유니폼 멀티프로세서 환경에서 단순 주기성 태스크를 위한 최적 RM 스케줄링

Optimal RM Scheduling for Simply Periodic Tasks on Uniform Multiprocessors

정명조*, 조문행*, 김주만**, 이철훈*

충남대학교 컴퓨터공학과*, 부산대학교 바이오정보전자공학과**

Myoung-Jo Jung(mjjung@cnu.ac.kr)*, Moon-Haeng Cho(root4567@cnu.ac.kr)*, Joo-Man Kim(joomkim@pusan.ac.kr)**, Cheol-Hoon Lee(clee@cnu.ac.kr)*

요약

본 논문에서는 유니폼 멀티프로세서 환경에서 단순 주기성 태스크 시스템을 성공적으로 스케줄 할 수 있는 알고리즘을 제안한다. 멀티프로세서 환경에서 주기성 태스크를 스케줄하기 위한 파티션드(partitioned) 스케줄링 알고리즘은 bin-packing 문제와 같은 문제로써 해결하는 게 불가능하다고 알려져 있다. 본 논문에서는 "task-splitting"기법을 이용하여 단순 주기성 태스크 시스템을 다른 단순 주기성 태스크 시스템으로 변환하는 글로벌(global) 스케줄링 알고리즘을 제시하고, 변환과정을 거친 단순 주기성 태스크 시스템은 유니폼 멀티프로세서에서 파티션드 스케줄링 알고리즘에 의해 성공적으로 스케줄 된다. 그리고 유니폼 멀티프로세서 환경에서 제안한 알고리즘이 이론적으로 최대 이용률 범위(utilization bound)까지 성공적으로 스케줄 할 수 있음을 증명한다.

■ 중심어 : | 실시간 스케줄링 | 단순 주기성 태스크 | Rate-monotonic | 유니폼 멀티프로세서 |

Abstract

The problem of scheduling simply periodic task systems upon a uniform multiprocessor is considered. Partitioning of periodic task systems requires solving the bin-packing problem, which is known to be intractable (NP-hard in the strong sense). This paper presents a global scheduling algorithm which transforms a given simply periodic task system into another using a "task-splitting" technique. Each transformed simply periodic task system is guaranteed to be successfully scheduled upon any uniform multiprocessor using a partitioned scheduling algorithm. It is proven that the proposed algorithm achieves the theoretical maximum utilization bound upon any uniform multiprocessor platform.

■ keyword : | Real-time Scheduling | Simply Periodic Tasks | Rate-monotonic | Uniform Multiprocessors |

I. 서 론

내장형 시스템에서 사용되는 하드웨어의 발전은 내장형 시스템이 가진 자원 제약을 극복할 수 있는 계기를 마련하도록 하였다. 기존의 휴대전화, 가전제품 등에서 사용되던 내장형 소프트웨어들은 하드웨어 성능의

* 본 연구는 지식경제부의 IT R&D 지원으로 수행되었습니다.[2008-F-048, 웨어러블 컴페니언 개발 사업].

접수번호 : #090901-005

접수일자 : 2009년 09월 01일

심사완료일 : 2009년 11월 05일

교신저자 : 이철훈, e-mail : clee@cnu.ac.kr

발전에 따라 더욱 많은 기능을 추가하게 되었고, 이러한 기능 확장은 내장형 시스템에서 사용되는 소프트웨어에도 큰 변화를 가져왔다. 내장형 시스템이 다양한 기능을 제공하기 위하여, 더 높은 컴퓨터 계산 능력을 필요로 함에 따라, 현재 내장형 시스템에는 확장성과 융통성이 뛰어난 멀티프로세서(multiprocessors) 플랫폼이 점차 사용되고 있는 추세이다. 멀티프로세서 구조는 유니프로세서(uniprocessor)구조보다 훨씬 향상된 동시성(concurrency), 시스템 밀도(system density), 와트당 성능(performance per watt)을 제공한다. 그러나 실시간 태스크(task) 스케줄링(scheduling) 면에서 멀티프로세서 구조는 유니프로세서 구조보다 훨씬 많은 복잡도(complexity)를 가지고 있다. 그 동안 많은 유니프로세서 스케줄링 알고리즘이 연구되어 왔고, 특히 EDF(earliest deadline first) 스케줄링 알고리즘[12]과 LL(Least Laxity) 스케줄링 알고리즘[14]은 최적(optimal)의 알고리즘이다. 그렇지만 이 2개의 알고리즘은 멀티프로세서 구조에서는 최적이 아니기 때문에[7], 멀티프로세서 구조를 위한 스케줄링 알고리즘이 활발히 연구되고 있다. 최근까지 연구된 멀티프로세서 구조를 위한 스케줄링 알고리즘 중에, 특히 Pfair 스케줄링 알고리즘[3]과 EKG 스케줄링 알고리즘[1], LLREF 스케줄링 알고리즘[5]은 멀티프로세서 구조에서 최적(optimal)임이 증명되었다. 그러나 위 3개의 알고리즘은 심각한 런타임 오버헤드(run-time overhead)를 가지며, 계산 복잡도(computation complexities)가 매우 크다는 단점이 있다. 그리고 100%의 최대 이용율 범위(utilization bound)까지 성공적인(데드라인을 어기지 않는) 스케줄을 하는 최적의 알고리즘은 아니지만 런타임 오버헤드와 계산 복잡도를 크게 낮춘 여러 스케줄링 알고리즘이 연구되었다[6][11].

위에서 소개한 알고리즘들은 동적 우선순위(dynamic-priority) 스케줄링 알고리즘에 속한다. 일반적으로 동적 우선순위 스케줄링 알고리즘은 단점으로 태스크가 한번 데드라인(deadline)을 못 지키게 되면 뒤에 오는 다른 작업(job)들도 데드라인을 어기게 되는 연쇄 누적 효과(domino effect) 특성과 주기적 실행에서 변화에 민감한 특성(내장형 시스템에 바람직하지 않는)을 가지고 있다. 동적 우선순위 스케줄링의 장점은 높은 이용률을 보인다는 점이지만, 멀티프로세서 환경에서는 높은 이용률을 보장하지 못한다. 반면에 정적 우선순위(static-priority) 스케줄링 알고리즘 연쇄 누적 효과와 주기적 실행에서 변화에 민감한 특성은 없으나, 동적 우선순위 알고리즘보다 달성할 수 있는 이용률이 상대적으로 낮다. 최근 Andersson 등[2]은 멀티프로세서에서 정적 우선순위 알고리즘이 최대 이용률 범위를 50%까지 달성할 수 있음을 증명한 알고리즘을 제시하였다. 그러나 이 알고리즘은 Pfair 스케줄링 알고리즘 특징을 가지고 있어서 태스크 선점(preemption) 빈도가 너무 잦다. Pfair 스케줄링 알고리즘에서는 태스크들이 여러 개의 작은 조각으로 분해되어 실행될 수 있는 특성이 있으나 이 특성은 매우 잦은 스케줄링과 프로세서 간 태스크 이동(migration)이 발생한다. Baruah와 Goossens[4]은 멀리프로세서 플랫폼에서 글로벌 RM(global Rate-monotonic) 스케줄링에 의해 주어진 주기성(periodic) 태스크 시스템의 스케줄 가능 여부에 대한 충분조건을 제시하였다. 그리고 m개의 동종의(homogeneous) 멀티프로세서 플랫폼 상에 주기성 태스크 시스템의 각 태스크의 이용률(utilization)이 1/3 이하이고 모든 태스크들의 이용률 합계가 m/3 이하이면 RM 스케줄링 알고리즘이 성공적으로 스케줄 할 수 있다고 증명했다. 또한 Kato 등[10]은 동종의 멀티프로세서에서 RMDP 알고리즘을 제안하였다. 이 알고리즘도 역시 50%의 최대 이용률 범위를 가지며, 대신 태스크 선점 빈도가 Pfair 계열 알고리즘에 비해 매우 적다. 그러나 RMDP의 우선순위 정책을 살펴보면 엄밀한 의미에서 정적 우선순위 스케줄링 알고리즘이라고 할 수 없다.

멀티프로세서 환경에서 주기성 태스크 시스템을 스케줄하는 알고리즘들은 크게 두 가지로 분류할 수 있는데, 첫 번째는 한 태스크에 의해 생성되는 모든 작업들이 같은 프로세서에서 실행되어야 하는 파티션드(partitioned) 스케줄이 있으며, 두 번째는 프로세서 간 태스크 이동(migration)이 허용되는(한 태스크의 여러 작업들이 각각 다른 프로세서에서 실행 가능한) 글로벌(global) 스케줄이다. 본 논문의 알고리즘은 유니폼

(uniform, 다양한 종류의) 멀티프로세서 환경에서 글로벌 스케줄링 알고리즘을 기반으로 한다. 그리고 인터럽트를 당한 작업은 다른 프로세서에서 다시 실행(resume) 될 수 있으며, 프로세서 간 이동 시 다른 비용은 없다고 가정한다. 그러나, 하나의 작업은 동시에 여러 개의 프로세서에서 수행될 수 없다.

본 논문에서는 m개의 유니폼 멀티프로세서 환경에서 단순 주기성 태스크 시스템을 스케줄하는 알고리즘을 제안한다. 단순 주기성 태스크 시스템은 각각 주기 $period_i < period_k$를 가지고 있는 태스크 T_i와 T_k가 있고, 주기 $period_k$가 주기 $period_i$의 정수 배수인 태스크들로 이루어진 태스크 시스템을 말한다. 단순 주기성 태스크 시스템은 헬리콥터 비행 컨트롤러의 소프트웨어 구조[8]같은 예에서 볼 수 있다. 이 시스템에는 모든 비행 관련 태스크들이 세 개의 주기 1/180, 1/90, 1/30초 중 하나를 주기로 삼는다. 유니프로세서 환경에서 RM 스케줄링 알고리즘은 단순 주기성 태스크 시스템에 최적(optimal)이라고 이미 증명된 바 있다[13]. 비록 단순 주기성 태스크 시스템이 주기성 태스크 시스템의 특별한 경우지만, 단순 주기성 태스크 시스템은 매우 예측이 쉽고(predictable), 견고한(stable) 특성을 가지기 때문에 내장형 시스템에 아주 유용하다.

각 프로세서는 속도나 계산 용량 s 라는 파라미터를 가지며, 각 작업이 한 프로세서에서 시간 t 동안 실행된다면, $(s \times t)$만큼 실행하게 된다. 동종의 멀티프로세서 환경은 모든 프로세서의 계산 용량(s)이 동일하며, 이것은 유니폼 멀티프로세서 환경의 특별한 경우에 해당된다. 플랫폼 π가 다양한 계산 용량($s_1, s_2, \cdots, s_m$)을 가진 m개의 프로세서를 가졌을 경우, $\pi = [s_1, s_2, \cdots, s_m]$이라고 표기한다. 그리고 프로세서들은 계산 용량에 따라 내림차순으로 정렬된다. 수식 1의 ($s_j \geq s_{j+1}, \forall j, 1 \leq j < m$). $S_{sum}(\pi)$는 프로세서들의 계산 용량 총 합계이다.

$$S_{\mathrm{sum}}(\pi) \stackrel{def}{=} \sum_{i=1}^{j} s_i. \quad (1)$$

본 논문에서는 주어진 단순 주기성 태스크 시스템을 "task-splitting"을 이용하여 다른 단순 주기성 태스크 시스템으로 변환한다. 변환과정을 거친 단순 주기성 태스크 시스템은 파티션드 스케줄링 알고리즘을 이용하여 유니폼 멀티프로세서 환경에서 성공적으로 스케줄 가능하다. 각 프로세서에 할당된 태스크들은 RM 스케줄링 알고리즘으로 스케줄 된다. 그리고 플랫폼이 "reasonably powerful" 한 경우, 제안한 알고리즘이 이론적으로 최대값인 $S_{sum}(\pi)$만큼 최대 이용률 범위를 보장함을 증명하였다.

본 논문 2장에서는 시스템 모델(system model)과 "task-splitting"기법에 대해서 설명하고, 3장에서는 "task-splitting"기법을 이용하여 주어진 단순 주기성 태스크 시스템을 다른 단순 주기성 태스크 시스템으로 변환하는 과정을 설명하고, 변환과정을 거친 단순 주기성 태스크 시스템이 RM 스케줄링 알고리즘에 의해 $S_{sum}(\pi)$만큼 최대 이용률 범위를 보장하면서 성공적으로 스케줄 된다는 것을 증명한다. 마지막으로 4장에서는 결론을 기술한다.

II. 시스템 모델과 "task-splitting" 기법

경성 실시간(hard real-time) 시스템은 작업의 인스턴스(intance)와 작업이 실행되는 컴퓨팅 플랫폼에 의해 기술된다. 실시간 작업 $j = (a, e, d)$은 도착 시간(a)과 실행 요구량(e) 그리고 데드라인(d)을 가지며, 작업은 반드시 시간 구간 $[a, d)$에서 실행 요구량(e)만큼 수행하여야 한다. 실시간 인스턴스 $J = \{j_1, j_2, \cdots,\}$는 작업들의 집합이다. 주기성 태스크 시스템 $\tau = \{T_1, T_2, \cdots, T_n\}$에서 각 주기성 태스크 $T_i = (a_i, e_i, d_i, p_i)$는 옵셋(offset) a_i(태스크에 의해 생성된 첫 번째 작업이 도착한 시간), 실행 요구량 e_i, 상대적 데드라인 d_i, 주기 p_i 4개의 특성을 갖는다. T_i는 계속 작업을 생성하며 각 시점 $(a_i + k \cdot p_i), \forall$ 정수 $k \geq 0$, 에서 실행 요구량 e_i만큼 실행되어야 하며, 시점 $(a_i + k \cdot p_i)$에서 생성된 작업은 데드라인이 $(a_i + k \cdot p_i + d_i)$이다. 주기성 태스크 시스템은 모든

태스크의 옵셋(a) 이 같은 경우 *synchronous*라고 하며 그렇지 않으면 *asynchronous*라고 한다. 태스크 T_i의 이용률 $U(T_i)$은 수식 2와 같이 정의한다.

$$U(T_i) \stackrel{def}{=} e_i/p_i. \tag{2}$$

여기에서 주기성 태스크 시스템 τ의 태스크들은 이용률에 따라 내림차순으로 정렬되고, $U(T_i) \geq U(T_{i+1})$, $\forall i, 1 \leq i < n$, 모든 태스크들의 이용률 총 합계는 $U_{sum}(\tau)$이고 태스크들 중 가장 큰 이용률을 $U_{max}(\tau)$ 라고 표기한다. 그리고 각 작업은 서로 독립적으로(같은 태스크나 다른 태스크들과 shared data에 대한 접근이나 메시지 교환은 일어나지 않음) 동작한다고 가정한다.

본 논문에서는 작업들이 RM^+ 정책에 따라 스케줄된다. RM^+ 정책은 RM 스케줄링 알고리즘과 기본적으로 똑같으며, 다른 점은 데드라인이 같은 작업들은 늦게 도착한 작업에게 더 높은 우선순위를 주고(LIFO), 데드라인과 도착 시간이 모두 같은 작업들은 높은 첨자(index)를 가진 작업에게 더 높은 우선순위를 준다는 점이다. 주어진 단순 주기성 태스크 시스템을 "task-splitting"을 이용하여 다른 단순 주기성 태스크 시스템으로 변환하고, 이것을 RM^+ 정책에 의해 스케줄하게 되면 더 높은 최대 이용률 범위를 얻을 수 있다. 정의1를 통하여 주어진 단순 주기성 태스크 시스템 τ_1을 스케줄 가능성(feasibility)을 유지하면서 다른 단순 주기성 태스크 시스템 τ_2로 변환할 수 있다.

정의 1 : 주어진 멀티프로세서 플랫폼 π에서 τ_2가 RM^+에 의해 정확하게 스케줄 될 수 있다면 단순 주기성 태스크 시스템 τ_1은 τ_2로 변환가능하고(표기 : $\tau_1 \overset{\pi}{\rightsquigarrow} \tau_2$), π상에서 τ_1도 또한 성공적으로 스케줄 될 수 있다.

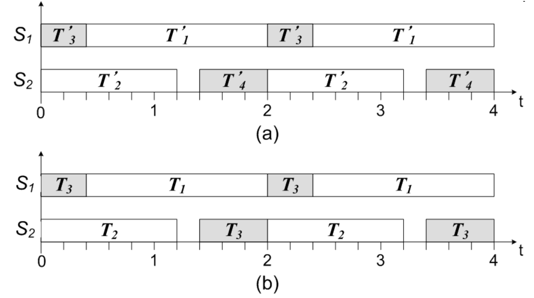

그림 1. 스케줄 예제:(a)τ_2의 RM+ 스케줄; (b) T_1, T_2, T_3을 T_1', T_2', $T_3'+T_4'$로 각각 매핑(mapping)한 τ_1의 스케줄

[그림 1]의 예제로 멀티프로세서 플랫폼 $\pi=[s=1, t=1]$이 있고 다음과 같이 가장 짧은 주기가 2인($P_{\min}=2$) 두 개의 단순 주기성 태스크 시스템들이 있다.

$$\begin{aligned}\tau_1 &= \{T_1, T_2, T_3\}\\ &\equiv \{(0, 3.2, 4, 4), (0, 1.2, 2, 2), (0, 2, 4, 4)\}\\ \tau_2 &= \{T_1', T_2', T_3', T_4'\}\\ &\equiv \{(0, 3.2, 4, 4), (0, 1.2, 2, 2),\\ &\quad (0, 0.4, 0.4, 2), (1.4, 0.6, 0.6, 2)\}\end{aligned}$$

첫 번째 프로세서에 T_1'과 T_3'를 할당하며 나머지 두 개의 태스크 T_2'와 T_4'를 두 번째 프로세서에 할당하고 RM^+ 정책으로 스케줄 할 경우 τ_2는 [그림 1(a)]에서 보는 바와 같이 성공적으로 수행될 수 있다. τ_1과 τ_2를 비교해보면, $T_1 \equiv T_1'$이고 $T_2 \equiv T_2'$이며, T_3은 T_3'과 T_4'로 분할된 것이다. T_3'과 T_4'이 동시에 수행되지 않고 각 시간 구간,

$[4 \cdot k, 4 \cdot (k+1))$,$\forall$ 정수 $k \geq 0$

$(=[a_3+k \cdot p_3, a_3+k \cdot p_3+d_3))$,

에서 $2(=e_3)$ 시간단위만큼 실행된다. [그림 1(b)]에서 보는 바와 같이 각각 T_1, T_2, T_3를 T_1', T_2', $T_3'+T_4'$으로 매핑(mapping) 시켜보면 τ_1은 π상에서 성공적으로 수행될 수 있다. 그래서 단순 주기성 태스

크 시스템 τ_1은 τ_2로 변환가능하다($\tau_1 \overset{\pi}{\rightsquigarrow} \tau_2$). 직관적으로 위의 예에서 볼 수 있듯이 "task-splitting"에 의해 주기성 태스크 시스템을 다른 주기성 태스크 시스템으로 변환가능하며, 변환과정은 정의2와 같다.

정의 2 : 다음에 오는 특성을 만족시키면서 태스크 $T_i = (a_i, e_i, d_i, p_i)$를 가장 짧은 주기 $P_{\min}$을 가진 j개의 태스크들,

$T_{i_k} = (a_{i_k}, e_{i_k}, d_{i_k}, p_{i_k}),\ 1 \le k \le j,$

로 "분할(split)"한다. 그리고 각 태스크 T_{i_k}는 프로세서 s_{i_k}에 할당한다.

$P1: p_{i_k} = p_{\min},$

$P2: d_{i_k} = e_{i_k} / s_{i_k},$

$P3: a_{i_k} + d_{i_k} \le a_{i_{k+1}} \le p_{\min} - d_{i_{k+1}},$

$P4: \sum_{k=1}^{j} U(T_{i_k}) \equiv \sum_{k=1}^{j} U(e_{i_k}/p_{\min}) = U(T_i).$

부정리 1 : 단순 주기성 태스크 시스템 τ_1가 있고 τ_1의 일부 태스크들을 "분할(split)"한 결과로 만들어진 단순 주기성 태스크 시스템 τ_2가 있을 때 멀티프로세서 플랫폼 π에서 $\tau_1 \overset{\pi}{\rightsquigarrow} \tau_2$는 존재한다.

증명 : 주어진 멀티프로세서 플랫폼 π에서 τ_2가 RM+에 의해 정확하게 스케줄 될 수 있다고 가정한다. τ_1의 태스크 $T_i = (a_i, e_i, d_i, p_i)$를 j개의 태스크들 $T_{i_k} = (a_{i_k}, e_{i_k}, d_{i_k}, p_{i_k}),\ 1 \le k \le j,$로 분할하고, 태스크 T_{i_k}들을 프로세서 s_{i_k}들에 할당한다. 이렇게 되면, 각 태스크 T_{i_k}는 자신의 실행 요구량 $e_{i_k} (= s_{i_k} \times d_{i_k})$만큼 수행되어야 하기 때문에 정확히 d_{i_k}동안 실행된다. 그리고 *P3*에 의해서 분할된 태스크 T_{i_k} 들은 전체 스케줄 시간동안 동시에 수행되지 않고 이 태스크들은 각 시간 구간 $[a_i + l \cdot p_i, a_i + l \cdot p_i + d_i),\ \forall$ 정수 $l \ge 0,$ 에서 총 e_i 만큼 수행된다. 태스크 T_i를 j개의 T_{i_k}들로 매핑(mapping)시켜보면 T_i도 π상에서 성공적으로 수행된다. 따라서, $\tau_1 \overset{\pi}{\rightsquigarrow} \tau_2$는 존재한다.

위에서 본 바와 같이, RM 스케줄링 알고리즘에 의해 성공적으로 스케줄 불가능한 단순 주기성 태스크 시스템들은 "task-splitting"에 의해 변환과정을 거친 다음 RM+에 의해 데드라인을 지키며 성공적으로 수행될 수 있고, 더 높은 최고 이용률 범위를 가질 수 있다. 본 논문에서는 단순 주기성 태스크 시스템들은 *synchronous*하며 모든 태스크들은 자신의 데드라인이 자신의 주기와 같다고 가정한다. 그리고 또한 유니폼 멀티프로세서 플랫폼은 "reasonably powerful"하다고 가정한다. "reasonably powerful"은 i번째로 빠른 프로세서가 i번째의 중량을 가진 태스크(i번째로 큰 이용률을 가진 태스크)를 데드라인을 지키며 단독으로 실행할 수 있다는 뜻이다. 부연 설명하자면, 프로세서들은 1조건1을 만족해야한다.

조건 1 : $s_i \ge U(T_i),\ \forall i, 1 \le i \le m.$

단순 주기성 태스크 시스템이 조건 1을 만족한다면 유니폼 멀티프로세서 플랫폼 π에서 이론적으로 최대값인 $S_{sum}(\pi)$만큼 최대 이용률 범위를 보장할 수 있다. 알고리즘에 대한 자세한 설명은 다음 장에서 기술한다.

III. 알고리즘과 증명

주기성 태스크 시스템 τ가 필요충분조건으로 "$\Sigma_{T_i \in \tau_i} U(T_i) \le s_i$을 만족하는 태스크들이 동시에 수행되지 않는 부분집합 $\tau_1, \tau_2, \cdots, \tau_m$으로 나눠진다."을 만족하면서, 유니폼 멀티프로세서 플랫폼 π상에 분배되는 문제는 bin-packing과 파티션드 스케줄링 사이에 상관관계를 연상시킨다. 동종(homogeneous)의 멀티프로세서 상에서 태스크 분배(partitioning)는 모든 상자(bin)들의 크기가 같은 경우의 bin-packing 문제와

일치한다. Johnson[9]은 이 문제가 엄밀하게 말해 "NP-complete"하다고 증명했다. 그러나, 이 장에서는 이론적으로 최대값인 $S_{sum}(\pi)$만큼 최대 이용률 범위를 가진 단순 주기성 태스크 시스템이 변환과정을 거쳐 유니폼 멀티프로세서 플랫폼 π에 분배될 수 있다는 것을 보이고 증명한다.

1. Pre-assignment 알고리즘

이 알고리즘은 "task-splitting"과정 전에 먼저 태스크들을 분할하지 않고 프로세서에 할당하는 알고리즘으로 태스크들을 이용률 기준으로 내림차순 정렬을 하고 First-Fit-Decreasing(FFD) 정책으로 프로세서에 할당한다. 태스크의 이용률이 프로세서의 남은 계산 용량($s_i - U_i$)보다 작거나 같다면(U_i는 해당 프로세서에 이미 할당된 태스크들의 이용률 합계) 태스크는 프로세서 s_i에 할당되고 이 경우에 태스크는 해당 프로세서에 *fit*하다고 정의한다. 자세한 알고리즘은 [그림 2]와 같다. 각 태스크는 *fit*인 가장 빠른 프로세서에 할당된다. 변수 gap(j)는 프로세서 s_j의 남은 용량이고 rem은 pre-assignment과정 후에도 프로세서에 할당되지 못한 태스크들의 집합이다.

FFD pre-assignment 알고리즘은 먼저 태스크 T_i를 태스크 자신의 이용률 $U(T_i)$보다 큰 gap을 가지며 동시에 가장 작은 첨자 j를(가장 빠른, 가장 계산 용량이 큰) 가진 프로세서 s_j에 할당한다. 이런 방식으로 계속 할당하다가 모든 프로세서의 gap들이 $U(T_i)$보다 작다면, T_i는 rem 집합에 포함된다. 이렇게 할당되지 못한 태스크들은 뒤에 task-splitting 알고리즘에 의해 분할되어 여러 개의 프로세서에 할당된다.

FFD pre-assignment 알고리즘의 계산 복잡도는 $O(n \log n)$(태스크들을 이용률 기준으로 내림차순 정렬) $+O(m \log m)$(프로세서들을 계산 용량 기준으로 내림차순 정렬) $+O(n \times m)$(프로세서에 태스크 할당)이며, 전체 계산 복잡도는 프로세서의 개수가 태스크의 개수를 초과하지 않으면 $O(n \cdot (\log n + m))$이다.

FFD task pre-assignment(τ, π)

Let $\tau = \{T_1, T_2, \cdots, T_n\}$ denote the tasks, with $U(T_i) \geq U(T_{i+1})$ for all i

Let $\pi = [s_1, s_2, \cdots, s_m]$ denote the processors, with $s_j \geq s_{j+1}$ for all j

1. for $j \leftarrow 1$ to m do **gap**$(j) := s_j$
2. **rem**:= 0
3. for $i \leftarrow 1$ to n do
4. Let j_o denote the smallest index such that **gap**$(j_o) \geq U(T_i)$
5. If no such j_o exists, then **rem**:=**rem** $\cup \{T_i\}$; break
6. Assign T_i to processor j_o
7. **gap**(j_o) :=**gap**$(j_o) - U(T_i)$
8. od

그림 2. FFD Pre-assignment 알고리즘

2. Task-splitting 알고리즘

Task-splitting 알고리즘은 Pre-assignment 과정 후에 남은 태스크들을 분할하고 분할된 태스크들을 프로세서에 할당하는 알고리즘이다. 선행 과정인 FFD pre-assignment 과정을 통해서 모든 태스크들을 프로세서에 할당할 수 있다면(결과적으로 rem이 공집합일 경우) "task-splitting" 과정은 필요 없다. 이 경우에는 단순 주기성 태스크 시스템은 변환과정을 거치지 않고 RM 알고리즘에 의해 성공적으로 스케줄된다. 그렇지 않고 rem이 공집합이 아닌 경우, rem에 속한 각 태스크들은 분할되어 여러 프로세서에 할당된다. 그러나 rem 안에 있는 태스크들의 숫자(‖rem‖)는 부정리 2와 같이 한계가 정해진다.

부정리 2 : 만약 $U_{sum}(\tau) \leq S_{sum}(\pi)$ 이면, ‖rem‖ $< m$이다.

증명 : ‖rem‖ $> m$이라고 가정하자. 선행 과정인 FFD pre-assignment 알고리즘에 의하여,

gap(j) $< U(T_i)$, $1 \leq j \leq m$, $T_i \in$rem,

이고, 그 결과 $\Sigma_{T_i \in rem} U(T_i) > \Sigma_{1 \leq j \leq m}$gap($j$)이다. FFD pre-assignment 과정에 의해 이미 할당된 모든

태스크들의 이용률 합계를 U_a라고 하면 (3)식이 성립된다.

$$\begin{aligned} U_{\rm sum}(\tau) &= U_a + \sum_{T_i \in \rm rem} U(T_i) \\ &> U_a + \sum_{1 \le j \le m} {\rm gap}(j) \\ &= S_{\rm sum}(\pi). \end{aligned} \quad (3)$$

수식 (3)에 의하면 $U_{sum}(\tau) \le S_{sum}(\pi)$과 모순이므로 부정리 2는 성립한다.

부정리 3 : 만약 $U_{sum}(\tau) \le S_{sum}(\pi)$ 이면,

$U(T_i) \le \frac{S_{sum}(\pi)}{m+1}$, $\forall T_i \in \rm rem$ 이다.

증명 : $u_{\max}$를 rem에 속하는 태스크들 중 가장 큰 이용률이라고 하면,

$$U_{sum}(\tau) \ge U_a + u_{\max}. \quad (4)$$

조건 1에 따라 m개의 큰 중량을 가진 태스크들은 할당을 보장받는다. 그래서

$$U_a \ge m \times u_{\max}. \quad (5)$$

수식 (4), (5) 의해서

$$U_{\max} \le \frac{U_{sum}(\tau)}{m+1} \le \frac{S_{sum}(\pi)}{m+1} \quad (6)$$

이므로, 부정리 3은 성립한다.

u'를 $(m+1)$번째 큰 중량을 가진 태스크의 이용률이라고 하면, 큰 중량을 가진 m개의 태스크들은 할당을 보장받기 때문에 $u_{\max} \le u'$이다. 그래서 추론(corollary)1이 성립한다.

추론 1 : $U(T_i) \le s_j$, $\forall j, 1 \le j \le m$, $T_i \in \rm rem$.

각 프로세서들의 남은 계산 용량이 rem에 포함된 각 태스크의 이용률보다 적기 때문에, 부정리 3에서 추론 2를 도출할 수 있다.

추론 2 : 만약 $U_{sum}(\tau) \le S_{sum}(\pi)$ 이면,

${\rm gap}(j) < \frac{S_{sum}(\pi)}{m+1}$, $\forall j$, $1 \le j \le m$이다.

부정리 4 : 만약 $\| \rm rem \| > 0$ 이라면,

${\rm gap}(j) < \frac{S_j}{2}$, $\forall j$, $1 \le j \le m$이다.

증명 : 조건 1에 의하면, $U(T_i) \le s_j$, $\forall j$, $1 \le j \le m$, $T_i \in \rm rem$이다. 그래서 각 프로세서 s_j에는 이용률이 $U(T_k) \ge U(T_i)$인 태스크 T_k가 적어도 한 개는 할당된다. U_j를 프로세서 s_j에 할당된 태스크들의 이용률 합계라고 한다면, 다음 수식(7)과 같은 결과가 성립된다.

$$\begin{aligned} s_j &= U_j + {\rm gap}(j) \\ &\ge U(T_i) + {\rm gap}(j) \\ &> 2 \times {\rm gap}(j). \end{aligned} \quad (7)$$

[그림 3]은 Task-splitting 알고리즘이다. 먼저 rem에 속한 태스크들을 이용률을 기준으로 내림차순 정렬을 하고 프로세서들은 남은 계산 용량을 기준으로 내림차순 정렬을 한다. 각 태스크는 알고리즘에 의해 더 작은 이용률을 가진 태스크들로 분할되는데 이 태스크들은 해당 프로세서의 남은 계산 용량과 같은 이용률을 가지고(마지막 남은 분할 태스크는 제외) 해당 프로세서에 할당된다. 변수 left(i)는 분할이후에 태스크 T_i'의 남은 이용률이다. 분할과정 이후에 분할된 태스크들이 동시에 실행되는 상황이 없도록 하기 위해 그 다음 분할 태스크의 옵셋(a)을 증가시킨다.(알고리즘의 9라인).

그리고 문맥전환(context switch) 횟수를 줄이기 위

해 마지막 남은 분할 태스크의 옵셋은 최소 주기($P_{\min}$) 구간 내에서 가장 나중에 실행되도록 정해진다(알고리즘의 13라인).

```
Task splitting(rem, π):

Let rem = {T'_1, T'_2, ··· , T'_k} denote the tasks,
with U(T'_i) ≥ U(T'_{i+1}) for all i

Let π = [s'_1, s'_2, ··· , s'_m] denote the processors,
with gap(j) ≥ gap(j + 1) for all j

1.  for j ← 1 to k do
2.  p := 1

3.  for i ← 1 to k do
4.     a := 0, q := 1
5.     while (p ≤ m) do
6.       If left(i) ≥gap(p), then
7.         e :=gap(p) · p_min; d := e/s'_p
8.         Make a new task T'_{i_q} = (a, e, d, p_min) and
             assign it to processor p;  q := q + 1
9.         a := a + d;  p := p + 1
10.        left(i) :=left(i)−gap(p); gap(p) := 0
11.        If left(i) = 0, then break
12.      else
13.        e :=left(i) · p_min; d := e/s'_p; a := p_min − d;
           left(i) := 0
14.        Make a new task T'_{i_q} = (a, e, d, p_min)
             and assign it to processor p
15.        gap(p) := gap(p)−left(i); break
16.      od
17.  od
```

그림 3. Task-splitting 알고리즘

[그림 3]의 Task-splitting 알고리즘의 계산 복잡도는 $O(k\log k)$(rem에 포함된 태스크들을 이용률 기준으로 내림차순 정렬) $+O(m\log m)$(프로세서들을 남은 계산 용량 기준으로 내림차순 정렬) $+O(k\times m)$(태스크 분할과정)이며, 전체 계산 복잡도는 부정리 2에 의해 $k<m$이기 때문에 $O(m^2)$이다.

rem에 포함된 태스크들은 자신의 이용률이 남아있는 각 프로세서의 계산 용량보다 크기 때문에 2개 이상의 태스크로 분할된다. 태스크를 분할하여 새로운 태스크들을 만드는 과정을 보면(Task-splitting 알고리즘의 내부 루프), 정의 2의 모든 특성을 만족한다는 것을 알 수 있다.

3. 알고리즘 증명

태스크 T_i'가 j개의 태스크들 T_{i_k}', $1\le k\le j$, 로 분할된다면, 알고리즘에 의해 분할된 태스크들은 j개의 프로세서 s_p', $l\le p\le l+j-1$,에게 하나씩 할당된다. 다음 태스크 T'_{i+1}가 j'개의 태스크들 $T'_{(i+1)_k}$, $1\le k\le j'$로 분할된다면, 역시 마찬가지로 분할된 태스크들은 j'개의 프로세서 s_p', $l'\le p\le l'+j'-1$, 에게 하나씩 할당된다. 그 결과 $l'=j$ 또는 $l'=j+1$이며, 추론 3을 정의할 수 있다.

추론 3 : 각 프로세서에는 분할된 태스크들이 많아야 2 개 할당된다.

부정리 4에 의하면, 분할된 태스크들은 자신이 할당된 프로세서의 계산 용량의 반 이하만 사용하게 된다. 태스크 T_i'가 j개의 태스크들 T_{i_k}', $1\le k\le j$,로 분할되고 j개의 프로세서 s_p', $l\le p\le l+j-1$, 에게 하나씩 할당될 때, 분할된 모든 태스크들이 각 프로세서에서 성공적으로 스케줄 된다면 부정리 1의 증명대로 각 T_i'는 정확히 $[a_{i_k}, a_{i_k}+d_{i_k})$동안 실행된다. 그래서 최소 주기($P_{\min}$) 구간 내에서 같은 태스크에서 분할된 태스크들이 동시에 수행되는 상황없이 스케줄된다. 이를 다음 부정리 5에서 정식으로 설명하고 증명한다.

부정리 5 : 다음의 조건을 만족하면서 태스크 T_i'가 j개의 태스크들 T_{i_k}', $1\le k\le j$로 분할된다.

E1 : $a_{i_1}=0$,

E2 : $a_{i_k}+d_{i_k}=a_{i_{k+1}}$, $\forall k, 1\le k\le j-1$

E3 : $a_{i_{j-1}}+d_{i_{j-1}}\le a_{i_j}=p_{\min}-d_{i_j}$.

증명 : E1과 E2는 알고리즘을 보면 명백하며, E3을 증명한다. E3이 성립하지 않는다고 가정하면 $\sum_{k=1}^{j} d_{ik} > p_{\min}$ 이다. 알고리즘에 의해 분할된 태스크들 T_{i_k}', $1\le k\le j$,가 j개의 프로세서 s_p', $l\le p\le l+j-1$,에 각각 할당된다면 다음과 같다.

$$p_{min} < \sum_{k=1}^{j} d_{i_k} = \frac{e'_{i_1}}{s'_l} + \frac{e'_{i_2}}{s'_{l+1}} + \cdots + \frac{e'_{i_j}}{s'_{l+j-1}}$$

[그림 3] Task-splitting 알고리즘의 7라인에 의해서,

$$p_{min} < \sum_{k=1}^{j} d_{i_k} = p_{min} \cdot \left\{ \frac{\mathrm{gap}(s'_l)}{s'_l} + \frac{\mathrm{gap}(s'_{l+1})}{s'_{l+1}} + \cdots + \frac{\mathrm{gap}(s'_{l+j-2})}{s'_{l+j-2}} + \frac{\mathrm{left}(T'_{i_j})}{s'_{l+j-1}} \right\}$$

$$\Rightarrow p_{min} < \sum_{k=1}^{j} d_{i_k} = p_{min} \cdot \left\{ \frac{U(T'_{l_1})}{s'_l} + \frac{U(T'_{l_2})}{s'_{l+1}} + \cdots + \frac{U(T'_{i_{j-1}})}{s'_{l+j-2}} + \frac{U(T'_{i_j})}{s'_{l+j-1}} \right\}$$

$s'_{\min} = \min\begin{cases} l+k-1 \\ p=l \end{cases} s'_p$라고 하면, 다음과 같은 결과를 얻을 수 있다.

$$1 < \sum_{k=1}^{j} d_{i_k} \le \frac{\sum_{k=1}^{j} U'_{i_k}}{s'_{\min}} = \frac{U(T'_i)}{s'_{\min}} \qquad (4)$$

추론 1에 의해, $U(T'_i) \le s'_j,\ \forall j,\ 1 \le j \le m$, $T'_i \in \mathrm{rem}$이며 이것은 식(4)에 모순된다.

Γ_i를 프로세서 s_i에 할당된 태스크들(분할된 태스크 포함)의 집합이라고 하고, 알고리즘에 따라 현재 프로세서의 남은 계산 용량이 0이 되면 프로세서 첨자(변수 p)가 증가한다. 만약 $U_{sum}(\tau) \le S_{sum}(\pi)$ 이면, $\Sigma_{T_j \in \Gamma_i} U(T_j) \le s_i,\ \ \forall i$,이고 $U_{sum}(\tau) = S_{sum}(\pi)$ 이면, $\Sigma_{T_j \in \Gamma_i} U(T_j) = s_i,\ \forall i$,이다. 그래서 다음 정리 2에서 증명한바와 같이 $U_{sum}(\tau) \le S_{sum}(\pi)$ 이면, 태스크 집합 Γ_i는 각 프로세서 s_i상에서 RM^+에 의해 성공적으로 스케줄 가능하다.

정리 1 : 상대적 데드라인이 자신의 주기보다 크거나 같고 다른 태스크에 독립적이며, 선점형 특성을 가진 단순 주기성 태스크 시스템은 총 이용률이 1보다 작거나 같으면 RM 스케줄링 알고리즘에 의해 유니프로세서에서 스케줄 가능하다.

증명 : 이 정리는 참고문헌[13]에서 이미 증명되었다.

정리 2 : $U_{sum}(\tau) \le S_{sum}(\pi)$ 이면, 태스크 집합 Γ_i는 각 프로세서 s_i에서 RM^+에 의해 성공적으로 스케줄 가능하다.

증명 : 추론 3에서 볼 수 있듯이 태스크 집합 Γ_i에는 분할 태스크들이 많아야 2개 할당된다. 분할된 태스크 $T_{i_k} = (a_{i_k}, e_{i_k}, d_{i_k}, p_{\min})$을 $T'_{i_k} = (0, e_{i_k}, p_{\min}, p_{\min})$으로 수정한다면 수정된 결과 태스크 집합 Γ'_i는 단순 주기성을 가지고 *synchronous*하며, 모든 태스크가 데드라인이 주기와 같게 된다. 그리고 $U_{sum}(\Gamma'_i) \le s_i$이기 때문에 RM 스케줄링 알고리즘으로 Γ'_i는 적절하게 스케줄 될 수 있다. RM 스케즐링 알고리즘에 의한 Γ'_i의 스케줄을 $RM.\Gamma'_i$이라고 하자. $RM.\Gamma'_i$ 스케줄에서 각 분할된 태스크 $T'_{i_k} = (0, e_{i_k}, p_{\min}, p_{\min})$는 각 시간 구간, $(j \cdot p_{\min}, (j+1) \cdot p_{\min}]$, $\forall j$, 정수 $j \ge 0$,에서 e_{i_k} 만큼 실행된다. 그리고 태스크들이 가진 가장 짧은 주기가 $p_{\min}$이기 때문에 실행 중에 다른 태스크에 의해 선점당하지 않는다. FFD pre-assignment 과정에 의해 프로세서에 할당된 태스크(분할되지 않은 태스크) T_o은 각 시간 구간 $(j \cdot p_{\min}, (j+1) \cdot p_{\min}]$에서 실행되고 데드라인은 정수 $l (\ge (j+1) \cdot p_{\min})$이다. 시간 구간 $[j \cdot p_{\min}, (j+1) \cdot p_{\min})$동안의 각 task-splitting 알고리즘에 의해 분할된 태스크 T_{i_k}의 실행을 시간 구간 $[j \cdot p_{\min} + a_{i_k},\ j \cdot p_{\min} + a_{i_k} + d_{i_k})$로 이동해보면 $RM.\Gamma'_i$로부터 새로운 스케줄 S'를 얻을 수 있다. 새로운 스케줄 S'상에서 T_o의 총 실행 시간은 각 시간 구간 $[j \cdot p_{\min}, (j+1) \cdot p_{\min})$동안 동일하며, 결론적으로 새로운 스케줄 S'상에서 각 분할된 태스크들과 T_o은 여전히 데드라인을 어기지 않는다. 그리고 새로운 스케줄 S'의 우선순위 정책은 RM^+와 동일하다. 프로세서에서 이미 할당된 태스크들의 우선순위 정책이 RM^+이기 때문에, S'는 태스크 집합 Γ_i가 RM^+으로 우

선순위가 정해지는 스케줄이다. 그래서 $U_{sum}(\tau) \le S_{sum}(\pi)$인 경우, 태스크 집합 Γ_i는 각 프로세서 s_i상에서 RM+에 의해 성공적으로 스케줄 가능하다.

지금까지 설명한 바과 같이 본 논문의 알고리즘(FFD pre-assignment와 task-splitting 알고리즘)에 의해 각 프로세서에 할당된 태스크 집합은 RM+에 의해 성공적으로 스케줄된다. 또한, 부정리 5에서 같은 태스크에서 분할된 태스크들이 동시에 수행되지 않다는 것을 보였다. 그래서 주어진 유니폼 멀티프로세서 플랫폼 π에서 단순 주기성 태스크 시스템 τ는 필요충분조건 "총 이용률 $U_{sum}(\tau)$이 프로세서들이 가진 계산 용량의 총 합계 $S_{sum}(\pi)$보다 크지 않다."을 만족할 때 성공적으로 스케줄 가능하다.

정리 3 : 주어진 유니폼 멀티프로세서 플랫폼 π에서 단순 주기성 태스크 시스템 τ은 다음의 필요충분조건을 만족할 경우 성공적으로 스케줄 가능하다.

$U_{sum}(\tau) \le S_{sum}(\pi)$.

증명 : τ'를 τ의 태스크들이 "task-splitting"을 통해 변환된 결과로 나온 단순 주기성 태스크 시스템이라고 하면, 정리 2와 부정리 5에 의하여, τ'는 유니폼 멀티프로세서 플랫폼 π상에서 $U_{sum}(\tau) \le S_{sum}(\pi)$인 경우 성공적으로 스케줄 가능하다. 그리고 부정리 1의 $\tau_1 \overset{\pi}{\rightsquigarrow} \tau_2$는 τ도 또한 π상에서 성공적으로 스케줄 가능하다는 것을 뜻한다. 게다가 $S_{sum}(\pi)$는 π의 이론적인 최고 이용률 범위이다.

정리 3에 의해, 최고 이용률 범위 관점에서 본 논문에서 제시한 알고리즘이 최적(optimality)이라는 것을 확인할 수 있다.

4. 멀티프로세서 알고리즘 비교분석

본 논문에서 제시한 알고리즘의 장점과 단점을 기존 연구되어진 Pfair 알고리즘, LLREF 알고리즘, RMDP 알고리즘과 비교 분석한다. 멀티프로세서를 위한 스케줄링 알고리즘은 시스템 모델과 가정에서 이론적인 접근을 많이 사용하고, 시스템 모델과 가정이 서로 다르기 때문에 다른 알고리즘과 직접 비교하기 어렵다. 일례로, Pfair 알고리즘, LLREF 알고리즘, RMDP 알고리즘은 동종의 멀티프로세서를 가정한 알고리즘으로 유니폼 멀티프로세서 환경에서 태스크들의 마감시간을 보장할 수 없다. 하지만 본 논문의 알고리즘은 최적이다. 또한, 동종의 멀티프로세서 환경이라면, 비교한 모든 알고리즘이 정상동작할 수 있으며 본 논문의 알고리즘은 프로세서들을 계산 용량에 따라 정렬할 필요가 없기 때문에 오버헤드가 감소한다.

[표 1]에서는 제시한 4개의 알고리즘을 비교 분석한다. 본 논문에서 제시한 알고리즘의 단순 주기성 태스크 시스템은 주기성 태스크 시스템의 부분집합이기 때문에 응용 범위가 축소되는 단점이 있지만, 예측성과 안정성이 중요시되어 단순 주기성 태스크 시스템을 주로 사용하는 산업이나 국방 분야의 내장형 시스템에 적용이 용이하다.

표 1. 알고리즘 비교 분석

알고리즘	Pfair	LLREF	RMDP	TA-RM+
우선순위정책	동적	동적	고정	고정
태스크종류	주기성	주기성	주기성	단순 주기성
이용률	100%	100%	50%	100%
멀티프로세서	동종	동종	동종	유니폼
장점	·최적 ·넓은 응용범위	·최적 ·넓은 응용범위	·낮은 오버헤드 ·뛰어난 예측성, 안정성	·최적 ·다양한 프로세서 지원 ·낮은 오버헤드 ·뛰어난 예측성, 안정성
단점	·높은 오버헤드 ·열악한 예측성 ·제한된 프로세서	·높은 오버헤드 ·열악한 예측성 ·제한된 프로세서	·최적이 아님 ·제한된 프로세서	·응용 범위 한정

5. 결론

본 논문에서는 유니폼 멀티프로세서 플랫폼에서 단순 주기성 태스크 시스템을 성공적으로 스케쥴 하기 위한 새로운 글로벌 스케줄링을 제안했다. 이 문제는 bin-packing 문제와 같은 문제로써 해결하는 게 불가능하다고 알려져 있다. 그러나 본 논문에서는 FFD pre-assignment와 task-splitting 알고리즘을 이용하여 이 문제를 해결하였다. 제시한 알고리즘이 "reasonably powerful"한 유니폼 멀티프로세서 플랫폼에서 $U_{sum}(\tau) \le S_{sum}(\pi)$라면 각 프로세서에 RM^{+}알고리즘을 적용하여 어떤 단순 주기성 태스크 시스템이라도 성공적으로 스케줄 가능함을 증명하였다.

본 논문의 알고리즘은 여전히 선점(preemption)이 자주 일어나지만, 단순 주기성 태스크 시스템이 사용되는 특수한 분야에 쓰이는 플랫폼의 가장 짧은 주기($p_{\min}$)가 큰 값인 경우, 선점에 관한 오버헤드(overhead)가 상대적으로 비례하여 줄어든다. 그리고 각 프로세서가 작은 슬랙(사용되지 않는 cycle)을 갖는 경우, 분할이 적어지고 cpu간 이동(migration)이 줄어 들 수 있기 때문에 오버헤드가 역시 줄어든다.

향후에는 "task-splitting"기법의 장점을 살려, 정적 우선순위 알고리즘뿐만 아니라 동적 우선순위 알고리즘에도 적용하는 연구가 필요하다.

참 고 문 헌

[1] B. Andersson and E. Tovar, “Multiprocessor Scheduling with Few Preemptions,” Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp.322-334, 2006(8).

[2] B. Andersson and J. Jonsson, “The Utilization Bounds of Partitioned and Pfair Static-Priority Scheduling on Multiprocessors are 50\%”, Proc. of the Euromicro Conference on Real-Time Systems, pp.33-40, 2003(7).

[3] S. Baruah, et al., “Proportionate Progress: A Notion of Fairness in Resource Allocation,” Algorithmica, Vol.15, pp.600-625, 1996.

[4] S. Baruah and Goossens. J, “Rate-Monotonic Scheduling on Uniform Multiprocessors,” IEEE Transactions on Computers, Vol.52, pp.966-970, 2003(7).

[5] H. Cho, B. Ravindran, and E. D. Jensen, “An Optimal Real-Time Scheduling Algorithm for Multiprocessors,” Proc. of the IEEE Real-Time Systems Symposium, pp.101-110, 2006(12).

[6] M. Cirinei and T. P. Baker, “EDZL Scheduling Analysis,” Proc. of the Euromicro Conference on Real-Time Systems, pp.9-18, 2007(7).

[7] S. K. Dhall and C. L. Liu, “On a Real-Time Scheduling Problem,” Operations Research, Vol.26, pp.127-140, 1978.

[8] J. R. Ellis, “A new approach to ensuring deterministic processing in a integrated avionics software systems,” Proc. IEEE NAECON, pp.756-764, 1985.

[9] D. Johnson, “Fast Algorithms for Bin Packing,” Joual of Computer and Systems Science, Vol.8, No.3, pp.272-314, 1974.

[10] S. Kato and N. Yamasaki, “Portioned Static-Priority Scheduling on Multiprocessors,” Proc. of the IEEE International Symposium on Parallel and Distributed Processing, pp.1-12, 2008(4).

[11] S. Kato and N. Yamasaki, “Real-Time Scheduling with Task Splitting on Multiprocessors,” Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp.441-450, 2007(8).

[12] C. L. Liu and J. W. Layland, “Scheduling algorithms for multi-programming for a hard real-time environment” JACM, Vol.20, No.1, pp.46-61, 1973(1).

[13] Jane W. S. Liu, *Real-Time Systems*, Prentice-Hall, 2000.

[14] A. K. Mok, "Fundamental Design Problems of Distributed Systems for Hard-Real-Time Environment," PhD thesis, Laboratory for Computer Science, Massachusetts Inst. of Technology, 1983, Technical Report No. MIT/LCS/TR-297.

저 자 소 개

정 명 조(Myoung-Jo Jung) 정회원

- 1999년 2월 : 원광대학교 컴퓨터공학과 (공학사)
- 2003년 2월 : 충남대학교 컴퓨터공학과 (공학석사)
- 2003년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 박사과정

<관심분야> : 자바가상머신, 실시간 스케줄링, 실시간 운영체제

조 문 행(Moon-Haeng Cho) 정회원

- 2004년 2월 : 충남대학교 컴퓨터공학과 (공학사)
- 2006년 2월 : 충남대학교 컴퓨터공학과 (공학석사)
- 2006년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 박사과정 재학

<관심분야> : 실시간 컴퓨팅, 실시간 운영체제, 초소형 초절전 실시간 운영체제

김 주 만(Joo-Man Kim) 정회원

- 1984년 : 숭실대학교 전자계산학(공학사)
- 1998년 : 충남대학교 컴퓨터공학과 (공학석사)
- 2003년 : 충남대학교 컴퓨터공학과 (공학박사)
- 1985년 1월 ~ 2000년 2월 : ETRI 책임연구원(운영체제연구팀장)
- 1995년 7월 ~ 1996년 6월 : Novell Inc. Research Center 방문 연구원
- 2000년 3월 ~ 2006년 2월 : 밀양대학교 정보통신공학부 교수
- 2006년 3월 ~ 현재 : 부산대학교 바이오정보전자 교수

<관심분야> : 실시간 임베디드 시스템, 저전력 OS, 바이오 센서 제어, 유비쿼터스 컴퓨팅

이 철 훈(Cheol-Hoon Lee) 정회원

- 1983년 : 서울대학교 전자공학과 (공학사)
- 1983년 ~ 1986년 : 삼성전자 컴퓨터개발실 연구원
- 1988년 : 한국과학기술원 전기 및 전자공학과 (공학석사)
- 1992년 : 한국과학기술원 전기 및 전자공학과 (공학박사)
- 1992년 3월 ~ 1994년 2월 : 삼성전자 컴퓨터사업부 책임연구원
- 1994년 2월 ~ 1995년 2월 : Univ. of Michigan 객원연구원
- 2004년 2월 ~ 2005년 2월 : Univ. of Michigan 교환교수
- 1995년 ~ 현재 : 충남대학교 컴퓨터공학과 교수

<관심분야> : 병렬처리, 운영체제, 실시간 커널 및 결함허용 컴퓨팅