi(\tau_0)$와 동일한 도함수 값을 가지는 k개의 태스크들만 스케줄링하기 때문에 복잡도는 줄지만, 알고리즘의 실행 횟수는 13% 범위 내에서 증가한다.

그러나 위의 방법은 앞 절에서 언급한 바와 같이 $\phi(\tau_0)$를 구하기 위해 bisection 방법을 이용하며, 이는 매 루프를 실행할 때마다 $g_i(0) \geq \phi(\tau_0)$인 태스크들에 대해 $y_i = g_i^{-1}(\phi(\tau_0))$를 계산하고, 또한 모든 태스크에게 할당한 서비스 시간이 하위 단계의 EDF 알고리즘에 의해 스케줄링 가능한지를 체크 하기 위해 $\sum_{i=1}^{k} y_i \leq \tau_k$ 인지를 체크해야 한다. 이는 비록 $O(cn)$의 알고리즘 복잡도를 가지지만 많은 태스크 개수를 가진 시스템에 적용하기에는 여전히 높은 실행 오버헤드를 가진다는 것을 의미한다.

따라서 본 연구에서는 [6]의 알고리즘의 실행 오버헤드를 줄일 수 있는 가장 간단한 방안으로 시스템 내의 모든 태스크들을 대상으로 스케줄링하는 것이 아니라, 사전에 정의된 조건을 만족하는 고정된 상수 W개의 태스크들을 선별하여 이들을 대상으로 스케줄링하는 방안을 제안한다. 이때 상수 W를 스케줄링 윈도우 크기(window size)라 한다. 사전에 정의된 조건의 예로 스케줄링 시점 τ_0에서의 순간 가치 증가율 $g_i(0)$가 가장 큰 순서 또는 만기가 가장 빠른 태스크 순서로 W개의 태스크를 선별할 수 있으며, 이는 시스템 설계자가 결정해야 할 사항이다. 다음은 이를 기반으로 하는 제안된 동적 스케줄링 알고리즘이다.

[휴리스틱(heuristic) 동적 스케줄링 알고리즘]

```
let T be a list of tasks that are currently in the system and sorted in
increasing order of their deadlines;
let τ0 be the current time, W be scheduling window size;
(1) select W's tasks from T by predefined condition; and let S be a
    list of selected tasks sorted by deadlines as T ;
    low = 0;   high = max{ gi(0) | ti ∈ S };
/* using bisection method, find φ(τ0) and k's tasks from S */
(2) if ((high – low) < ErrorBound) then goto step (2);
      1)  sum = 0;
          med = (high + low) / 2;
      2)  for k = 1, 2, ..., W do
              if (med < gk(0))  then   yk = gi^-1(med)
                                else   yk = 0;
              sum = sum + yk ;
              if (sum > τk - τ0) then
                    low = med, and go to step (1);
              else if (sum == (τk - τ0)) then
                    go to step (2); /* found φ(τ0) = mid */
      3)  high = med, and go to step (1) ;
(3) set the next scheduling instant to τk ;
    schedule tasks ti such that τi ≤ τk, by the EDF until τk or a new
        task arrival, and remove all terminating tasks from T;
```

모든 태스크들이 그들의 도착 시 그들의 만기 순으로 관

리된다고 가정할 때, 단계 (1)은 사전에 정의된 기준을 만족하는 W개의 태스크들을 만기 순으로 골라 내는 것이므로 $O(Wn)$의 복잡도를 가지며, 단계 (3)은 스케줄된 W개의 태스크들을 EDF에 의해 스케줄링하므로 $O(1)$을 가진다. 단계 (2)는 $\phi(\tau_0)$를 찾기 위해 c번 반복하고 매번 단계 2)를 W번 반복하지만, c와 W가 상수이므로 시간 복잡도는 $O(1)$이다. 여기서, c는 $ErrorBound \le \mathrm{Max}\{ g_i(0) \mid i = 1, 2, \ldots, W \} / 2^c$ 인 적은 정수 상수이다. 이는 단계 (2)를 위해 $O(cn)$ 복잡도를 가지는 기존 알고리즘과 대조적이다. 따라서 제안 알고리즘의 시간 복잡도는 $O(Wn)$이다. 실험 결과 c와 W는 시스템 및 태스크 특성에 따라 차이는 있겠지만 그렇게 큰 차이를 가지지 않는다. 따라서 두 알고리즘이 비슷한 시간 복잡도를 가지지만 알고리즘의 실행 오버헤드는 기존 알고리즘이 더 높다. 이는 실질적인 실행 오버헤드가 단계 (2)에서 높기 때문이다.

동적 스케줄링 알고리즘으로서의 제안 알고리즘은 새로운 태스크가 도착하거나 또는 이전에 스케줄된 태스크가 단계 (3)의 EDF에 의해 모두 서비스 되었을 때 다시 실행되어 재 스케줄링한다. 만약 새로운 태스크가 도착하기 전에, 상위 단계 (1), (2)에 의해 매번 1개의 태스크가 스케줄되고 이렇게 해서 시스템 내의 n개의 태스크가 모두 서비스될 경우 (최악의 경우), 알고리즘 복잡도는 $O(Wn^2)$이 된다. 그러나 기존의 스케줄된 태스크가 모두 서비스 되기 전에 새로운 태스크가 항상 도착한다면 (최선의 경우) 복잡도는 $O(Wn)$이 된다. 따라서 제안 알고리즘은 $O(Wn)$와 $O(Wn^2)$ 사이의 알고리즘 복잡도를 가진다. 그러나 시뮬레이션 결과에서 보여지듯이 새로운 태스크가 도착하기 전까지 스케줄된 태스크들 중 소수만이 실제 실행되므로, 실제적으로는 거의 $O(Wn)$에 가까운 복잡도를 가진다.

4. 시뮬레이션

제안 알고리즘의 실질적인 실행 복잡도와 생성된 총 가치를 확인하기 위해 시뮬레이션을 실시하였다. 성능 평가를 위해 두 가지 평가 기준인 R/O와 $(S-T)/T$를 정의하였다. 여기서 R과 O는 각각 제안 알고리즘과 최적 알고리즘에 의해 생성된 총 가치를 의미하며, S와 T는 각각 제안 알고리즘이 실행된 총 횟수와 태스크 도착 횟수를 의미한다. 따라서 R/O는 두 알고리즘에 의해 생성된 총 가치의 비율이며, $(S-T)/T$는 태스크 도착을 제외한 추가로 실행된 제안 알고리즘의 실행 횟수와 태스크 도착 횟수의 비율이다. 따라서 제안 알고리즘이 추구하는 성능은 R/O이 1이고 $(S-T)/T$가 0인 경우이다. (이후 $(S-T)/T$는 ST/T로 표기함).

각 태스크의 가치함수는 기존의 연구와 동일한 $f_i(x) = 1 - e^{-w_i x}$형태를 가진다. 태스크 도착은 포아송(Poisson) 분포를 따르며, 평균 도착율은 λ 이다. 태스크들이 시스템에 존재할 수 있는 시간 p_i (즉, $\tau_i - r_i$)는 평균 $1/\mu$ 를 가지는 지수 분포를 따른다. 시스템 내에서 서비스를 기다리는 태스크들의 평균 개수 ρ 는 λ/μ 와 같다. 서로 다른 w_i의 최대치 w_u와 λ, ρ 를 가지는 태스크 집합을 시뮬레이션하기 위해 매번 $M/M/\infty$ 큐잉 모델을 기반으로 25000개의 태스크를 생성하였으며, 또한 동일한 태스크 집합에 대해 서로 다른 스케줄링 윈도우 크기 W별로 매번 스케줄링하였다. 생성된 태스크들에게 부여된 w_i는 $(0, w_u)$ 범위에서 균등 분포를 따른다. 각 스케줄링에서 순간 가치 증가율 $g_i(0)$가 가장 큰 순서로 W개의 태스크를 선택하였다.

표 1은 $\rho = 40, w_u = 8$일 때의 λ와 W를 변화시키면서 스케줄링한 결과인 R/O와 ST/T를 보인 것이다. 표에서 $\lambda = 0.001$ 0.01, 100, 1000인 경우에는 $W = 1$인 경우를 제외하고, W의 크기에 상관없이 최적의 알고리즘과 동일한 총 가치($R/O = 1$)를 생성함을 알 수 있다. 즉, $W = 2$를 가지고도 충분히 최적의 알고리즘과 동일한 총 가치를 생성할 수 있다는 것을 의미하며, 반면에 스케줄링 알고리즘의 실행 오버헤드도 그 만큼 줄일 수 있다는 것을 의미한다 ($ST/T = 0$).

표에서 $W = 1$인 경우와 또는 $\lambda = 10$인 경우 적어도 90% 이상의 총 가치를 생성한다. 이는 응용에 따라 다르겠지만 스케줄링 오버헤드가 줄어드는 대신 이정도의 총 가치의 감소를 수용할 수 있는 응용에서는 적은 W를 가지고도 스케줄링 할 수 있다는 것을 의미한다. 예외적으로 $\lambda = 1$인 경우처럼 총 가치가 80%인 경우 응용의 특성에 맞게 W 크기를 적절히 늘릴 필요가 있다.

ST/T는 W가 7 이상인 경우 태스크가 도착할 때마다 제안 알고리즘이 수행되고 λ에 상관없이 추가로 수행되는 경우는 거의 없음을 확인할 수 있으며, 기존의 알고리즘과 비교해 볼 때 동일한 ST/T를 가진다. 이는 곧 $W = 7$을 가지고도 충분히 최적의 알고리즘과 동일한 총 가치를 생성할 수 있으며, 반면에 스케줄링 알고리즘의 실행 오버헤드는 $O(n)$이 된다는 것을 의미한다. 특이한 것은 $\lambda = 0.1$인 경우에 한해서만 W가 7이 될 때까지 다른 것에 비해 약간 높게 나타난다는 것이다. 또한 표에서는 보이지 않았지만 이

표 1. $\rho = 40, w_u = 8$일 때의 *R/O*와 *ST/T*의 변화

	Window Size	1	2	3	4	5	7	10	15	20	40
R/O	$\lambda = 0.001$	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	$\lambda = 0.01$	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	$\lambda = 0.1$	0.95	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	$\lambda = 1$	0.84	0. 86	0. 86	0. 86	0. 86	0. 86	0. 86	0.86	0.87	0.99
	$\lambda = 10$	0.92	0.93	0.93	0.93	0.93	0.93	0.93	0.95	0.97	1.00
	$\lambda = 100$	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	$\lambda = 1000$	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
ST/T	$\lambda = 0.001$	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\lambda = 0.01$	0.02	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\lambda = 0.1$	0.03	0.20	0.08	0.04	0.02	0.01	0.00	0.00	0.00	0.00
	$\lambda = 1$	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\lambda = 10$	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\lambda = 100$	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\lambda = 1000$	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

런 현상은 ρ가 10, 20, 40, 80인 경우에도 똑 같이 발생한다. 이는 곧 W가 7 이상인 경우, 선택된 W개만을 대상으로 스케줄링함으로써 상대적으로 적은 수의 태스크가 스케줄링되고, 추가로 알고리즘이 수행되는 회수가 증가할 것으로 예측하였지만, 실제로는 그렇지 않다는 것을 의미한다. 따라서 제안 알고리즘의 실행시 복잡도는 W가 7 이상인 경우 $O(n)$임을 의미한다.

결론적으로 제안 스케줄링 알고리즘의 실행 오버헤드인 *ST/T*는 기존 알고리즘과 거의 차이가 없는 $O(Wn)$인 반면, 총 가치는 $\lambda = 1$인 경우를 제외하고는 스케줄링 윈도우 크기를 2로 해도 기존 알고리즘과 동일한 총 가치를 생성한다고 할 수 있다. 따라서 실제 실행시의 알고리즘 복잡도는 $O(n)$으로 줄어 든다.

5. 결론

본 연구에서는 멀티미디어 응용을 위한 스케줄링 윈도우 크기를 가진 동적 태스크 스케줄링 알고리즘을 제안하였으며, 시뮬레이션 결과 태스크 도착율에 따라 근소한 차이는 있지만 실제 실행시의 알고리즘 복잡도는 $O(n)$으로 줄일 수 있음을 확인했다.

[참고문헌]

[1] E. Chang and A. Zakhor, "Scalable Video Coding Using 3-D Subband Velocity Coding and Multi-Rate Quantization," *Proc. IEEE Int'l Conf. Acoustic, Speech, and Signal Processing*, Minneapolis, July 1993.

[2] G. Jung, K. Yim, J. Jung, J. Shin, K. Choi, D. Kim, and J. Shim, "An Imprecise DCT Computation Model for Real-Time Applications," *Multimedia Technology and Applications*, edited by V. Chow, pp.153-161, Springer, Dec. 1996.

[3] J.K. Dey, J.F. Kurose, and D. Towsley, "On-Line Scheduling Policies for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks," *IEEE Trans. Computers*, vol. 45, no. 7, pp. 802-813, July 1996.

[4] J.F. Kurose, D. Towsley, and C.M. Krishna, "Design and Analysis of Processor Scheduling Policies for Real-Time Systems," in *Foundation of Real-Time Computing: Scheduling and Resource Management*, ed. by A.M.V. Tilborg and G.M. Koob, Kluwer Academic Publishers, pp. 63-89, 1991.

[5] K. Choi and G. Jung, "Comment on On-Line Scheduling Policies for a Class of IRIS Real-Time Tasks," *IEEE Trans. Computers*, vol. 50, no. 5, pp. 526-528, May 2001.

[6] G. Jung, T. Kim, S. Park, and K. Choi, "A Low complexity Dynamic Scheduling Algorithm for Real-Time Tasks," *Electronic Letters*, IEE, vol. 35, no. 24, pp. 2106-2108, Nov. 1999.