경성 실시간 태스크를 위한 확장된 스케줄 가능성 검사를 갖는 비율단조 스케줄러

Rate-Monotonic Scheduler with Extended Schedulability Inspection for Hard Real-Time Task

신동헌
금오공과대학교 컴퓨터공학과

조수현
금오공과대학교 컴퓨터공학과

김영학
금오공과대학교 컴퓨터공학부

김태형
금오공과대학교 컴퓨터공학부

Dong-Hern Shin
Dept. of Computer Eng., Kumoh National Institute of Technology

Soo-Hyun Cho
Dept. of Computer Eng., Kumoh National Institute of Technology

Young-Hak Kim
Dept. of Computer Eng., Kumoh National Institute of Technology

Tae-Hyong Kim
Dept. of Computer Eng., Kumoh National Institute of Technology

중심어 : 실시간 시스템, 비율단조, 스케줄러

Keyword : Real-time System, Rate-monotonic, Scheduler

요 약

오늘날 대부분의 내장형 시스템은 목적상 많은 기능뿐만 아니라 실시간성도 함께 요구하고 있다. 특히, 경성 실시간 시스템에서는 주기 태스크들의 엄격한 마감시간 보장이 시스템의 성능을 좌우한다.

본 논문에서는 CPU 이용률이 높아 비율단조 기법으로는 마감시간을 보장 할 수 없는 주기 태스크 셋을 위한 비율단조 기반의 스케줄러를 설계하고 구현한다. 이 스케줄러는 확장된 스케줄 가능성 검사를 실시하여, 태스크 셋의 수행 전 태스크들의 공통주기를 찾아 마감시간 우선 기법을 기반으로 마감시간 보장 수행패턴을 생성한다. 이렇게 생성된 수행패턴을 참조하여 결정된 우선순위에 따라 태스크 셋을 실행하게 된다.

마감시간 우선 기법을 기반으로 생성된 패턴은 그 특성에 따라 CPU 이용률을 100% 까지 가능하게 하며, 수행패턴을 참조하여 수행함으로써 동적 우선순위 할당 기법의 단점인 실행시간 스케줄링 오버헤드를 없앨 수 있다.

Abstract

Recently, most of the embedded system is required not only many functions but also real-time characteristics in purpose. In the hard real-time system, especially, strict deadline of periodic task can affect the performance of the system.

In this paper, we design and implement the scheduler based on RM(Rate-Monotonic) rule. This scheduler makes feasible patterns based on EDF(Earliest deadline first) rule with extended schedulability inspection before execution, for periodic task-set that has high CPU utilization and then, execute periodic task-set depended on feasible patterns.

The feasible pattern formed into EDF rule is capable of the efficiency of CPU up to 100 percentage and by the referenced execution of the feasible pattern is possible of removing the real-time scheduling overhead that is the defect of the order of dynamic assignment rule.

I. 서론

최근 빠르게 변화하고 있는 정보산업의 양상은 개인 상상력의 한계를 넘어서서 급격하게 변화하고 있으며, 각 분야별 분

접수번호 : #040504-001
접수일자 : 2004년 5월 4일, 심사완료일 : 2004년 5월 19일

*교신저자 : 조수현, e-mail : shcho@kumoh.ac.kr

화와 통합들이 활발히 진행되고 있다. 하나의 분야는 좀더 세분화, 전문화되어 나눠지고 발전된 각각의 기술들은 다시 통합되어 새로운 패러다임과 관련 기술들을 파생시키고 있다. 통합된 정보산업은 관련 기술자들에 의해 빠르게 새로운 시스템들로 대체되고 있다. 이 과정에서 등장한 많은 시스템들이 시장에 의해 짧은 시간 안에 사라지기도 하지만, 선택되어 발전, 진화된 시스템들은 정보산업의 흐름과 축을 변화시켜 전통적 과거 기술들의 중심을 바꾸어 놓게 된다. 이렇게 등장한 새로운 기기들 중 대표적인 것이 내장형 시스템(Embedded System)이다. 내장형 시스템들은 이미 그 수요에서 고전적인 컴퓨터 시스템을 압도하고 있고 세계적으로 판매되고 있는 마이크로프로세서의 90%가 내장형 시스템에서 사용되고 있다. 내장형 기기들은 과거와는 달리 단순한 제어 기능에서 발전하여 유무선 통신망을 통해 인터넷에 연결되며, 지능성을 갖춰 자체적으로 정보처리 능력을 보유하고 있다. 다시 말해, 오늘날의 일반적인 데스크 탑 컴퓨터에 버금가는 성능과 기능을 요구하고 있다[1].

본 논문에서는 오늘날 대부분의 내장형 시스템들이 필수적으로 요구하고 있는 실시간성과 안정성을 보장하기 위해 실시간 주기(Periodic) 태스크 셋을 위한 스케줄링(Scheduling) 기법에 관해 연구한다. 특히, 경성 실시간 시스템(Hard Real-Time System)에서는 주기 태스크들의 엄격한 마감시간(Deadline) 보장이 시스템의 성능을 좌우한다. 먼저 CPU 이용률(Utilization)이 높아 비율단조(RM : Rate-Monotonic) 기법으로는 마감시간을 보장 할 수 없는 주기 태스크들을 위해 확장된 스케줄 가능성(Schedulability)을 검사한다. 또한 수행할 태스크들의 공통 주기(L.C.M : Least Common Multiple)내에서 마감시간 우선(EDF : Earliest Deadline First)기법을 기반으로 마감시간 보장 수행패턴(Feasible Pattern)을 찾고, 이를 참조하여 우선순위를 고려하지 않고 태스크들을 강제 수행할 수 있는 비율단조 기반의 스케줄링 기법을 제안한다. 마감시간 우선 기법을 기반으로 생성된 패턴은 마감시간 우선 기법의 특성에 따라 CPU 이용률을 100%까지 가능하게 하며, 패턴을 참조하여 강제 수행함으로써 마감시간 우선정책이 갖는 실행시간 스케줄링 오버헤드를 없앨 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 실시간 운영체제의 요구사항과 기존의 스케줄링 기법에 대해 전반적으로 기술하고, 3장에서는 기존 방법의 분석과 제안된 방법에 대해 설명한다. 4장에서 실험 및 결과분석을 하고 5장에서 결론을 내린다.

II. 기존 실시간 스케줄링 기법

1. 실시간 운영체제의 요구사항

일반적으로 실시간성을 지원하기 위해 운영체제(Operating System)가 갖추어야 할 기본적인 사항은 다음과 같다[2],[3].

- o 실시간 프로세스를 위한 스케줄링
- o 선점형 스케줄링
- o 향상된 인터럽트 처리
- o 사용자 수준에서의 자원 통제

운영체제는 시스템내의 여러 프로세스들 중 실시간 프로세스와 일반 프로세스를 구분하여 가장 빨리 처리되어야 할 프로세스에게 자원을 할당해야 한다. 이를 위해 일반 프로세스와 실시간 프로세스를 구분할 수 있는 기능과 구분된 실시간 프로세스들 간의 우선순위 관리기능을 지원해야 한다. 현재, 많이 쓰이고 있는 방법으로는 고정 우선순위(Fixed Priority), 비율단조, 마감시간 우선 기법 등이 있다.

또한, 실시간 프로세스가 발생했거나 실행가능 상태가 되었을 때 빨리 수행가능하기 위해 선점형 스케줄링이 필요하다. 즉, 현재 수행중인 낮은 우선순위의 프로세스를 중단시키고 우선순위가 높은 실시간 프로세스가 수행가능 하도록 해야 한다. 외부의 인터럽트에 의해 기능을 수행해야 할 경우 이를 처리하기 위한 첫 코드가 수행되기까지의 시간인 인터럽트 지연 시간(Interrupt Latency Time)을 최소화해야 한다. 이는 실시간성 측정의 중요한 지표 중 하나이다. 마지막으로 고려해야 할 것은 하나의 운영체제 내에 모든 실시간 시스템의 기능을 넣을 수 없기 때문에 프로세스의 수행 보장을 위해 CPU 스케줄링을 위한 사용자 수준의 우선순위 지정과 같은 자원통제가 필요하다.

2. 비율단조 기법

비율단조 기법은 최적의 정적 알고리즘으로 독립적이라고 가정된 각 태스크를 수행 전 정적으로 우선순위를 부여받는 방법이다[4],[5]. 우선순위는 각 태스크의 주기를 기준으로 주기가 짧을수록 높은 우선순위가 주어진다. 다음은 Liu와 Layland에 의해 제안된 스케줄 가능성 검사 방법이다[6].

주기 태스크 집합 $t = t_1, t_2, \ldots, t_n$는 고정 우선순위 방식으로 스케줄링 되며, 우선순위 순으로 (t_1이 가장 높다) 정렬 되어있다고 가정한다. T_i는 t_i의 주기, C_i는 수

행시간(Completion Time), C_i / T_i는 t_i의 프로세서 이용률(Utilization, U_i)이라 하면, 모든 태스크들의 CPU 이용률은 다음과 같다. 여기서, $1 \leq i \leq n$ 이다.

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots\cdots + \frac{C_n}{T_n} \quad (1)$$

식 (1)의 이용률 U가 다음을 만족하면 태스크 집합 t의 모든 태스크들은 동시에 마감시간을 만족할 수 있다.

$$U \leq n(2^{\frac{1}{n}} - 1) \quad (2)$$

식 (2)에서 $n \rightarrow \infty$이면, $U = \ln 2$(약 0.693)이다. 즉, 주기를 기반으로 정적 우선순위를 부여하는 알고리즘에서최대 CPU 이용률은 약 0.693(69.3%)이 된다. 따라서 만약 태스크들의 CPU 이용률이 최대 값 보다 작으면 마감시간을 만족하지만 최대 값 보다 크고 1(100%) 보다 작으면 스케줄링 가능한지, 즉, 마감시간을 만족하는지를 판단 할 수 없다. 이 기법은 태스크 실행 전 우선순위가 할당되기 때문에 우선순위 재 계산을 위한 오버헤드가 없는 것이 장점이지만, CPU 이용률이 낮은 것이 단점이다.

3. 마감시간 우선 기법

마감시간 우선 기법은 동적 우선순위 기법으로서 최적의 방법이다[5],[6],[7]. 비율단조 기법과 달리 동적으로 우선순위를 할당하는 방식으로 태스크의 주기가 아닌 마감시간을 기준으로 한다. 임의의 실행시점에서 실행이 완료되지 않은 태스크들 중 가장 가까운 마감시간을 갖는 태스크에게 다음 우선순위를 할당한다. 즉, 마감시간이 현시점에서 가까울수록 높은 우선순위가 할당된다.

주기 태스크 집합 $t = t_1, t_2, \ldots, t_n$이 동적 우선순위 방식으로 스케줄링 된다고 가정 할 때, T_i는 t_i의 주기, C_i는 수행시간이라 하면, 다음 조건을 만족할 때 태스크들은 스케줄 가능하다고 할 수 있다. 여기서, $(1 \leq i \leq n)$ 이다.

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots\cdots + \frac{C_n}{T_n} \leq 1 \quad (3)$$

식 (3)에서 동적 우선순위 알고리즘은 CPU 이용률이 최대 1(100%)까지 가능함을 알 수 있다. 그러나 새로운 태스크를 할당 할 때마다 우선순위를 계산하기 때문에 실행시간 스케줄링 오버헤드가 크다는 단점이 있다.

III. 제안된 스케줄링 기법

본 장에서는 기존의 정적 우선순위 기반 스케줄링 기법의 동작에 대한 소개와 그 문제점을 보이고, 이를 보완하기 위해 본 논문에서 제안하는 기법에 대해 기술한다.

1. 기존 기법 분석

대부분의 실시간 운영체제는 선점형 다중처리를 지원한다. 즉, 태스크에 우선순위를 할당하고 준비된 태스크들 중에서 우선순위가 높은 태스크부터 실행하게 된다. 이런 스케줄링 기법은 태스크에 우선순위를 부여하는 방법이다. 종류로는 정적 우선순위, 동적 우선순위, 혼합형 우선순위 할당 기법이 있다[8]. 정적 우선순위 할당 기법은 태스크의 실행 전에 모든 태스크의 우선순위를 미리 정하는 방법으로 한 번 할당된 우선순위는 태스크가 끝날 때까지 변하지 않고 유지된다. 반면, 동적 우선순위 할당 기법은 태스크의 마감시간에 따라 실행시간에 동적으로 우선순위를 결정하는 방법이다[9].

혼합형 우선순위 할당 기법은 앞의 두 가지 방법을 혼합한 형태로서 태스크들의 수행시간, 주기 등에 따라 정적 혹은 동적으로 우선순위 할당 방법을 달리하여 두 가지 기법이 갖는 장점을 취하면서 단점을 최소화하는 본 논문에서 제안하는 기법이 이 범주에 속한다[10]. 정적 우선순위 할당 기법의 동작 방식을 살펴보기 위해 두 개의 실시간 태스크가 있는 시스템을 고려한다. 두 태스크는 모두 주기 태스크이고 각각의 주기와 최대 실행 시간은 표 1과 같으며 주기와 실행시간은 같다고 가정한다.

표 1. 태스크의 주기와 실행 시간

태스크	주기	최대 실행 시간
t_1	T_1 = 50	C_1 = 25
t_2	T_2 = 100	C_2 = 40

주기 태스크 t_1을 보면, 주기(T_1)가 50이고 최대 실행시간(C_1)은 25 이므로 CPU 이용률(U_1)은 50% (25 / 50)가 된다. 주기 태스크 t_2에서는 주기(T_2)가 100이고 최대 실행 시간(C_2)은 40 이므로 CPU 이용률(U_2)은 40% (40 / 100)가 된다. 따라서 총 이용률은 90%($U = U_1 + U_2$)이다. CPU 이용률이 100% 보다 작으므로, 두 태스크 셋의 실행은 충분히 가능하다. 하지만 표 1의 두 태스크 셋에는 다

음과 같은 두 가지 상황이 발생할 수 있고, 그 결과를 그림 1에서 보여준다.

- Case 1: t_1의 우선순위 > t_2의 우선순위
- Case 2: t_1의 우선순위 < t_2의 우선순위

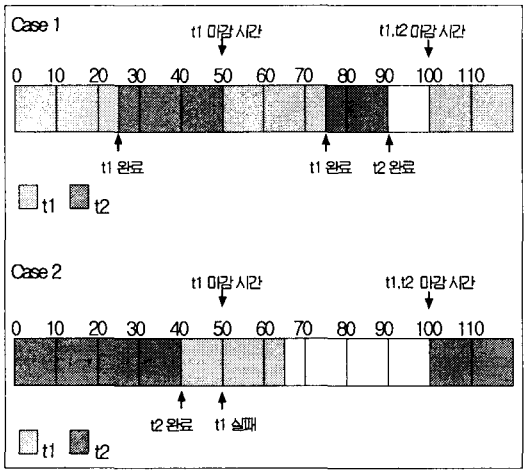


그림 1. 우선순위에 따른 다른 결과

Case 1의 경우 두 태스크 모두 마감시간을 만족하고 있지만, Case 2의 경우는 그렇지 못하다. 두 경우 모두 10%의 CPU 유휴시간(Idle Time)이 있지만, 우선순위에 따른 결과는 다를 수 있음을 보여준다.

비율단조 기법은 스케줄 가능성을 최대화하기 위해 고정 우선순위를 할당하는 기법으로 모든 태스크들이 자신의 마감 시간을 만족하게 되는지 검사하여 주기가 가장 짧은 태스크가 가장 높은 우선순위를 갖게 된다. 위의 경우에서 t_1의 주기가 t_2의 주기보다 짧기 때문에 비율단조 기법의 규칙에 따라 t_1의 우선순위가 더 높게 된다. 그림 1의 Case 1에서 이 경우를 볼 수 있으며 결과적으로 두 태스크 모두 마감시간을 만족하며, 정상적으로 실행된다는 것을 알 수 있다. 반면, Case 2의 결과를 보면 t_1의 마감시간을 만족하지 못하고 있다. 비율단조 기법은 정적 우선순위 할당 기법 중 최적으로 알려져 있지만, 식 (2) 에서와 같이 CPU 이용률에 한계가 있다. 예로서 그림 2의 경우를 살펴본다. 태스크들의 실행 시간과 주기는 표 2와 같다.

표 2. 태스크의 주기와 실행 시간

태스크	주기	최대 실행 시간
t_1	$T_1 = 50$	$C_1 = 25$
t_2	$T_2 = 75$	$C_2 = 30$

단지 t_2의 주기와 최대 실행 시간만을 변경하였고, CPU 이용률은 표 1의 경우와 동일한 90%가 된다. 표 2의 두 태스크 셋에도 앞의 경우와 같이 다음과 같은 두 가지 상황을 고려해 본다.

- Case 1: t_1의 우선순위 > t_2의 우선순위
- Case 2: t_1의 우선순위 < t_2의 우선순위

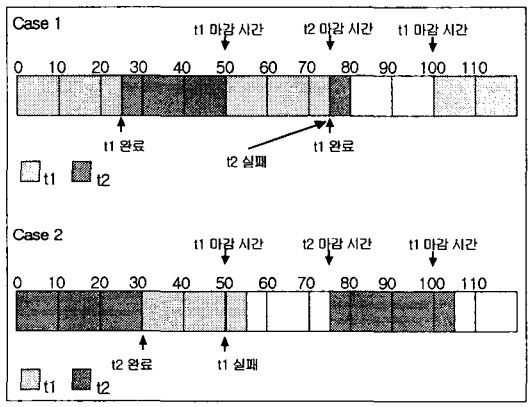


그림 2. CPU 이용률에 대한 한계

두 가지 경우의 결과를 그림 2에서 보여주고 있다. 앞의 예에서와 동일한 CPU 이용률에도 불구하고, 두 가지 경우 모두 실패함을 알 수 있다. 즉, 비율단조 기법에서의 주기와 수행 시간을 고려한 CPU 이용률에 따른 우선순위 할당은 그 이용률이 식 (2)를 만족한다 하더라도, 그림 2에서와 같이 태스크들의 우선순위에 상관없이 실패할 수도 있음을 보여준다. 이 경우, 동적 우선순위 할당 기법만이 문제를 해결 할 수 있다.

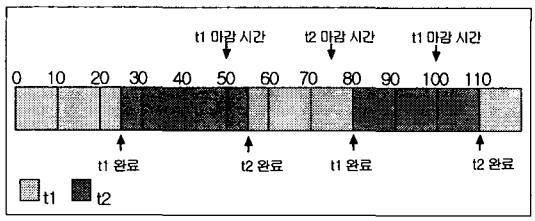


그림 3. 동적 우선순위 할당에 따른 결과

그림 3은 동적 우선순위 할당에 따른 결과를 보여준다. 그러나 시스템의 복잡도를 증가시키고 태스크들의 실행시간에 동적으로 우선순위가 결정되기 때문에 실행시간 오버헤드가 발생한다.

그림 1의 Case 1과 같이 특정 태스크 셋의 총 CPU 이용률이 식 (2)를 만족하지 못하고 고정된 우선순위를 갖고 스케줄 가능한 경우도 있지만, 그림 2의 두 경우와 같이 모두 실패 할 수도 있음을 확인할 수 있다. 또한 그림 3에서는 우선순위 할당 기법의 변화를 통해 문제를 해결할 수 있음을 살펴 보았다. 따라서 스케줄 가능성은 태스크 셋 내의 각각의 태스크들이 갖는 주기와 수행시간의 특성에 의존적이며, 마감시간 보장을 위한 완벽한 하나의 방법은 존재하지 않는다. 즉, 시스템의 상황에 맞는 적절한 스케줄 가능성 검사와 우선순위 할당 기법이 필요하다.

2. 제안된 기법

본 절에서는 앞서 살펴본 실시간 주기 태스크를 위한 관련 연구와 기존의 방법들에 대한 분석을 통해 알아본 문제점과 개선 방안을 종합하여, 본 논문에서 제안하고 있는 개선된 우선순위 할당 기법에 대해 기술한다. 먼저, 3.1절의 기존방법에 대해 다음과 같은 몇 가지 결론을 얻을 수 있다.

o 비율단조 기법에 의한 고정 우선순위 할당이 언제나 태스크 셋의 마감시간을 모두 보장하는 것은 아니다.

o CPU 총 이용률이 식 (2)를 만족하면, 모든 태스크들은 그들의 마감시간을 보장 받을 수 있다.

o CPU 총 이용률이 식 (2)를 만족하지 못하면, 그 태스크 셋이 스케줄 가능한지를 결정하기 위해 태스크 셋의 특성을 분석하는 추가적인 작업이 필요하다.

o CPU 총 이용률이 식 (2)를 만족하지 못하면, 최대 CPU 이용률과 각 태스크들의 마감시간 보장을 위한 우선순위 할당 기법의 변화를 통해 해결 할 수 있다.

o 고정 우선순위 할당으로 100% CPU 이용률을 얻기 위해서는 모든 태스크들이 정수배의 주기를 갖도록 해야한다. 이것은 모든 태스크들의 주기는 태스크 셋 내의 가장 짧은 주기의 정수배가 되어야함을 의미한다.

즉, 본 논문에서는 그림 1, 2에서 실패한 경우에 대한 해결을 그림 3과 같이 CPU 이용률을 식 (2)를 기준으로 정적, 동적 우선순위 할당 기법의 혼용을 통해 해결하고자 한다. 태스크 셋의 CPU 총 이용률을 계산하여 식 (2)의 조건에 따라 정적인 기법과 동적인 기법을 혼용하여 각 태스크의 우선순위를 결정한다. 물론, CPU 총 이용률이 1(100%)을 넘을 수 없다.

본 논문에서 제안하는 기법은 다음과 같은 제한 사항들을 갖는다.

o 태스크 수행에 필요한 비용을 제외한 모든 비용은 무시한다.

o 주기 태스크의 주기는 상수이다.

o 수행 중 인터럽트는 없고, 마감시간은 주기보다 크거나 같다고 가정한다.

주기 태스크 t_n 도착, $(n > 1)$

$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ /* 이용률 계산 */

```
if( U ≤ 1) {
  if( U ≤ n(2^(1/n) − 1)) {
    비율단조 기반 스케줄링
  }else {
    가능성 검사 대상 태스크들의 주기 값의 L.C.M 계산
                                  /* 공통 반복 주기 */
    T = ( U * L.C.M) / 100   /* 할당 가능한 총 시간 */

    while(T) {
       마감시간 우선 정책을 이용한 태스크 선택
       if(태스크 선택 불가능)
          스케쥴 불가능, 보류

       T -= 선택된 태스크의 수행 가능 시간 량
            /* 수행 가능 시간 량 계산 */
    }
    테이블에 저장된 태스크 번호와 시간 량을 이용하여
    태스크 수행
  }
}else {
   스케줄 불가능, 보류
}
```

그림 4. 제안한 스케줄링 알고리즘

그림 4는 본 논문에서 제안한 확장된 스케줄 가능성 검사를 통한 혼용 방식의 우선 순위 할당방법 알고리즘을 나타낸다. 먼저, 임의의 시간에 주기 태스크 t_i가 도착하면, t_i와 동시에 스케줄링이 되어야 할 이전 태스크들과 t_i의 CPU 총 이용률 U를 구한다. 계산된 U가 1보다 클 경우, t_i를 포함한 모든 태스크들의 U가 100%를 넘는다는 것을 의미하므로, t_i는 스케줄링 불가능을 의미한다. 그러나 U가 식 (2)를 만족할 경우, 완벽하게 비율단조 기법으로 스케줄링 가능

하다는 것을 의미한다. 따라서 U가 1 보다 작고 식 (2)를 만족하지 않는 경우 비율단조 기법으로는 마감시간을 보장받지 못한다.

본 논문에서는 위의 경우를 위해 스케줄 가능성 검사 기법을 확장한다. U가 1 보다 작기 때문에 마감시간 우선 기법을 이용하면 보장 가능성이 커질 수 있다[11]. 따라서 마감시간 우선 기법을 이용하여, 마감시간을 보장할 수 있는 수행패턴과 수행가능 시간 량을 계산한다. 그러나 마감시간 우선 기법이 실패한다면 역시 스케줄 불가능하게 된다.

계산된 시간(U)을 실제 L.C.M까지의 실제 시간 량으로 바꾸고, 찾아진 태스크와 시간 량의 쌍을 테이블에 유지하면서 남은 시간 량이 0이 될 때까지 반복한다. 탐색이 한번 끝나면 각 태스크들의 L.C.M을 공통주기로 하여 반복할 수 있는 마감시간 보장 수행패턴이 생성된다[12]. 생성된 수행 패턴을 우선순위로 적용하여 순차적으로 계산된 시간만큼만 수행한 후, 테이블에서 참조되고 있는 다음 태스크가 자동으로 선택되어 실행된다. 수행패턴의 마지막까지 실행하고 나면, 마지막 태스크의 종료시간부터 첫 번째 공통 주기까지의 시간은 유휴시간이 된다.

수행패턴의 마지막 태스크가 종료되고 공통주기가 끝나면, 그 다음 공통 주기 내에서 수행패턴을 그대로 다시 적용 할 수 있기 때문에, 또 다른 태스크가 도착하여 수행패턴을 다시 계산해야할 경우가 아니면, 더 이상의 스케줄링 오버헤드는 발생하지 않는다.

표 3과 같은 태스크 셋들이 있을 때 t_1, t_2, t_3의 주기와 수행시간은 각각 (50, 25), (75, 30), (150, 5)이고, CPU 총 이용률(U)은 약 93.3%가 된다. 또한 공통주기(L.C.M)는 150임을 알 수 있다.

표 3. 주기 태스크의 주기와 실행 시간

태스크	주기	최대 실행 시간
t_1	$T_1 = 50$	$C_1 = 25$
t_2	$T_2 = 75$	$C_2 = 30$
t_3	$T_3 = 150$	$C_3 = 5$

CPU 이용률이 1 보다 작지만, 식 (2)를 만족하지 않으므로 비율단조 기반의 우선순위 할당 기법을 통한 태스크의 수행은 실패한다. 하지만 동적 우선순위 할당 기법을 통해 해결할 수 있으며 태스크 t_3를 제외하면 나머지 태스크들의 결과는 그림 3과 동일하다. 태스크 t_3의 CPU 이용률이 3.3%이므로 10%의 유휴시간에 실행 가능함을 알 수 있다.

표 4. 생성된 수행 패턴

수행 시간	수행 태스크	도착 시간	마감 시간
0 ~ 25	1	0	50
25 ~ 55	2	0	75
55 ~ 80	1	50	100
80 ~ 110	2	75	150
110 ~ 135	1	100	150
135 ~ 140	3	0	150
140 ~ 150	유휴시간 (10)		

따라서, 표 3의 태스크 셋을 표 4와 같이 제안한 기법에 따라 마감시간 우선 기법 기반의 우선순위 할당 기법과 동일한 결과의 수행패턴을 미리 생성한다. 표 4에서 주어진 태스크 셋의 공통주기가 150이고 CPU 총 이용률이 약 93.3% 이므로, 한 주기 내에서의 유휴시간(150 * 0.066)은 10이 됨을 알 수 있다.

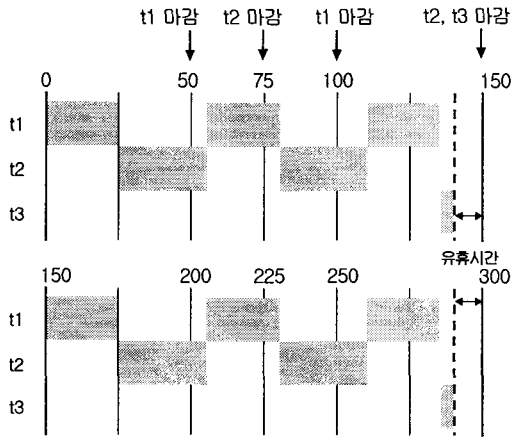


그림 5. 생성된 수행패턴을 바탕으로 한 결과

또한 그림 5에서 생성된 수행패턴을 기반으로 한 수행결과를 볼 수 있는데, 수행패턴이 공통주기 150을 기준으로 동일하게 반복된다. 앞에서 언급한 모든 경우에서, 식 (2)를 만족하지 못하고 있기 때문에 비율단조 기반의 정적 우선순위 할당 기법을 통한 우선순위 할당은 전체 태스크 셋 내의 모든

태스크들에 대해 마감시간을 보장 할 수 없다. 그러므로 CPU 총 이용률을 기준으로 하는 스케줄 가능성 검사를 통해, 식 (2)의 조건에 따라 비율단조 기법과 마감시간 우선 기법을 혼용하여 해결할 수 있다.

IV. 실험 및 결과분석

본 장에서는 제안한 스케줄링 기법에 대한 성능 측정을 위해 사용된 실험 환경에 대해 설명하고, 실험 방법과 평가 결과 및 분석에 대해서 기술한다.

1. 실험 평가 방법

성능 평가를 위해 운영체제는 RTLinux Ver. 3.1과 Linux Ver. 2.4.4를 사용하였으며, 하드웨어는 Pentium III 733MHz, 256MB RAM을 갖춘 PC를 사용한다.

표 5의 태스크 셋처럼, 식 (2)의 조건을 만족하지 못하는 태스크 셋에 대해 제안된 기법으로 스케줄 가능성 검사를 수행하여, 최대한 CPU 이용률을 높이면서 마감시간을 만족시킨다. 식 (2)를 만족하는 경우는 비율단조 기법을 통한 우선순위 할당이 가능하므로, 비율단조 기법을 그대로 적용한다. 또한 정적 우선순위 할당을 통한 실시간 스케줄링 오버헤드의 최소화를 목표로 하기 때문에, 마감시간 우선 기법을 통한 우선순위 할당은 고려 대상이 아니며, CPU 이용률에 근거하여 선택적으로 적용한다.

표 5. 실험 태스크 셋

태스크	주기	최대 실행 시간	우선 순위
t_1	$T_1 = 50$	$C_1 = 10$	1
t_2	$T_2 = 80$	$C_2 = 20$	2
t_3	$T_3 = 100$	$C_3 = 30$	3
t_4	$T_4 = 120$	$C_4 = 12$	4

표 5의 태스크 셋을 실험 대상으로 하여 조건이 식 (2)를 만족하지 않으므로 태스크 셋의 수행 전에 마감시간을 보장해 줄 수 있는 수행패턴을 생성하여 이를 참조한 우선순위 할당을 통해 모든 태스크의 마감시간을 보장하도록 한다.

2. 실험 대상 범위

본 실험의 대상이 되는 태스크 셋은 식 (2)를 만족하지 못하여 CPU 이용률이 그림 6의 빗금 친 부분에 해당하며 본 논문에서 제안한 기법이 적용되는 범위이기도 하다. 그림 6은 식 (2)에 따른 정적 우선순위 할당 기법인, 비율단조 기법의 태스크 수와 CPU 총 이용률간의 관계를 나타낸다. 이는 태스크 수와 관계없이 최소 69.3%의 CPU 이용률이 보장됨을 알 수 있다.

본 논문에서는 인터럽트 지연시간, 문맥 교환과 같은 외적인 비용은 고려하지 않으며 태스크의 수행 비용만을 고려한다.

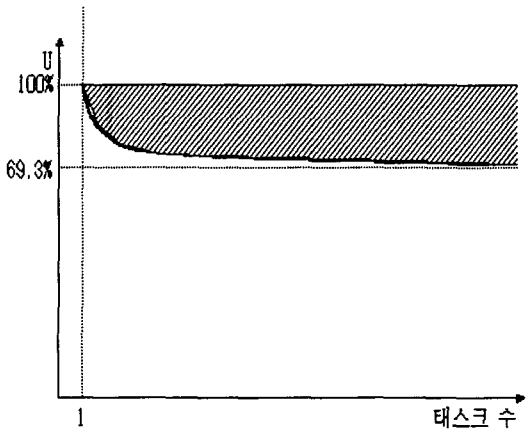


그림 6. 제안된 기법의 적용 범위

3. 결과 분석

표 5를 대상으로 태스크 t_1, t_2, t_3의 서브셋 총 CPU 이용률(U)은 0.75(75%)로 식 (2)를 만족한다. 즉, 비율단조 기법으로 할당된 우선순위에 따라 수행 가능함을 의미한다. 표 6은 태스크 t_1, t_2, t_3만을 고려한 서브셋의 수행순서를 나타낸다.

그림 7은 서브셋의 수행 결과로서 서브셋 태스크 t_1, t_2, t_3은 식 (2)를 만족하기 때문에 모든 태스크가 마감시간을 지키면서 수행한다. 그러나 태스크 t_4가 포함된 원래의 태스크 셋을 고려하면 총 CPU 이용률(U)은 0.85(85%)가 된다. 따라서 식 (2)를 만족하지 못하고 식 (3)만을 만족하여 비율단조 기반의 우선순위 할당을 통한 수행 결과는 그림 8과 같다.

표 6. 서브셋의 수행 순서

수행 시간	수행 태스크	도착 시간	마감 시간
0 ~ 10	1	0	50
10 ~ 30	2	0	80
30 ~ 50	3	0	100
50 ~ 60	1	50	100
60 ~ 70	3	0	100
70 ~ 80	-	-	-
80 ~ 100	2	80	160
100 ~ 110	1	100	150
110 ~ 140	3	100	200
140 ~ 150	-	-	-
150 ~ 160	1	150	200
160 ~ 180	2	160	240
180 ~ 200	-	-	-
200 ~ 210	1	200	250
:	:	:	:

['-' : 유휴시간]

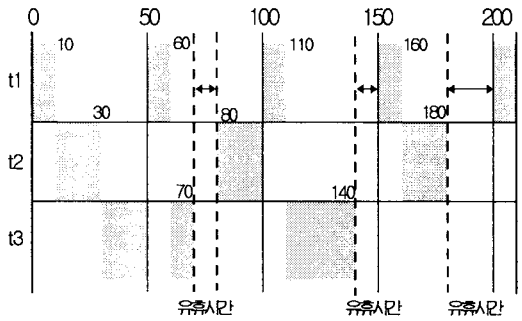


그림 7. 서브셋의 수행 결과

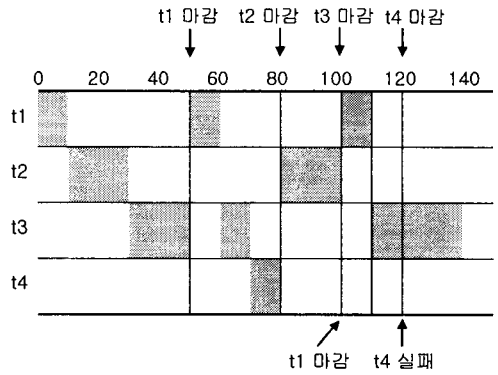


그림 8. 비율단조 기법에 따른 결과

즉, 15%의 CPU 유휴시간에도 불구하고 t_4가 첫 번째 마감시간(120)을 놓치게 된다. 이 경우 동적 우선 순위 할당 방법을 통해 해결해야 한다. 그림 9는 표 5의 태스크 셋에 대한 동적 우선순위 할당 기법인 마감시간 우선 기법을 사용한 결과이다. 태스크들의 실행시간에 우선순위를 동적으로 할당함으로써 그림 8의 문제점을 해결할 수 있다. 이는 t_4의 추가로 인한 태스크들의 마감시간을 보장받지 못하기 때문에, 제안한 마감 시간 우선 기법을 적용하면 각 태스크들은 마감시간을 보장받기 때문이다.

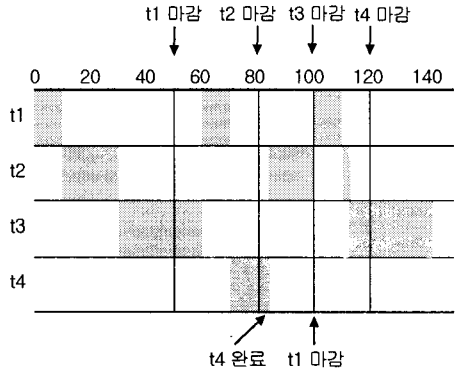


그림 9. 마감시간 우선 기법에 따른 결과

그림 5에서 미리 생성된 수행 패턴은 각 태스크 주기의 L.C.M을 공통 주기로 반복되며, 이 수행패턴을 참조하여 수행되기 때문에 새로운 태스크가 셋에 추가되지 않으면, 스케줄러가 호출되더라도 우선순위 재 계산 작업이 없으므로 스케줄링 오버헤드가 없다.

```
// 수행패턴 생성
for( 현재 시간 <= L.C.M ) {
    for( 태스크 수 ) {
        가장 짧은 주기를 갖는 태스크를 찾는다.
    }

    if( 태스크 ) {
        선택된 태스크의 주기와 수행시간을 재 계산한다.
    } else {
        실행 가능한 태스크가 없으면 유휴시간을 갖는다.
    }
}
```

그림 10. 수행패턴 생성 방법

표 7. 생성된 수행패턴

수행 시간	태스크	도착시간	마감시간	수행 시간	태스크	도착시간	마감시간
0 ~ 10	1	0	50	602 ~ 612	1	600	650
10 ~ 30	2	0	80	612 ~ 642	3	600	700
30 ~ 60	3	0	100	642 ~ 654	4	600	720
60 ~ 70	1	50	100	654 ~ 664	1	650	700
70 ~ 82	4	0	120	664 ~ 684	2	640	720
82 ~ 102	2	80	160	684 ~ 700	-	-	-
102 ~ 112	1	100	150	700 ~ 710	1	700	750
112 ~ 142	3	100	200	710 ~ 740	3	700	800
142 ~ 154	4	120	240	740 ~ 760	2	720	800
154 ~ 164	1	150	200	760 ~ 770	1	750	800
164 ~ 184	2	160	240	770 ~ 782	4	720	840
184 ~ 200	-	-	-	782 ~ 800	-	-	-
200 ~ 210	1	200	250	800 ~ 810	1	800	850
210 ~ 240	3	200	300	810 ~ 830	2	800	880
240 ~ 260	2	240	320	830 ~ 860	3	800	900
260 ~ 270	1	250	300	860 ~ 870	1	850	900
270 ~ 300	-	-	-	870 ~ 900	-	-	-
300 ~ 310	1	300	350	900 ~ 910	1	900	950
310 ~ 322	4	240	360	910 ~ 922	4	840	960
322 ~ 352	3	300	400	922 ~ 942	2	880	960
352 ~ 372	2	320	400	942 ~ 972	3	900	1000
372 ~ 382	1	350	400	972 ~ 982	1	950	1000
382 ~ 400	-	-	-	982 ~ 1002	2	960	1040
400 ~ 410	1	400	450	1002 ~ 1012	1	1000	1050
410 ~ 422	4	360	480	1012 ~ 1024	4	960	1080
422 ~ 442	2	400	480	1024 ~ 1054	3	1000	1100
442 ~ 472	3	400	500	1054 ~ 1064	1	1050	1100
472 ~ 482	1	450	500	1064 ~ 1084	2	1040	1120
482 ~ 500	-	-	-	1084 ~ 1096	4	1080	1200
500 ~ 510	1	500	550	1096 ~ 1100	-	-	-
510 ~ 530	2	480	560	1100 ~ 1110	1	1100	1150
530 ~ 542	4	480	600	1110 ~ 1140	3	1100	1200
542 ~ 572	3	500	600	1140 ~ 1160	2	1120	1200
572 ~ 582	1	550	600	1160 ~ 1170	1	1150	1200
582 ~ 602	2	560	640	1170 ~ 1200	-	-	-
				1200 ~ 1210	1	1200	1250

그림 10은 수행패턴을 생성하는 방법에 대해 설명하고 있다. 현재 시간부터 공통주기까지 모든 태스크들의 주기와 마감시간을 계산하여 스케줄 가능한 수행패턴을 찾는다. 표 5의 모든 태스크 셋에 대해 생성된 수행패턴의 결과는 표 7과 같다. 공통 주기는 1200이고, CPU 총 유휴시간은 180이다. 실제 생성된 수행패턴내의 유휴시간(180)의 합과 CPU 이용률을 근거로 계산한 유휴시간(1200 * 0.15)이 일치하고 있는 것을 알 수 있다.

표 7의 수행패턴에 따른 각 태스크 별 수행 결과를 그림 11에서 나타내고 있다. 각각의 수행 결과에서 모든 태스크가 주기에 따른 마감시간 내에서 수행이 완료된다. 또한, 그 수행패턴이 다음 공통 주기(2400)까지 그대로 적용 될 수 있음을 알 수 있다. 표 5의 태스크 셋의 이용률은 0.85로, 식 (2)를 만족하기 위한 0.75 보다 크므로 정적 우선순위 할당 기법으로는 그림 8과 같이 모든 태스크가 마감시간을 만족하지 못하였으나, 그림 9에서 동적 우선순위 할당 기법을 통하여 가능함을 알 수 있다. 그러나 동적으로 우선순위를 할당하는

방법의 가장 큰 단점은 실행시간 오버헤드다. 새로운 태스크가 시작할 때마다 우선순위를 재 계산해야 하기 때문이다.

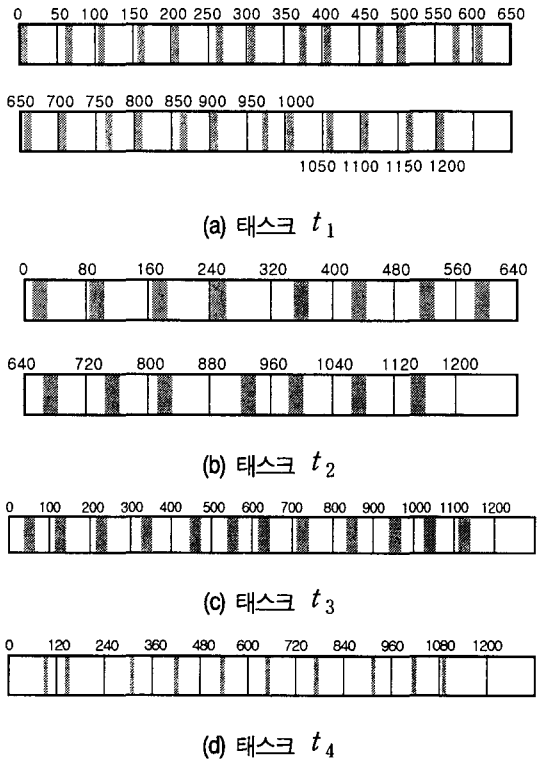


그림 11. 수행패턴에 의한 태스크들의 수행 결과

본 논문에서는 표 5와 같이 식 (2)를 만족하지 못하는 태스크 셋의 경우처럼, 정적 우선순위 할당 기법으로는 모든 태스크들의 마감시간을 보장할 수 없을 때 마감시간 우선 기법을 사용하여, 스케줄 가능한 패턴을 태스크 셋의 수행 전에 미리 생성하여 실행시간 오버헤드를 없애고, 동시에 CPU 이용률의 변화 없이 모든 태스크들의 마감시간을 보장할 수 있음을 확인하였다.

V. 결론

본 논문에서는 비율단조 기법을 기반으로 하는 정적 우선순위 할당 알고리즘에서, 좀 더 정확하고 효율적인 스케줄을 하기 위해 확장된 스케줄 가능성 검사를 수행하는 스케줄러를 설계하고 구현하였다. 비율단조 기법으로 마감시간을 보장할 수 없는 태스크들을 위해 미리 수행 패턴을 생성함으로써, 정적 우선순위 알고리즘이 갖는 정확성과 신속함을 유지하면서 동적 우선순위 알고리즘이 갖는 높은 CPU 이용률을 얻을 수 있다. 또한, 동적 우선순위 알고리즘이 갖는 단점인 실행시간 오버헤드를 없앨 수 있었다.

태스크 셋 내의 각 태스크 주기의 L.C.M 값을 이용하여 공통주기를 찾아 반복 수행패턴을 찾음으로써, 스케줄링 오버헤드를 줄이고 마감시간 보장을 통한 실시간성을 높일 수 있었지만, 공통주기의 길이가 길어지고 태스크의 수가 많아질수록 오버헤드가 증가하게 되는 단점을 갖고 있다. 또한, 스케줄러의 실질적인 적용에 있어서 CPU 이외의 다른 시스템 자원에 대한 고려를 하지 않았으며, 일정한 특성을 갖는 태스크만을 고려하고 있다. 따라서 실제 예측 가능한 시스템을 구성하기 위해서는 다양한 실제 환경에서의 여러 가지 태스크들을 적용한 성능평가를 통해 검증이 이루어져야 할 것이다.

참 고 문 헌

[1] S. Hong, "Coping with Embedded Software Crisis using Real-Time Operating Systems and Embedded Middleware," Invited for presentation at IEEE Asian Pacific ASIC Conference. Cheju, Korea, 2000.

[2] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach," Proceedings of IEEE Workshop on Real-Time Operating Systems and Software, pp. 133-137, 1991.

[3] Liu. and Jane W.S., "Real-Time systems," United States of America, Prentice-Hall, pp. 26-34, 2000.

[4] S. Baruah, A. Mok and L. Rosier, "Preemptively scheduling hard real-time sporadic tasks on one processor," IEEE Real-Time Systems Symposium, pp. 182-19, 1990.

[5] J. Lehoczky, Lui Sha. and Ye Ding, "The Rate Monotonic Scheduling Algorithm : Exact Characterization And Average Case Behavior," Proc. of IEEE Real-Time Systems Symposium, pp. 166-171, 1989.

[6] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in hard real-time environment," Journal of ACM, pp. 46-61, 1973.

[7] J. A. Stankovic, M. Spuri, K. Ramamritham and G.

Buttazzo, "Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms," Kluwer Academic Publishers, 0-7923-8269-2, 1998.

[8] C. Hollabaugh, "Embedded Linux Hardware, Sofrware, and Interfacing," Indianapolis, Addison Wesley, pp. 8-12, 2002.

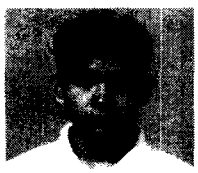
[9] http://www.embeddedconference.co.kr/news/16/16_6.htm

[10] Jack G. Ganssle and Michael Barr, "Embedded Systems Dictionaryed," CMP Books, 2003.

[11] H. Chetto and M.Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm," IEEE Trans On Sofrware Eng., pp. 1216-1269, 1989.

[12] I. Ripoll, A. Crespo and A. Mok, "Improvement in feasibility testing for real-time tasks," Journal of Real-Time Systems, pp. 19-40, 1996.

신 동 헌(Dong-Hern Shin) 정회원



2001년 2월 : 금오공과대학교 컴퓨터공학과 (공학사)
2004년 2월 : 금오공과대학교 컴퓨터공학과 (공학석사)

<관심분야> : 운영체제, 임베디드 시스템 등

조 수 현(Soo-Hyun Cho) 정회원

2000년 2월 : 금오공과대학교 컴퓨터공학과(공학사)
2002년 2월 : 금오공과대학교 컴퓨터공학과(공학석사)
2002년 3월 ~ 현재 : 금오공과대학교 컴퓨터공학과 박사과정

<관심분야> : 병렬/분산처리, 이동 에이전트, Cluster Computing, Grid Computing

김 영 학(Young-Hak Kim) 정회원

1984년 2월 : 금오공과대학교 전자공학과(공학사)
1989년 2월 : 서강대학교 전자계산학과 (공학석사)
1997년 2월 : 서강대학교 전자계산학과 (공학박사)
1989년 ~ 1997년 : 해군사관학교 전산과학과 교수
1998년 ~ 1999년 : 여수대학교 멀티미디어학부 교수
1999년 ~ 현재 : 금오공과대학교 컴퓨터공학부 교수

<관심분야> : 병렬 알고리즘, 분산 및 병렬처리 등

김 태 형(Tae-Hyong Kim) 정회원

1992년 2월 : 연세대학교 전자공학과 (공학사)
1995년 8월 : 연세대학교 전자공학과 (공학석사)
2001년 2월 : 연세대학교 전기전자공학 (공학박사)
2002년 : University of Ottawa (Post-Doc.)
현재 : 금오공과대학교 컴퓨터공학부 전임강사

<관심분야> : 컴퓨터네트워크, 프로토콜 공학