다목적 유전 알고리즘을 이용한 실시간 태스크의 정적 스케줄링 기법

(A Multiobjective Genetic Algorithm for Static Scheduling of Real-time Tasks)

오 재 원 † 김 희 천 †† 우 치 수 †††
(Jaewon Oh) (Heechern Kim) (Chisu Wu)

요 약 본 본문에서는 다중 처리기 시스템에서 실시간 태스크를 정적으로 스케줄링하기 위한 새로운 기법을 제안한다. 태스크는 실행 시간과 마감시간을 지니고, 태스크 사이에는 선행 관계가 존재하며 이러한 사항을 태스크 그래프로 표현한다. 본 논문에서는 스케줄링을 위해 사용하는 처리기의 개수를 줄이면서 태스크들의 마감시간 지연의 총합을 최소화하는 스케줄을 생성하는 것에 목적을 둔다. 이 문제는 같은 단위로 측정할 수 없고 또한 서로 상충하는 두 가지 목적을 지닌 것이다. 그렇지만 기존 방법들은 마감시간 지연의 총합만을 최소화하려하거나 두 가지 목적을 하나의 기준으로 결합시킨 후 최적화하고자 한다. 본 논문에서는 두 개의 목적을 독립적으로 고려하며 최적화를 위하여 다목적 유전 알고리즘을 사용한다. 태스크 스케줄링 문제에 적합한 문제 표현 전략, 우세 개념에 기초한 선택 연산, 그리고 교차 연산을 제시한다. 그리고 지역 개선 작업을 위해 세 개의 휴리스틱을 제안하였으며 이 것을 통해 유전 알고리즘의 수렴성을 높이고자 하였다. 성능 평가를 위해 기존에 알려진 유전 알고리즘과 4 개의 리스트 스케줄링 알고리즘과 비교한다. 평가 결과를 보면 제안한 기법이 180 개의 임의로 생성한 태스크 그래프 중에서 178 개에 대해 기존 5 개의 알고리즘과 유사하거나 더 나은 스케줄을 생성하였다.

키워드 : 스케줄링, 실시간 시스템, 마감시간, 마감시간 지연의 총합, 다목적 유전 알고리즘

Abstract We consider the problem of scheduling tasks of a precedence constrained task graph, where each task has its execution time and deadline, onto a set of identical processors in a way that simultaneously minimizes the number of processors required and the total tardiness of tasks. Most existing approaches tend to focus on the minimization of the total tardiness of tasks. In another methods, solutions to this problem are usually computed by combining the two objectives into a single criterion to be optimized. In this paper, the minimization is carried out using a multiobjective genetic algorithm (GA) that independently considers both criteria by using a vector-valued cost function. We present various GA components that are well suited to the problem of task scheduling, such as a non-trivial encoding strategy, a domination-based selection operator, and a heuristic crossover operator. We also provide three local improvement heuristics that facilitate the fast convergence of GA's. The experimental results showed that when compared to five methods used previously, such as list-scheduling algorithms and a specific genetic algorithm, the performance of our algorithm was comparable or better for 178 out of 180 randomly generated task graphs.

Key words : Scheduling, Real-time System, Deadline, Total Tardiness, Multiobjective Genetic Algorithm

† 정 회 원 : 서울대학교 컴퓨터공학부
jwoh@selab.snu.ac.kr
†† 정 회 원 : 한국방송통신대학교 컴퓨터과학과 교수
hckim@knou.ac.kr
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
wuchisu@selab.snu.ac.kr
논문접수 : 2003년 4월 18일
심사완료 : 2003년 12월 22일

1. 서 론

태스크 스케줄링이란 주어진 응용 프로그램을 구성하는 태스크들에 대해 태스크를 어떤 처리기에 할당하며 태스크의 수행 시작 시간을 언제로 할 것인가를 결정하는 문제를 말한다. 실시간 태스크 스케줄링에서 고려해야할 두 가지 요소가 있다. 즉, 시간 측면에서 태스크의 마감시간 지연(tardiness)과 자원 측면에서 사용하는 처

리기의 수이다. 시스템 설계자는 재설계를 위해 혹은 연성 실시간 시스템(soft real-time system)의 수용 여부를 가늠하기 위해 태스크가 얼마나 마감시간을 지연하는지에 관한 정보를 알 필요가 있다. 또한 실용적이기 위해서는 스케줄링 알고리즘이 생성한 스케줄이 사용하는 처리기 수의 측면에서 경제적일 필요가 있다[1].

태스크 스케줄링 기법의 유형에는 스케줄이 프로그램의 수행 시간 이전에 미리 정해지는 정적 기법(static method)과, 프로그램이 실제 수행되는 시점에 정해지는 동적 기법(dynamic method), 그리고 프로그램 수행 시에도 확정적이지 않은 이주 기법(migration method) 등이 있다. 본 논문에서는 정적 스케줄링 기법을 제안한다.

태스크 t_i는 실행 시간 e_i와 마감시간 d_i를 지니며, 두 개의 태스크 t_i와 t_j 사이에는 선행 관계(precedence relation) $<t_i, t_j>$가 존재할 수 있다. $<t_i, t_j>$는 반드시 t_j의 실행 이전에 t_i의 실행을 마쳐야 한다는 것을 의미한다. 이러한 사항을 태스크 그래프(graph)로 표현할 수 있으며, 이 경우 그래프의 노드(node)는 태스크를 의미하며 선행 관계는 간선(edge)으로 표현할 수 있다. 앞으로 다중 처리기 시스템이라고 언급하면, 이 시스템은 동일한 처리기(identical processor) 여러 개로 구성된 시스템이라고 생각하자. 본 논문에서 고려하는 문제는 다중 처리기 시스템에서 태스크들을 선행 관계를 만족시키며 태스크의 선점(preemption) 없이 스케줄링하는 것으로 다음 두 가지 목적을 동시에 최소화하는 스케줄을 찾는 것이다. 첫째는 스케줄이 사용하는 처리기 수이며 둘째는 태스크의 마감시간 지연의 총합이다. 여기서 태스크 t_i의 실행 종료 시간이 γ_i일 때, 태스크 t_i의 마감시간 지연(tardiness)은 $\max\{0,(\gamma_i-d_i)\}$로 정의한다. 즉, 태스크가 마감시간을 넘기며 종료할 때는 어긴 시간이 마감시간 지연이며, 그렇지 않으면 마감시간 지연은 0이다. 따라서 태스크가 n 개일 때 마감시간 지연의 총합(total tardiness)은 $\tau=\sum_{i=1}^{n}\max\{0,(\gamma_i-d_i)\}$으로 정의할 수 있다.

본 논문에서 고려하는 문제는 Yalaoui와 Chu[2]가 해결하고자 했던 문제와 비슷하다. 그들은 다중 처리기 시스템에서 선점 없이 마감시간 지연의 총합을 최소화하고자 하였다. 여기서 Yalaoui와 Chu가 언급한 문제는 태스크 사이의 선행 관계를 고려하지 않으며 모든 태스크가 시간 *0*에서부터 실행이 가능하다는 점에서 본 논문에서 다루는 문제와 다르다. 또한 처리기 수의 최소화도 고려하지 않았다. Yalaoui와 Chu[2]에 따르면, 그들이 풀고자 하는 문제는 적어도 NP-hard이다. 왜냐하면 Du와 Leung이 처리기가 한 개일 경우에 이 문제가 NP-hard임을 증명하였고[3], Lenstra와 그 동료가 두 개의 처리기에 대해서 NP-hard임을 증명하였기 때문이다[4]. 그렇지만 현재까지 세 개 이상의 처리기의 경우에 대해서는 정확하게 이 문제의 복잡도(complexity)가 알려져 있지 않다. 여기서 본 논문이 고려하는 스케줄링 문제가 적어도 NP-hard임을 알 수 있다. 왜냐하면 태스크의 선행 관계를 고려하지 않은 스케줄링 문제는 그것을 고려하는 문제로 다항 시간(polynomial time)에 환원(reduction)할 수 있기 때문이다.

한편, 본 논문에서 고려하는 두 가지 목적을 살펴보면, 처리기의 수를 줄이면 마감시간 지연의 총합이 늘어날 수 있고, 후자를 줄이면 전자가 늘어날 수 있다. 본 문제는 이와 같이 서로 상충하며 같은 단위로 측정할 수 없는 두 개의 목적을 동시에 독립적으로 최소화하는 것을 요구한다. 그러나 기존 연구는 주로 마감시간 지연만 고려하거나 혹은 두 개의 목적을 고려한다고 하더라도 이 두 가지 목적을 결합하여 하나의 비용 함수로 변환하여 최적화를 하고자 하였다. 예를 들면 Coli 와 Palazzari는 마감시간 지연의 총합과 사용하는 처리기 수의 곱을 최소화하고자 하였다[5].

하나의 비용 함수로 결합하는 방식은 하나의 절충적인 해를 찾으며 의사 결정자의 개입이 필요 없다는 장점을 지닌다. 그러나 적절한 결합 함수를 결정하기 위해서는 여러 번의 시행착오를 필요로 한다. 왜냐하면 결합 함수가 최적화 이전에 알려지지 않은, 문제의 특징을 배제했거나 또는 결합 함수에서 요구하는 계수(coefficient)들을 부적절하게 선택했을 가능성이 있기 때문이다[6]. 이와 달리 [6-10]에서는 상충하는 목적들을 독립적으로 고려하며 주어진 목적들을 동시에 최적화시키는 해를 구하기 위해서 일반적인 유전 알고리즘을 수정하였고 최종적으로 해의 집합으로 이루어진 파레토 최적해 집합(Pareto-optimal set)을 찾고자 하였다. 본 논문에서는 이러한 알고리즘을 다목적 유전 알고리즘이라 부르도록 하겠다.

본 논문에서 정적 태스크 스케줄링을 위한 다목적 유전 알고리즘을 제안한다. 이 알고리즘은 두 가지 목적을 정량적으로 표현할 때, 스칼라가 아닌 이차원 벡터 형태를 취하여 각각의 목적을 독립적으로 고려한다.

논문의 구성은 다음과 같다. 먼저 2장에서 기존에 제안된 스케줄링 알고리즘에 관하여 살펴본다. 3장에서 태스크 스케줄링을 위한 다목적 유전 알고리즘을 제안한다. 실험 결과를 통해 본 알고리즘의 유용성을 4장에서 설명하며, 마지막으로 5장에서 결론을 내린다.

2. 관련 연구

1장에서 언급하였듯이 논문에서 고려하는 문제는 NP-hard이다. 따라서 수많은 휴리스틱들(heuristics)이

제안되었다. 잘 알려진 세 가지 휴리스틱으로는 반복적 향상 알고리즘(iterative improvement algorithm), 확률적 최적화 알고리즘(probabilistic optimization algorithm)과 건설적인 휴리스틱(constructive heuristic)이 있다.

첫 번째 반복적 향상 알고리즘은 지역 최적(sub-optimal) 해를 가지고 시작하여 이 해에 대해 반복적으로 지역 변경(local change)을 시도한다. 이 부류의 알고리즘의 수행시간은 길지 않지만 지역 최적해에 빠지는 경향이 있다. 타부 서치(Tabu search)는 이런 부류에 속하는 알고리즘으로 최근에 방문한 적이 있는 해들의 목록을 유지하면서 이들에 대한 반복적인 방문을 피함으로써 공간 탐색의 효율을 꾀한다. 이 부류의 알고리즘이 [11]에 사용되었다.

두 번째 확률적 최적화 기법으로는 시뮬레이티드 어닐링(Simulated annealing, SA)과 유전 알고리즘(Genetic algorithm, GA)이 아주 많이 연구되었다. 두 가지 기법은 모두 지역 최적점(local optima)을 피할 수 있는 능력을 지니고 있다. SA는 스케줄링 문제 해결을 위해 [5]에 사용되었고, [11-14]에서는 GA를 사용하였다.

Coli와 Palazzari는 선행 관계와 마감시간을 지닌 태스크들을 선점 없이 다중 처리기 시스템에서 스케줄링하기 위하여 SA를 이용하였다[5]. 스케줄의 비용은 사용한 처리기 수와 마감시간 지연 총합의 곱으로 정의하였다. 이 기법은 SA를 이용하여 두 목적을 동시에 독립적으로 최소화하기보다는 두 목적이 결합된 비용 함수를 최소화하고자 하였다.

Mitra와 Ramanathan은 선행 관계와 마감시간을 지닌 태스크들을 선점 없이 다중 처리기 시스템에서 스케줄링하기 위하여 GA를 이용하였다[12]. 또한 Minimum-laxity-first 휴리스틱과 성능 비교를 하였다. 이 기법은 태스크 사이의 통신 시간을 고려하였다. 스케줄의 비용은 태스크들의 마감시간 지연 중에서 가장 큰 값(maximum tardiness of tasks)으로 정의하였으며, 이 비용을 최소화하는 스케줄을 찾는 것이 목적이다. 사용하는 처리기 수의 최소화를 고려하지 않았다.

Monnier와 그의 동료는 선행 관계와 마감시간을 지닌 태스크들을 선점 없이 다중 처리기 시스템에서 스케줄링하기 위하여 GA를 이용하였다[13]. [11,12]의 기법들과 달리 주기적인 태스크와 버스 충돌(bus contention)을 고려하였다. 스케줄의 비용은 후행자 태스크(successor task)가 없는 태스크들의 마감시간 지연의 합으로 정의하였으며 이러한 합이 최소화되는 스케줄을 찾고자 하였다. 이 기법의 성능을 리스트 스케줄링(list-scheduling) 알고리즘, SA, 그리고 클러스터링 알고리즘과 비교하였다. 후행자가 있는 태스크의 마감시간을 고려하지 않았다는 제한점을 지니며 사용하는 처리기 수의 최소화를 고려하지 않았다.

세 번째로 건설적인 휴리스틱은 최선의 정책(best-effort policy)을 이용하여 가능한 해(feasible solution)를 찾으려고 한다. 이 부류의 기법으로 리스트 스케줄링 알고리즘이 많이 연구되었다[15-20]. 리스트 스케줄링 알고리즘은 어떤 태스크를 어떤 처리기에 할당해야 할지 결정하기 위하여 우선순위 리스트(priority list)를 사용한다. 리스트 스케줄링 알고리즘은 이러한 리스트를 생성하기 위하여 사용하는 우선순위 함수에 따라 다양하다. 일반적으로 이 알고리즘은 적은 차수의(low-order) 다항 시간 안에 수행되지만 스케줄의 품질은 우선순위 리스트에 놓인 태스크들의 순서에 많은 영향을 받는다. 또한 마감시간 지연보다는 스케줄의 길이(makespan)가 주요한 최적화 목적이다.

Altenbernd와 Hansson은 경성 실시간 태스크(hard real-time task)들을 스케줄링하기 위하여 Slack Method라는 건설적 휴리스틱을 제안하였다[21]. 이 기법은 리스트 스케줄링과 같은 건설적 휴리스틱들과는 달리 태스크의 마감시간을 고려한다. 마감시간 지연을 줄이고 사용하는 자원의 수를 줄이는 두 가지 목적을 가지고 있지만, 주로 전자에 초점을 맞추고 있다.

3. 다목적 유전 알고리즘을 이용한 태스크 스케줄링 기법

서론에서 언급하였듯이 실시간 응용 프로그램을 방향이 있고 순환이 없는 그래프(directed acyclic graph), $G = (T, E)$로 표현할 수 있으며, 이 경우 그래프의 노드(node)는 태스크를 의미하고 선행 관계는 간선(edge)으로 표현한다.

태스크 그래프, $G = (T, E)$에 대해서 앞으로 다음과 같은 정의를 사용할 것이다.

> **정의 1.** 태스크 그래프 G에서 태스크 t_i로부터 태스크 t_j로 가는 방향 경로(directed path)가 존재하면 태스크 t_i를 태스크 t_j의 ***선행자(predecessor)***라 한다. 그래프 G에 간선 $<t_i, t_j>$가 존재하면 태스크 t_i를 태스크 t_j의 ***직속 선행자(immediate predecessor)***라 한다. 태스크 t_i가 태스크 t_j의 선행자라면 t_j는 t_i의 ***후행자(successor)***이다. 태스크 t_i가 태스크 t_j의 직속 선행자라면 t_j는 t_i의 ***직속 후행자(immediate successor)***이다. $P(t_i)$는 태스크 t_i의 직속 선행자들의 집합이며, $S(t_i)$는 태스크 t_i의 직속 후행자들의 집합이다. $P(t_i)$가 공집합이면 t_i를 ***시작 태스크(entry task)***라 한다. $S(t_i)$가 공집합이면 t_i를 ***종료 태스크(exit task)***라 한다.

스케줄링 기법을 간결하게 설명하기 위해 태스크의 ***가장 이른 시작 시간(earliest start time)***과 ***가장***

늦은 시작 시간(latest start time)을 정의한다. 태스크 t_j의 가장 이른 시작 시간(***estStT[t_j]***)은 시작 태스크로부터 t_j로 가는 가장 긴 경로의 길이를 의미한다. 예를 들면, 그림 1(a)에서 태스크 t_2의 가장 이른 시작 시간은 5이다. 태스크 t_j의 가장 늦은 시작 시간(***lstStT[t_j]***)은 t_j와 t_j의 모든 후행자들이 각각의 마감시간을 만족하면서 t_j가 수행을 시작할 수 있는 가장 늦은 시간을 의미한다. 예를 들면, 그림 1(a)에서 태스크 t_1의 가장 늦은 시작 시간은 7이다. 왜냐하면 그 시간 이후에 t_1이 시작하면 후행자인 t_2가 마감시간을 어기기 때문이다. 이러한 두 시작 시간을 다음과 같이 정형적으로 정의할 수 있다.

정의 2. 태스크 t_j의 가장 이른 시작 시간(***estStT[t_j]***)을 다음과 같이 정의한다.

$$estStT[t_j] = \begin{cases} 0 & if \neg\exists\, t_i : (t_i, t_j) \in E \\ \max\limits_{t_i \in Pr(t_j)} \{estStT[t_i] + e_i\} & otherwise \end{cases}$$

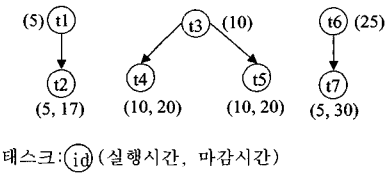
정의 3. 태스크 t_j의 가장 늦은 시작 시간(***lstStT[t_j]***)을 다음과 같이 정의한다.

$$lstStT[t_j] = \begin{cases} d_j - e_j & if \neg\exists t_i : (t_j, t_i) \in E \\ \min\left\{ \min\limits_{t_i \in S(t_j)} \{lstStT[t_i] - e_j\}, d_j - e_j \right\} & otherwise \end{cases}$$

3.1 문제 표현과 비용 함수

유전 알고리즘에서 모집단의 각 개체는 문자열이나 염색체(chromosome)로 표현(encoding)되며 주어진 문제의 가능한 해를 의미한다. 스케줄링 문제에서 하나의 염색체는 처리기들에 태스크들을 할당하는 모든 가능한 방법들 중 하나이다. 그림 1(a)의 태스크 그래프가 주어지고 p_1, p_2, 그리고 p_3이 동일한 처리기라고 가정할 때, 가능한 두 가지 해가 그림 2에 나와 있다. 해 i는 $sched_i$와 $alloc_i$로 나뉘는데 $sched_i$는 스케줄 순서를 의미하며 $alloc_i$는 할당 정보를 표현한다. $sched_i$와 $alloc_i$의 길이는 태스크들의 총 개수와 같다. 타당한 스케줄이 되려면 $sched_i$가 주어진 태스크 그래프에 나타나는 선행 관계를 모두 만족하여야 한다. $alloc_i[k]$는 태스크 $sched_i[k]$가 할당된 처리기를 표시한다. 예를 들어 그림 2(a)의 $alloc_i[6]$은 t_2(즉 $sched_i[6]$)가 p_1에 할당된다는 것을 표현한다.

태스크 마감시간 지연(tardiness)을 계산하여 각각의 해를 평가하여보자. 태스크 t_i의 마감시간 지연은 $\max\{0, (\gamma_i - d_i)\}$이다. 그림 3에 나타나 있는 해의 평가 프로시저를 보면, 루프를 반복할 때마다 우선순위 리스트 *sched*에서 순위가 가장 높은 태스크 *sched[i]*를 꺼



(a) 태스크 그래프

태스크 ID	가장 이른 시작 시간	가장 늦은 시작 시간
t1	0	7
t2	5	12
t3	0	0
t4	10	10
t5	10	10
t6	0	0
t7	25	25

(b) 가장 이른 시작 시간과 가장 늦은 시작 시간

그림 1 태스크 그래프와 가장 이른 시작 시간 및 가장 늦은 시작 시간

	[1]	[2]	[3]	[4]	[5]	[6]	[7]
$sched_i$	t6	t3	t1	t5	t4	t2	t7
$alloc_i$	p2	p1	p3	p3	p1	p1	p1

(a) t_2의 마감시간을 지키지 못하는 해 i

	[1]	[2]	[3]	[4]	[5]	[6]	[7]
$sched_j$	t3	t6	t1	t4	t5	t2	t7
$alloc_j$	p1	p2	p3	p1	p3	p3	p1

(b) 모든 마감시간을 지키는 해 j

그림 2 해의 표현

1) **do** {
2) Remove the first task, say t_i (*sched[i]*), from *sched*
3) Schedule t_i in processor *alloc[i]* using the insertion approach to allow t_i to start the earliest;
4) Compare the finish time of t_i against its deadline to compute its tardiness
5) } **until** (all the tasks are scheduled)

그림 3 해의 평가

(a) 그림 2(a)를 간트 차트로 표현 (b) 그림 2(b)를 간트 차트로 표현

그림 4 그림 2의 스케줄을 간트 차트(Gantt chart)로 표현

내 할당된 처리기(*alloc[i]*)에서 삽입 방법을 이용하여 스케줄한다. 삽입 방법에서는 태스크 *sched[i]*는 처리기 *alloc[i]*에서 비어있으면서 가장 이르게 시작할 수 있는 타임 슬롯에 할당된다. 그림 4는 간트 차트를 사용하여 그림 2에 나타난 두 가지 해가 어떻게 처리기에 할당되고 스케줄되는지를 보여준다. 그림 4(a)를 보면 t_2가 마감시간을 어기고 있음을 알 수 있다. 그림 2(b)를 보면 t_2의 스케줄링 우선순위가 t_5보다 더 낮다. 그러나 그림 4(b)에서는 t_2가 처리기 p_3에서 t_5보다 먼저 시작됨을 알 수 있는데 이것은 위와 같은 삽입 방법을 사용하기 때문이다.

유전 알고리즘에서 각 해의 품질을 측정하기 위해 *비용 함수(cost function)*가 필요하다. 본 논문에서 비용 함수는 벡터 값을 가지는 함수로 $F = \langle f_1, f_2 \rangle$이다. 여기서 f_1은 사용하는 처리기의 수이며 f_2는 마감시간 지연의 총합이다. 예를 들면, 그림 2(a)가 표현하는 스케줄은 처리기 p_1, p_2, p_3를 사용하고 있으므로 $f_1 = 3$이고, t_2만이 마감시간을 어기며 t_2의 마감시간 지연이 8이므로 $f_2 = 8$이다. f_1과 f_2를 결합한 하나의 인자로 해의 품질을 측정하지 않고 우세(domination) 개념을 가지고 해의 품질을 측정할 것이다. 우세 개념이란 다음과 같다.

정의 4. $F(a) = \langle f_1(a), f_2(a) \rangle$와 $F(b) = \langle f_1(b), f_2(b) \rangle$를 비용 함수 F의 벡터 값이라 하자. 각각은 해 a와 해 b의 품질을 측정한 값이다. 모든 목적 함수 f_i ($i = 1, 2$)에 대해 $f_i(a) \leq f_i(b)$이고 적어도 하나의 목적 함수 f_j에 대해 $f_j(a) < f_j(b)$이면 a가 b보다 ***우세***하다고 한다.

3.2 유전 알고리즘

이제 해 영역을 탐색하는데 사용할 *RT-MGA*라 하는 유전 알고리즘을 제시한다(그림 5).

```
1) Create initial Population of fixed size;
2) do {
3)         Select two solutions Parent1 and Parent2 from
Population;
4)      Child ← crossover(Parent1, Parent2);
5)      Mutation(Child);
6)      Local-improvement(Child);
7)      Replace(Population, Child);
8) } until (stopping condition is satisfied)
9) Return the set of best solutions in Population;
```

그림 5 태스크 스케줄링을 위한 유전 알고리즘

본 논문에서 제시하는 이 유전 알고리즘은 임의의 타당한 해로 구성된 초기 *모집단(population)*을 가진다. 이 모집단은 종료 조건을 만족할 때까지 다른 모집단으로의 진화를 반복한다. 진화가 멈추면 알고리즘은 최종 모집단에 존재하는 최선의 해들을 반환한다. 각 반복 과정에서 진화는 다음과 같이 진행된다. 모집단에서 특정 확률 분포에 근거하여 부모가 되는 두 원소를 *선택(selection)*한다. 이 둘은 *교차(crossover)* 연산자를 통해 결합되어 자식을 낳게 된다. 자식은 *변이(mutation)* 연산자에 의해 변형되는데, 경험하지 못한 탐색공간을 모집단에 도입시키기 위해서이다. 그 다음에 자식에 대해 *지역 개선(local improvement)* 과정을 수행한다. 마지막으로 *대체(replacement)* 연산자에 의해 모집단에서 하나의 해가 선택되어 자식과 대치된다. 다음은 초기 모집단의 생성을 비롯한 각 연산자들에 관해 기술한 것이다.

초기 모집단의 생성: 해 i의 $sched_i$는 태스크 그래프에 대해 임의로 생성되는 위상 순서(topological order)이며 각 태스크에는 임의로 선택된 처리기를 할당하며 이러한 할당 정보는 $alloc_i$로 표현된다.

선택 연산: 대표적인 선택 방법인 품질 비례 룰렛휠 선택법(proportional roulette-wheel selection)[22]을 사용한다. 본 논문에서는 가장 우수한 해가 선택될 확률이 가장 열등한 해가 선택될 확률의 4배가 되도록 각 해들의 선택 확률들을 정규화한다. 우세 관계는 부분 순서 관계이기 때문에 품질 비례 룰렛휠 선택법을 사용하기 위해 해를 품질 측면에서 완전 순서화할 필요가 있다. 이를 위해 정의 4의 우세 개념에 기반을 두어 해의 품질을 측정한다[7]. 즉 만일 세대 t에서 해 x_i보다 $p_i^{(t)}$ 개의 다른 해가 우세하다면 해 x_i의 품질은 $1 + p_i^{(t)}$와 같이 주어진다.

교차 연산: 전통적인 일점 교차(one-point crossover)는 교차점을 임의로 정한 다음 부모 p와 q의 각 부분을 서로 바꾸어 자식 c를 생산한다. 일점 교차는 온당치 못한 해를 생산할 수도 있기 때문에, 즉 자식 c의 $sched_c$가 위상 순서를 만족하지 못할 수 있기 때문에 일점 교차 수정안(그림 6)을 사용한다. 부모 p와 q가 주어질 때, 먼저 임의로 한 개의 교차점(k)을 정한 다음 교차점 왼쪽 부분을 p로부터 자식 c에 복사한다. c의

```
// sched_p, alloc_p: chromosome of parent p
// sched_q, alloc_q: chromosome of parent q
// sched_c, alloc_c: chromosome of child c
// n: the number of tasks
1) Generate a crossover point k randomly, 1≤k < n;
2) The left segments of sched_c and alloc_c of child c are inherited directly from sched_p and
alloc_p, respectively;
3) Set m to k;
4) for i = 1 to n { // tasks in sched_q of parent q are visited from the leftmost task to the
rightmost one
5)     Let t_i be the i^th task in sched_q of parent q;
6)     if t_i ≠ sched_p[j] for ∀j (= 1,2, ..., k), then {
7)         sched_c[m+1] and alloc_c[k+1] are copied from sched_q[i] and alloc_q[i], respectively;
8)         Increase m by 1;
9)     }
10) }
```

그림 6 일점 교차 수정안

나머지 부분은 q로부터 복사하되, q를 처음부터 읽어가며 c에서 아직 사용하지 않은 기호들을 q에서 나타난 순서대로 복사한다. 본 논문에서 정의한 프로시저를 두 부모에게 적용하여 생산한 자식 해는 위상 순서를 만족한다.

교차점 좌우 부분에 대칭성을 주기 위해 그림 6 프로시저에서 left와 right를 바꾸어서 수정한 또 다른 교차 프로시저 또한 사용한다. 여기서 자식은 오른쪽에서 왼쪽으로 진행하며 생산된다. 그림 1(a)의 태스크 그래프를 가지고 일점 교차 수정안을 적용한 예를 그림 7에 나타내었다.

변이 연산: 염색체 *alloc*에서 임의로 유전자 하나를 선택한 다음 그 값을 임의의 새로운 값으로 변경한다.

대치 연산: 자식이 두 부모 중 어느 하나 보다 우세하면 그 부모를 자식으로 대치한다. 그렇지 않다면 모집단에서 가장 열등한 것과 대치한다. 그 이유는 너무 많은 시간을 소비하지 않으며 모집단의 다양성을 유지하기 위해서이다[22].

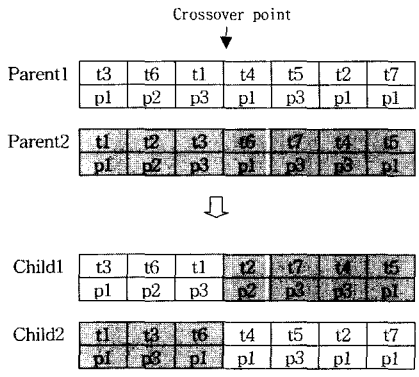


그림 7 교차 연산자를 적용한 결과

3.3 지역 개선 휴리스틱(Local improvement heuristic)

유전 알고리즘은 지역 최적점으로의 접근 속도가 일반적으로 느리다. 정도의 차이는 있지만 교차와 변이 연산이 임의적으로 이루어지므로 지역 최적점 근처에서의 미세 조정(fine tuning) 능력이 떨어지기 때문이다. 이를 보완하기 위해 교차와 변이로 만들어진 해에 지역 개선 알고리즘을 적용한다. 이렇게 하면 교차와 변이는 해를 지역 최적점 근처에 갖다 놓는 역할을 하고 지역 개선 알고리즘은 해를 지역 최적점으로 안내한다[22,23]. 이러한 이유로 본 논문에서는 세 가지 지역 개선 휴리스틱을 제안한다.

첫 번째로 비용 함수 F에 f_3을 추가한다. 처리기에서 수행하는 태스크들의 총 수행 시간을 해당 처리기의 부하라고 하자. f_3은 처리기 간의 부하 균형을 측정하기 위한 것으로 부하의 분산(variance)을 의미하여 수식은 다음과 같다.

$$f_3 = \sum_p (\sum_i e_i \cdot x_{ip} - \sum_j e_j/N)^2/N$$

여기서 N은 사용하는 처리기의 수이며, 태스크 i가 처리기 p에 할당되면 $x_{ip} = 1$이고 그렇지 않으면 0이다. f_3의 추가로 우세 개념을 다시 정의한다. 해 a가 해 b보다 우세할 조건(정의 4 참조)에 다음의 경우를 추가한다. 두 개의 목적 함수 f_i (i = 1, 2)에 대해 $f_i(a) = f_i(b)$이고 목적 함수 f_3에 대해 $f_3(a) > f_3(b)$이면 a가 b보다 ***우세***하다고 한다. 큰 f_3 값을 가지는 해의 의미는 작업 부하가 특정 처리기에 치우쳐 있어서 적은 부하를 가지는 처리기가 태스크 스케줄링에 불필요할 수 있다

```
// num_task[j]: the number of tasks allocated to processor j
// proc_set: {j: num_task[j] > 0}
1) Find the processor m ∈proc_set such that num_task[m] ≤ num_task[i] for∀ i∈ proc_set ;
2) Find the processor m_next ∈proc_set - {m} such that num_task[m_next] ≤ num_task[i] for
∀ i ∈proc_set - {m};
3) Randomly select r tasks allocated in the processor m (r ≤ num_task[m]);
4) Reallocate the r tasks to the processor m_next ;
```

(a) f_1을 개선하는 휴리스틱

```
// sched, alloc: chromosome of input solution s
1)  for i = 1 to #_of_tasks { /* 1st phase: reordering */
2)  for j = i to 2 step -1 {
3)          if solution s satisfies all the task deadlines, then return;
4)          Let n_{j-1} and n_j be the j-1th and j th task in sched of solution s, respectively;
5)          if n_j starts not later than its latest start time, then break;
6)          if n_{j-1}'s latest start time is greater than n_j's latest start time then {
7)                  Swap the genes sched[j] and sched[j-1];
8)                  Swap the genes alloc[j] and alloc[j-1];
9)          }
10)         else break;
11)     }
12) }
13) for i = 1 to #_of_tasks { /* 2nd phase: reallocation */
14)     Let n_i be the i th task in sched of solution s ;
15)     if n_i starts later than its latest start time, then {
16)         Find the processor j such that mRStT_j[n_i] ≤ mRStT_k[n_i] for any processor k,
                where mRStT_k[n_i] denotes the start time of n_i when n_i is reallocated to
processor k;
17)         Allocate the task n_i to the processor j ;
18)     }
19) }
```

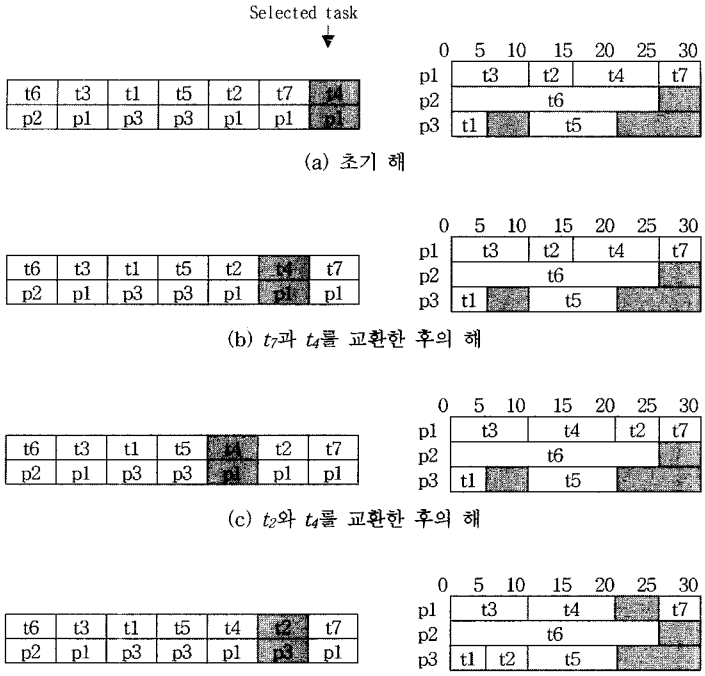
(b) f_2를 개선하는 휴리스틱

그림 8 지역 개선 휴리스틱

고 해석될 수 있다. 큰 f_3 값을 가지는 해를 선호하는 이유는 우리의 유전 알고리즘이 최소 개수의 처리기를 사용하는 해로 더 빨리 수렴할 수 있기 때문이다.

두 번째 휴리스틱은 태스크 재할당에 의해 사용하는 처리기의 개수(f_1)를 최소화하려고 시도한다. 가장 적은 수의 태스크를 수행하는 처리기 m을 선택하고 그 태스크들 중 몇 개를 다른 처리기 *m_next*에 이동시킨다. *m_next*는 처리기 m을 제외했을 때 가장 적은 수의 태스크를 수행하는 처리기이다. 이렇게 하는 이유는 사용하는 처리기의 개수를 줄일 가능성이 있기 때문이다. 이 연산은 그림 8(a)에 기술하였다.

세 번째 휴리스틱은 태스크들의 마감시간 지연의 총합(f_2)을 최소화하려는 시도로 해의 스케줄 순서를 다시 정하고(그림 8(b)의 라인 1-12) 태스크들을 더 나은 처리기로 재할당한다(라인 13-19). 첫 단계(순서 조정 단계)에서 *sched*에 있는 태스크들을 왼쪽 끝에서부터 오른쪽 끝까지 가면서 순서대로 방문한다. 라인 2에서 11 사이는 각 태스크 n_j(*sched[j]*)에 대해 n_j와 그 앞의 태스크를 맞바꿀 수 있는가를 살핀다. 만약 n_j가 그것의 가장 늦은 시작 시간보다 늦게 시작한다면(라인 5) 그 앞의 태스크 n_{j-1}(*sched[j-1]*)의 가장 늦은 시작 시간이 n_j의 것 보다 큰 지를 검사한다(라인 6). 만약 그렇다면 n_j의 시작 시간을 빠르게 하기 위해 n_j와 n_{j-1}을 서로 맞바꾼다(라인 7-8). 이러한 교환 과정은 바뀐 태스크(*sched[j-1]*)에 대해서 계속된다. 한편, n_{j-1}의 가장 늦은 시작 시간이 n_j의 것 보다 크지 않다면(라인 6), 수행 불가능한 해가 만들어질 수 있으므로 교환은 이루어지지 않는다(라인 10). 어떤 태스크가 자신의 가장 늦은



(a) 초기 해

(b) t_7과 t_4를 교환한 후의 해

(c) t_2와 t_4를 교환한 후의 해

(d) t_2를 p_3에 재할당한 후의 해

그림 9 f_2를 개선하는 휴리스틱을 적용한 예

시작 시간보다 늦게 수행을 시작한다면, 그 태스크 또는 그것의 후행자가 마감시간을 어기게 된다는 사실을 주목할 필요가 있다.

두 번째 단계는 자신의 가장 늦은 시작 시간 보다 늦게 시작하는 태스크 n_i를 찾고 그 태스크를 최소의 $mRStT_j[n_i]$ 값을 가지는 처리기 j에 할당한다. 여기서 $mRStT_j[n_i]$는 n_i가 처리기 j에 재할당될 때의 시작 시간을 의미한다(라인 15-18). 이러한 휴리스틱의 목적은 태스크들이 가장 늦은 시작 시간보다 늦게 시작하지 않도록 하기 위해 순서 조정과 재할당을 통해 시작 시간을 앞당기려는 것이다. 변이 연산자에 의해 수정된 자식에 그림 8(b)의 프로시져를 적용하면 새로운 위상 순서를 표현하는 해가 생성될 것이다. 그림 1(a)의 태스크 그래프를 가지고 f_2 개선 휴리스틱을 적용한 예를 그림 9에 보였다. 그림 9(a)에 초기 해가 주어진다. 첫 단계에서 연산자는 태스크 t_6에서 t_4까지 태스크를 차례로 방문한다. 그림 9(a)의 간트 차트에서 알 수 있듯이 마지막 태스크 t_4만이 가장 늦은 시작 시간보다 늦게 시작하기 때문에 t_4가 선택된다. t_7의 가장 늦은 시작 시간이 t_4의 것보다 크기 때문에 이 둘을 맞바꾼다. 그림 9(b)는 수정되어진 해이다. t_4는 아직도 마감시간을 어기고 있으며 t_2의 가장 늦은 시작 시간이 t_4의 것보다 크기 때문에 그 둘을 맞바꾼다(그림 9(c)). 이제 t_4는 마감시간을 어기지 않으므로 더 이상의 맞교환은 수행되지 않으며 t_5다음의 위치에 놓이게 된다. 첫 단계를 거쳐 수정된 해에서는 t_2의 마감시간이 지켜지지 않는다. 두 번째 단계에서는 t_2를 선택하여 처리기 p_3에 재할당한다. 그림 9(d)에 나오는 최종의 해는 모든 태스크들의 마감시간을 만족시킨다.

4. 검 증

이 장에서는 임의로 생성한 태스크 그래프를 가지고 본 논문에서 제안하는 알고리즘의 성능을 평가한다.

지금까지 스케줄의 길이(makespan), 태스크 마감시간 지연, 또는 처리기 수의 최소화와 같은 다중 목적을 동시에 만족시킬 수 있는 태스크 스케줄링 방법은 알려져 있지 않다[24]. 기존 대부분의 방법은 여러 목적들 중 하나만 고려하는 경향이 있다. 이러한 이유 때문에 먼저 *LSTF*(Least space-time first)[19]을 선택하여 제안한 알고리즘의 성능을 확인하겠다. LSTF는 Highest- level-first[15], Earliest-deadline-first[16], 그리고 Least-laxity-first[17] 등과 같은 기존의 휴리스틱 알고리즘보다 태스크들의 마감시간 지연의 최댓값(maximum tardiness of tasks)을 최소화하는 면에서 성능이 우수하다고 알려져

```
// nOfTasks: the number of tasks in a task graph
// listAlg(n): a list-scheduling algorithm whose input n denotes the number of processors
that can be used by the algorithm
1) Set left and right to 1 and nOfTasks, respectively;
2) Set minProc to nOfTasks;
3) while (left ≤right)
4)     middle = (left + right) / 2;
5)     if listAlg(middle) finds a schedule that satisfies all the task deadlines, then {
6)         minProc = middle;
7)         right = middle - 1;
8)     }
9)     else left = middle + 1;
10) }
11) Return minProc;
```

그림 10 리스트 스케줄링 알고리즘에서 사용하는 처리기의 최소 개수를 알아내기 위한 프로시저

있기 때문이다. Monnier 등이 제안한 유전 알고리즘인 *Monnier-GA*[13]를 확률적 최적화 기법들 가운데서 비교를 위해 선택하였는데, 이유는 이 알고리즘이 마감시간 지연의 총합(total tardiness of tasks)의 최소화를 시도하였고 다른 특정 휴리스틱 알고리즘보다 우수하였기 때문이다. 또한 다음의 알고리즘들도 비교 대상으로 고려하였다. 이 방법들이 단지 태스크들의 마감시간 지연의 최소화만을 고려하기는 하지만 여러 연구[13,21]에서 성능 비교에 사용되어왔기 때문이다.

SList-Est*와 *LSTF: 이것들은 미리 계산된 우선순위에 따라 태스크들을 스케줄하는 정적 리스트 스케줄링 알고리즘(static list-scheduling algorithm)이다. SList-Est는 가장 이른 시작 시간이 작은 태스크에게 높은 우선순위를 부여한다. 반면에 LSTF는 가장 늦은 시작 시간이 작은 태스크에게 높은 우선순위를 부여한다. 각 스케줄링 단계에서 우선순위 리스트에서 가장 순위가 높은 태스크가 제거되어 가장 일찍 시작할 수 있는 처리기에 할당된다.

ETF-Est*와 *ETF-Lst: 이것들은 동적 리스트 스케줄링 알고리즘(dynamic list-scheduling algorithm)이다. 각 스케줄링 단계에서 실행이 가능한, 모든 태스크들을 임시로 각 처리기에 할당한 다음 그 중에 최선의 쌍을 선택한다. ETF[18]에서 최선의 쌍이란 가장 일찍 시작할 수 있는 태스크와 해당 처리기를 의미한다. 두 개의 태스크가 동시에 시작할 수 있을 때는 정적 우선순위가 높은 태스크를 우선 스케줄한다. ETF-Est에서는 가장 이른 시작 시간을 가진 태스크에 높은 정적 우선순위를 부여한다. 반면에 ETF-Lst에서는 가장 늦은 시작 시간이 작은 태스크에 높은 정적 우선순위를 부여한다.

각 리스트 스케줄링 알고리즘이 요구하는 처리기의 최소 수를 알아내기 위해 이진 탐색과 유사한 프로시저를 사용한다(그림 10). *left*는 검사할 처리기의 최소 개수를 의미하며 *right*는 최대 개수를 의미한다. *left*의 초기 값은 *1*이며 *right*는 태스크 그래프에서 태스크들의 개수로 초기화된다. *middle*은 *left*와 *right*의 중간 값이다. 탐색은 *middle* 개수의 처리기로 리스트 스케줄링 알고리즘이 모든 태스크의 마감시간을 만족하는 스케줄을 만드는지를 검사하는 것이다. 만약 그러하다면 *middle*을 *minProc*에 저장해 두고 *right*를 *middle-1*로 설정한다. 그리고 계속해서 탐색을 진행한다. 왜냐하면 스케줄링 알고리즘이 요구하는 처리기의 최소 개수는 *middle*보다 크지 않기 때문이다. 그렇지 않다면 *left*를 *middle+1*로 설정하고 탐색을 계속 진행한다. 이 경우 처리기의 최소 개수는 *middle*보다 클 것이기 때문이다. 탐색이 끝나면 *minProc*을 반환한다.

그림 10의 프로시저는 Monnier-GA의 경우에는 적당치 않은데 이 알고리즘은 리스트 스케줄링 알고리즘보다 훨씬 많은 수행시간을 요구하기 때문이다. Monnier-GA의 경우에는 처리기의 최소 개수를 알아내기 위해 다음 방법을 사용한다. *minNOfProc*을 네 개의 리스트 스케줄링 알고리즘과 RT-MGA가 사용하는 최소 처리기 수들 중에서 최솟값이라고 하자. *nOfProc*의 초기 값은 *minNOfProc*으로 설정한다. 먼저 *nOfProc* 개수의 처리기로 Monnier-GA가 모든 태스크의 마감시간을 만족하는 스케줄을 찾는지를 검사한다. 그러하다면

*nOfProc*을 *1* 감소시키고 계속 탐색을 진행한다. Monnier-GA가 그러한 스케줄을 찾지 못할 때까지 계속한다. 그렇지 못한 경우에는 *nOfProc*을 *1* 증가시키고 Monnier-GA가 그러한 스케줄을 찾을 때까지 탐색을 계속한다. 탐색이 끝나면 *nOfProc*을 반환한다.

우리는 리눅스 운영체제가 탑재된 펜티엄III 650에서 스케줄링 알고리즘을 구현하였다. 실험을 수행하면서 다음과 같은 요인들을 가정하였다. 모집단의 크기는 *70*이며 교차 확률은 *1.0*이고 변이 확률은 *0.015*이다. 종료 조건은 모집단의 *90%*가 같은 품질의 해들로 채워질 때이다. 그것들의 염색체가 똑 같을 필요는 없다. 여기서 두 해가 같은 품질이란 의미는 f_1과 f_2의 값이 똑같다는 의미이다. 어떤 경우든지 반복 횟수는 *25,000*회를 넘지 못한다.

4.1 태스크 그래프 생성

본 논문의 알고리즘을 검증할 수 있는 벤치마크가 부족하여 임의로 생성한 다양한 태스크 그래프들을 가지고 시험하였다. 기존의 관련 연구를 보면 임의 그래프를 사용했던 다수의 예가 있다[13,20].

실험을 위해 기존 문헌을 참고하여 다음과 같은, 여러 실제 알고리즘을 표현하는 태스크 그래프 유형을 선택하였다.

1) *CHAIN*: 모든 태스크는 많아야 하나의 직속 선행자와 직속 후행자를 가질 수 있다.
2) *BTREE*: 모든 태스크는 많아야 하나의 직속 선행자와 두 개의 직속 후행자를 가질 수 있다.
3) *RBTREE*: 모든 태스크는 많아야 두 개의 직속 선행자와 하나의 직속 후행자를 가질 수 있다.
4) *LATTICE*: 모든 태스크는 많아야 두 개의 직속 선행자와 두 개의 직속 후행자를 가질 수 있다.
5) *GEN*: 모든 태스크는 임의 수의 직속 선행자와 직속 후행자를 가질 수 있다.
6) *COM*: 위에 기술된 그래프들 중에서 두 개 이상의 그래프의 결합(union)으로 구성된다.

그림 11은 각 종류 별로 태스크 그래프를 보여 준다.

우리의 태스크 그래프 생성기에서 사용된 요인 값을 표 1에 정리하였다. 태스크 그래프 예제 집합은 태스크의 총 수(# of tasks)에 따라 세 그룹, 즉 E_{32}, E_{64}, 그리고 E_{96}으로 나뉜다. 각 그룹에는 태스크 그래프 종류당 *10* 개씩의 예제가 존재하며 따라서 총 예제는 *180* 개가 있다. 각 예제에 포함된 태스크 그래프의 개수는 $[min_{\#ofTGs}, max_{\#ofTGs}]$ 사이에서 균등히 분포하도록 하였다. 각 예제에서 하나의 태스크 그래프에 포함된 태스크의 개수는 $[min_{\#ofTasks}, max_{\#ofTasks}]$ 사이에서 균등히 분포하도록 하였다. 태스크의 실행시간은 *[5, 10]* 사이에서 균등히 분포하도록 한다. 태스크 n_i의 마감시간은 $[estStT[n_i], 1.5*estStT[n_i]]$ 사이에서 균등히 분포하도록 하였다.

4.2 성능 평가

여기서 우리는 RT-MGA의 성능을 f_1(사용하는 처리기의 개수)을 기준으로 하여 다른 다섯 가지 알고리즘과 비교한다. *180* 개의 예에 대해, 각 알고리즘별로 모든 태스크 마감시간을 만족하고 최소 수의 처리기를 사용하는 스케줄을 얻도록 한다. 알고리즘 *A*를 태스크 그래프 *TG*에 적용하여 얻은 해의 f_1 값을 $f_1^{min}(A, TG)$라 하자. 만약 $f_1^{min}(A, TG) < f_1^{min}(B, TG)$이면 태스크 그래프 *TG*에 대해서 알고리즘 *A*를 *B*보다 더 우수한 것으로 평가한다.

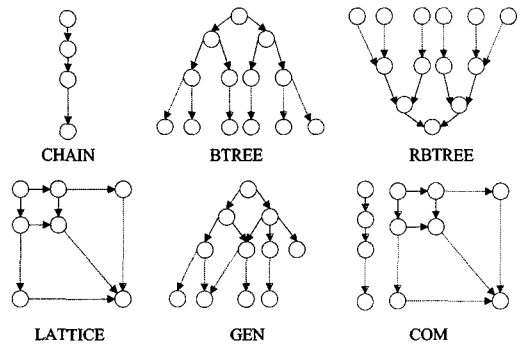


그림 11 유형별 태스크 그래프

실험 결과를 표 2에 나타내어 비교하였다. 표의 왼쪽에 나타난 알고리즘들을 RT-MGA와 비교한다. 표의 각 셀에는 3개의 수가 존재하며 그 수의 앞에는 각각 ">", "=", 그리고 "<"가 전제되어 있다. 이것의 의미는 해당 알고리즘에서 그 개수만큼의 경우가 RT-MGA보다 "더 낫다", "동등하다", 또는 "더 못하다"를 나타낸다. 예를 들어 그룹 E_{96}에서 ETF-Lst와 RT-MGA를 비교해보면 *31* 가지 경우에서 RT-MGA가 보다 우수한 성능을 보이며 *28* 가지 경우에서 동등한 수준이며 *1* 가지 경우만 ETF-Lst가 더 우수하다. 태스크 그래프 유

표 1 임의 예제에서 사용된 요인 값

Example type	# of tasks	min#ofTGs	max#ofTGs	min#ofTasks	max#ofTasks
E32	32	2	2	10	22
E64	64	2	3	10	54
E96	96	2	4	10	86

표 2 RT-MGA와 다른 다섯 가지 알고리즘을 "보다 우수", "같다", 그리고 "보다 나쁨"으로 비교한 성능 비교표

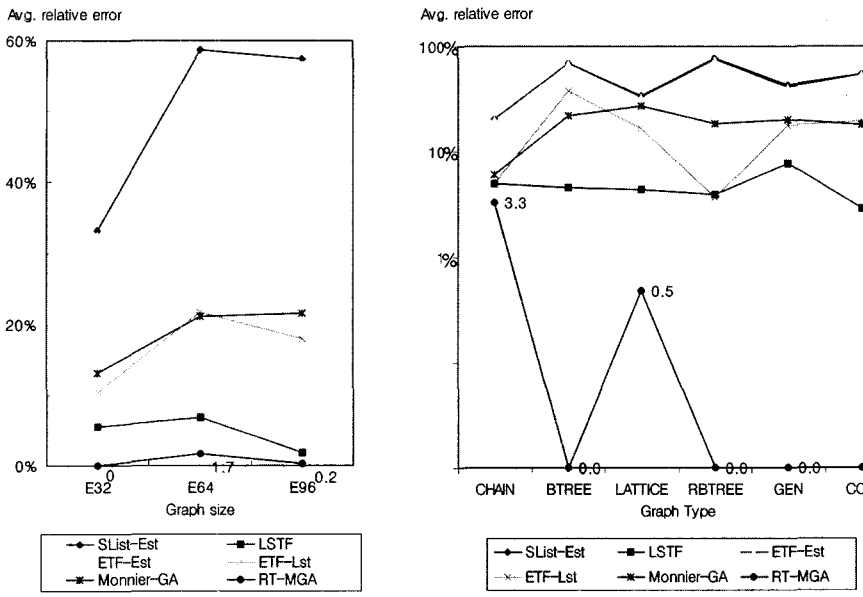
알고리즘	비교 결과 유형	그래프 크기			그래프 유형						총 합
		E32	E64	E96	CHAIN	BTREE	LATTICE	RBTREE	GEN	COM	
SList-Est	>	0	0	0	0	0	0	0	0	0	0
	=	19	9	7	22	0	3	0	6	4	35
	<	41	51	53	8	30	27	30	24	26	145
LSTF	>	0	0	0	0	0	0	0	0	0	0
	=	47	41	47	29	18	23	19	23	23	135
	<	13	19	13	1	12	7	11	7	7	45
ETF-Est	>	0	0	0	0	0	0	0	0	0	0
	=	20	9	7	22	0	4	0	6	4	36
	<	40	51	53	8	30	26	30	24	26	144
ETF-Lst	>	0	0	1	0	0	1	0	0	0	1
	=	43	27	28	29	5	13	20	17	14	98
	<	17	33	31	1	25	16	10	13	16	81
Monnier-GA	>	0	1	0	1	0	0	0	0	0	1
	=	29	13	10	25	0	4	2	14	7	52
	<	31	46	50	4	30	26	28	16	23	127
휴리스틱 적용하지 않은 RT-MGA	>	0	0	0	0	0	0	0	0	0	0
	=	20	9	3	18	0	1	0	9	4	32
	<	40	51	57	12	30	29	30	21	26	148
총 합		60	60	60	30	30	30	30	30	30	180

표 3 최선의 스케줄을 찾은 경우의 수, CPU 소비시간, 오차, 그리고 상대 오차

알고리즘	최선의 스케줄을 찾은 경우의 수		CPU 소비시간 (초)	오차		상대 오차	
	최선의 스케줄	%	평균	합	평균	합	평균
SList-Est	33	18	0.016	778	4.322	89.72	0.498
LSTF	133	74	0.016	47	0.261	8.79	0.049
ETF-Est	34	19	0.260	768	4.267	88.40	0.471
ETF-Lst	98	54	0.257	241	1.339	30.12	0.167
Monnier-GA	53	29	933.906	249	1.383	33.79	0.188
휴리스틱 적용하지 않은 RT-MGA	31	17	56.907	515	2.861	71.44	0.397
RT-MGA	178	99	1483.138	2	0.011	1.14	0.006

형 GEN에서 RT-MGA는 LSTF보다 *7* 가지 경우에서 보다 좋은 성능을 보이고 나쁜 경우는 없으며 *23* 가지 경우에서 동등한 수준이다. 이 표를 보면 모든 태스크 그래프 유형과 크기별 예제 그룹에서 RT-MGA가 다른 다섯 개의 알고리즘과 동등하거나 보다 우수한 성능을 보이고 있음을 알 수 있다. 아울러 RT-MGA는 필요한 처리기의 개수를 감소시키고 있음을 알 수 있다.

표 3은 알고리즘별로 *180* 개의 임의 예 가운데 최선의 스케줄(best found schedule)을 찾은 경우의 수를 보여준다. 태스크 그래프 *TG*가 주어질 때, 최선의 스케줄 *BestSched(TG)*이란, 여섯 개의 알고리즘에서 찾은 해들 중에서 가장 품질이 좋은 스케줄을 의미한다. "CPU 소비시간"은 평균 CPU 소비시간(초)을 나타낸다. 또한 오차의 총합과 평균을 나타냈는데 여기서 태스크 그래프 *TG*가 주어질 때 알고리즘 *A*의 오차를 *Error (A, TG)*라 하고 $f_1^{min}(A, TG)-(BestSched(TG)$의 $f_1)$으로 정의한다. 예를 들어 SList-Est를 사용하여 나온 스케줄들은 *180* 개의 임의 예들의 최선의 스케줄보다 평균적으로 약 4 개 정도 많은 처리기를 사용한다. 이러한 절대 오차는 실험 환경에 의존적이므로 제안한 알고리즘의 유용성을 보이기 힘들 수 있다. 따라서 표 3에 상대 오차를 추가로 보였다. 여기서 태스크 그래프 *TG*가 주어질 때 알고리즘 *A*의 상대 오차는 $Error(A, TG)/(BestSched(TG)$의 $f_1)$으로 정의된다. 예를 들어



(a) 그래프 크기별 상대 오차 평균 (b) 그래프 유형별 상대 오차 평균(로그단위)

그림 12 그래프 크기와 유형별 상대 오차 평균

최선의 스케줄이 N 개의 처리기를 사용한다고 가정할 때 SList-Est에 의해 만들어진 해는 $N*0.498$개 더 많은 처리기를 사용한다는 의미이다. 표 3에 의하면 RT-MGA는 *180* 개의 태스크 그래프 예제 가운데 *178* 개에서 최선의 스케줄을 찾으며, 나머지 *2* 개의 태스크 그래프 예제에 대해서는 최선의 해보다 *1* 개 많은 처리기를 사용하는 스케줄을 발견한다. 상대 오차 평균은 *0* 에 가깝다.

그림 12는 태스크 그래프의 유형과 태스크 그래프의 크기별로 RT-MGA의 성능을 보여 준다. 그림 12(a)에서 RT-MGA의 상대 오차 평균은 그래프 크기가 *32*, *64*, 그리고 *96*일 때 각각 *0%*, *1.7%*, 그리고 *0.2%*이다. RT-MGA의 상대 오차 평균은 다른 모든 알고리즘보다 항상 작다. 그래프 크기가 *32*일 때 상대적으로 각 알고리즘의 상대 오차 평균이 작은 것을 알 수 있는데, 이것은 그래프 크기가 작을수록 태스크 스케줄링 문제의 탐색 공간이 적어지기 때문에 최선의 해를 찾기가 덜 어렵다는 사실로 이해할 수 있다. 그림 12(b)를 보면 그래프 유형이 CHAIN일 때만 RT-MGA의 상대 오차 평균이 *3.3%*이고 그 이외에는 *0.5%*가 가장 큰 값이다. CHAIN 유형의 경우 RT-MGA를 제외한 다른 알고리즘의 상대 오차 평균이 다른 유형보다 상대적으로 작은 값을 보이는데 이것은 태스크간의 상호 관계가 단순하기 때문이다. 태스크 유형이 CHAIN인 경우 RT-MGA의 평균값이 상대적으로 큰 점이 두드러지는데, 표 2에 보이지만 RT-MGA가 그 유형의 태스크 그래프 예 가운데 최선의 해를 찾지 못하는 경우가 한 가지 있기 때문이다. BTREE, LATTICE, RBTREE, GEN, 그리고 COM의 경우에는 명백히 개선되었다.

표 4와 5는 그래프 크기와 유형별로 평균 CPU 소비 시간(초)을 보여준다. 네 개의 리스트 스케줄링 알고리즘 가운데 다른 세 개보다 성능이 우수하기 때문에 LSTF를 선택하였다. LSTF는 결정적(deterministic) 다항 시간 알고리즘이므로 RT-MGA와 Monnier-GA보다 확실히 빠르다. Monnier-GA는 RT-MGA보다 평균적으로 약 *1.6*배 더 빠르다.

최종 해집합의 다양성과 수렴 정도를 평가하기 위해 RT-MGA의 초기 해집합과 최종 해집합을 그림 13에 나타내었다. 이 그림은 그래프 크기가 96, 그래프 유형이 BTREE인 태스크 그래프 예제에 대해서 RT-MGA를 한번 수행한 결과를 나타낸다. x축은 사용한 처리기

표 4 그래프 크기별 CPU 소비 시간(초)

알고리즘	그래프 크기		
	E32	E64	E96
LSTF	0.005	0.015	0.029
Monnier-GA	78.183	588.284	2135.252
RT-MGA	102.502	1042.169	3304.744

표 5 그래프 유형별 CPU 소비 시간(초)

알고리즘	그래프 유형					
	CHAIN	BTREE	LATTICE	RBTREE	GEN	COM
LSTF	0.005	0.015	0.029	0.016	0.018	0.016
Monnier-GA	78.183	588.284	2135.252	1078.750	743.605	1103.750
RT-MGA	102.502	1042.169	3304.744	1666.103	1011.212	1437.383

의 수, *y*축은 마감시간 지연의 총합을 의미한다. 즉, 이 차원 상의 각 점은 스케줄이 갖는 사용한 처리기 수와 마감시간 지연의 총합을 나타낸다. x로 표현한 해는 초기 해, □로 표시한 해는 최종 해를 의미한다. 태스크 스케줄링 문제는 최소화 문제이기 때문에, 이 그래프에서 원점에 가까울수록 최적 해에 근접한 해를 의미한다. 그림 13을 보면 초기 해집합은 탐색 공간에 널리 퍼져 있는 것을 볼 수 있다. 또한 RT-MGA가 초기 해들에 비해 향상된 해들을 최종 해집합으로 갖는 것을 볼 수 있다. 즉, GA의 수행을 통해 최적 해에 가까운 쪽으로 해들이 수렴하였으며 최종 해집합에 열성 해의 수가 초기 해집합에 비해 많이 감소된 것을 알 수 있다. 여기서 해집합 안에서 자신 보다 우세한 해가 존재하는 해를 열성해라 한다. 또한 사용하는 처리기 수 별로 해들이 다양하게 분포된 모습을 볼 수 있다. 다른 5 개의 비교 알고리즘의 경우는 여러 차례의 수행을 통해서야 비로소 다양한 해들을 찾을 수 있다. RT-MGA의 경우는 두 개의 목적을 동시에 독립적으로 고려한 결과로 한번의 수행을 통해 이러한 다양한 해들을 얻을 수 있다.

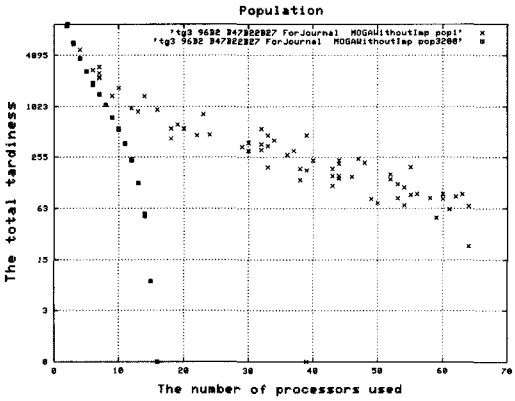


그림 13 그래프 크기=96, 그래프 유형=BTREE인 태스크 그래프 예제에 대한 RT-MGA의 수행 결과

5. 결 론

본 논문에서는 다중 처리기 환경에서 실시간 태스크들의 최적 스케줄링을 정하는 문제를 다루었는데 사용하는 처리기의 개수와 태스크들의 마감시간 지연 총합을 최소화하고자 하였다. 이 것은 같은 단위로 측정할 수 없고 또한 서로 상충하는 두 가지 기준을 동시에 최적화하는 문제이다. 지금까지 두 개의 목적을 동등하게 만족시키는 태스크 스케줄링 방법은 알려져 있지 않다. 기존 대부분의 방법들은 이러한 목적들 중 하나에만 치우치는 경향이 있다. 다른 기존 방법으로는 목적들을 하나의 기준으로 결합시킨 후 최적화하여 이러한 문제를 해결하려는 기법이 있다. 본 논문에서는 다목적 유전 알고리즘을 사용하여 최적화를 수행하였다. 다목적 유전 알고리즘은 두 가지 목적을 독립적인 것으로 간주하므로 벡터 값의 비용함수를 사용한다. 또한 태스크 스케줄링 문제에 잘 맞도록 하기 위해 유전 알고리즘과 관련된 여러 가지를 제안하였다. 문제 표현 전략, 우세 개념에 기초한 선택 연산, 그리고 교차 연산 등이 그것이다. 그리고 지역 개선 작업을 위해 세 개의 휴리스틱을 제안하였으며 이 것을 통해 유전 알고리즘의 수렴성을 높이고자 하였다.

실험 결과를 통해 본 논문에서 제안한 알고리즘이, *180* 개의 임의로 생성된 태스크 그래프 가운데 *178* 개의 경우에서, 리스트 스케줄링 알고리즘들을 포함한 기존의 다섯 가지 알고리즘들과 동등하거나 더 우수한 성능을 보임을 밝혔다. 제안한 알고리즘은 태스크의 유형별 분류를 통한 성능 평가에서도 더 우수하였다. 또한 우리가 제안한 기법은 새로운 목적이 추가되더라도 기존 목적과의 충돌 없이 새 것을 받아들일 수 있다.

향후 우리는 태스크 간의 통신 시간과 버스 충돌을 고려하여 연구할 계획이다.

참 고 문 헌

[1] Kwok, Y.-K. and Ahmad, I., *"Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors,"* *IEEE Trans. Parallel and Distributed Systems*, Vol. 7, No. 5, pp. 506-521, 1996.

[2] Yalaoui, F. and Chu, C., *"Parallel machine scheduling to minimize total tardiness,"* *International Journal of Production Economics*, Vol. 76, No. 3, pp. 265-279, 2002.

[3] Du, J. and Leung, J. Y. T., *"Minimizing total tardiness on one machine is NP-hard,"* *Mathe-*

matics of Operational Research, Vol. 15, pp. 483-495, 1990.

[4] Lenstra, J. K., Rinnooy Kan, A. H. G., and Brucker, P., *"Complexity of machine scheduling problems," Annals of Discrete Mathematics*, Vol. 1, pp. 343-362, 1977.

[5] Coli, M. and Palazzari, P., *"A new method for optimization of allocation and scheduling in real time applications," Proceedings of the 7th Euromicro Workshop on Real-Time Systems*, pp. 262-269, 1995.

[6] Coello Coello, Carlos A., *An Empirical Study Of Evolutionary Techniques For Multiobjective Optimization In Engineering Design*, Ph. D. Thesis, Tulane University, New Orleans, Louisiana, USA, 1996.

[7] Fonseca, C. and Fleming, P., *"Genetic algorithms for multiobjective optimization: formulation, discussion and generalization,"* Proceedings of the 5th Int'l Conf. on Genetic Algorithms, pp. 416-423, 1993.

[8] J. Horn and N. Nafpliotis, *MultiObjective Optimization Using The Niched Pareto Genetic Algorithm*, Technical Report No. 93005, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 1993.

[9] D. Goldberg and J. Richardson, "Genetic algorithm with sharing for multimodal function optimization," *Genetic Algorithm and Their Applications: Proceedings of the second ICGA*, pp. 41-49, 1987.

[10] Dick, R. and Jha, N., *"MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. 17, No. 10, pp. 920-935, 1998.

[11] Lin, M. and Yang, L. T., *"Hybrid genetic algorithms for scheduling partially ordered tasks in a multi-processor environment," Proceedings of the 6th Int'l Conf. on Real-Time Comp. Systems and App.*, pp. 382-387, 1999.

[12] Mitra, H. and Ramanathan, P., *"A genetic approach for scheduling non-preemptive tasks with precedence and deadline constraints," Proceedings of the 26th Hawaii Int'l Conf. on System Sciences*, pp. 556-564, 1993.

[13] Monnier, Y., Beauvais, J., and Deplanche, A., *"A genetic algorithm for scheduling tasks in a real-time distributed system," Proceedings of the 24th Euromicro Conf.*, pp. 708-714, 1998.

[14] Faucou, S., Deplanche, A.-M., and Beauvais, J.-P., *"Heuristic techniques for allocating and scheduling communicating periodic tasks in distributed real-time systems," Proceedings of 2000 IEEE Int'l Workshop on Factory Communication Systems*, pp. 257-265, 2000.

[15] Hu, T. C., *"Parallel sequencing and assembly line programs," Oper. Res.*, pp. 841-848, 1961.

[16] Liu, C. and Layland, J., *"Scheduling algorithms for multiprogramming in a hard-real-time environment," Journal of the ACM*, Vol. 20, No. 1, 1973.

[17] Leung, J. Y. T., *"A new algorithm for scheduling periodic, real-time tasks," Algorithmica*, Vol. 4, pp. 209-219, 1989.

[18] Hwang, J. J., Chow, Y. C., Anger, F. D., and Lee, C.Y., *"Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," SIAM J. Comput.*, Vol. 18, No. 2, pp. 244-257, 1989.

[19] Cheng, B.-C., Stoyenko, A. D., Marlowe, T. J., and Baruah, S. K., *"LSTF: A new scheduling policy for complex real-time tasks in multiple processor systems," Automatica*, Vol. 33, No. 5, pp. 921-926, 1997.

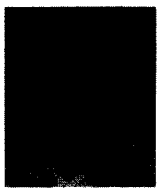
[20] Brest, J. and Zumer, V., *"A performance evaluation of list scheduling heuristics for task graphs without communication costs," Proceedings of 2000 Int'l Workshops on Parallel Processing*, pp. 421-428, 2000.

[21] Altenbernd, P. and Hansson, H., *"The Slack Method: A New Method for Static Allocation of Hard Real-Time Tasks," Real-Time Systems*, Vol. 15, No. 2, pp. 103-130, 1998.

[22] Bui, T. N. and Moon, B. R., *"Genetic algorithm and graph partitioning," IEEE Trans. Computers*, Vol. 45, No. 7, pp. 841-855, 1996.

[23] 문병로, *유전 알고리즘*, 두양사, 2003.

[24] Ge, Y. and Yun, D. Y. Y., *"Simultaneous Compression of Makespan and Number of Processors Using CRP," Proceedings of 1996 Int'l Parallel Processing Symp.*, pp. 332-338, 1996.



오 재 원

1997년 서울대학교 계산통계학과(학사) 1999년 서울대학교 대학원 전산과학과(석사). 2004년 서울대학교 대학원 컴퓨터공학부(박사). 관심분야는 소프트웨어 품질 시험/평가/인증, 실시간 태스크 스케줄링, 분산 객체 기술

김 희 천

1989년 서울대학교 계산통계학과(학사) 1991년 서울대학교 대학원 전산과학과(석사). 1998년 서울대학교 대학원 전산과학과(박사). 1998년~2003년 한세대학교 컴퓨터공학과 조교수. 2004년~현재 한국방송통신대학교 컴퓨터과학과 조교수. 관심분야는 웹 공학, 소프트웨어 테스팅

우 치 수

1972년 서울대학교 응용수학과(공학사) 1972년~1974년 한국과학기술원 연구원 1977년 서울대학교 대학원 전산학(석사) 1982년 서울대학교 대학원 전산학(박사) 1978년 영국 라퍼러대학 연구원. 1975년~1982년 울산대학교 전자계산학과 조교수, 부교수. 1985년~1986년 미국 미시간대학교 Post-doc. 1982년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 소프트웨어 공학, 프로그래밍 언어