센서 노드에서 에너지 효율적인 실시간 및 비실시간 태스크의 혼합 스케줄링 기법

탁성우*

An Energy-Efficient Hybrid Scheduling Technique for Real-time and Non-real-time Tasks in a Sensor Node

Sungwoo Tak*

이 논문은 부산대학교 자유과제 학술연구비(2년)에 의하여 연구되었음

요 약

본 논문에서는 제한된 에너지 자원을 사용하는 센서 노드에서 실시간 및 비실시간 태스크의 실행이 요구되는 경우, 효율적인 전력 소비와 실시간 태스크의 마감시한 보장 및 비실시간 태스크의 향상된 평균 응답 시간을 제공하는 혼합 태스크 스케줄링 기법을 제안하였다. 제안한 혼합 태스크 스케줄링 기법은 기존 **EDF** 기반 **DVS** 스케줄링 기법, **FIFO** 기반 **TinyOS** 스케줄링 기법과 태스크 클러스터링 기반의 비선점형 실시간 스케줄링 기법과 성능을 비교하여 그 우수성을 검증하였다.

ABSTRACT

When both types of periodic and aperiodic tasks are required to run on a sensor node platform with limited energy resources, we propose an energy-efficient hybrid task scheduling technique that guarantees the deadlines of real-time tasks and provides non-real-time tasks with good average response time. The proposed hybrid task scheduling technique achieved better performance than existing EDF-based DVS scheduling techniques available in the literature, the FIFO-based TinyOS scheduling technique, and the task-clustering based non-preemptive real-time scheduling technique.

키워드

무선 센서 노드, 저전력, 실시간 시스템, TinyOS, 스케줄링

Key word

Sensor node, Low-power, Real-time system, TinyOS, Scheduling

* 종신회원 : 부산대학교 정보컴퓨터공학부 (교신저자, swtak@pnu.edu)

접수일자 : 2011. 03. 22
심사완료일자 : 2011. 05. 06

Ⅰ. 서 론

최근 무선 센서 네트워크에서 제공하는 서비스는 환경오염 모니터링, 디지털 홈 제어, 그리고 군사 장비와 같은 실시간성이 요구되는 분야에 활용되고 있다. 이와 같은 고수준 서비스들은 공통적으로 센서 노드에서 수행되는 작업들을 실시간 혹은 비실시간으로 동시에 처리하도록 요구한다.

한편, 배터리 기반의 제한된 전력 공급원을 사용하는 무선 센서 노드 플랫폼에서 프로세서의 기본 구성 요소인 CMOS의 개별 전력 소비와 프로세서의 전체 동작에 따른 전력 소비를 고려할 수 있다 [1][2]. 그리고 시스템 소프트웨어 측면에서는 프로세서에 입력되는 시스템 부하가 시간에 따라 변하기 때문에 프로세서에 입력되는 작업 부하에 따라 프로세서에 공급되는 전압과 동작 주파수를 동적으로 조절하여 센서 노드의 효율적인 전력 소비를 제공하면서 실시간 태스크의 마감시한 보장과 비실시간 태스크의 평균 응답 시간을 향상시킬 수 있어야 한다.

현재 무선 센서 노드의 하드웨어 플랫폼에서 운용되는 대표적인 시스템 소프트웨어 플랫폼은 TinyOS [3], MANTIS [4], CONTIKI [5], 그리고 AvrX [6]가 있다. 이들 플랫폼 중에서 TinyOS는 다른 플랫폼에 비해 꾸준한 연구가 진행되고 있다. TinyOS는 이벤트 기반 비선점형 스케줄링 정책을 사용하며, 처리해야 할 태스크 혹은 이벤트와 같은 작업이 없으면, 센서 노드를 수면 상태로 전환하여 센서 노드의 전력소모를 줄이는 DPM (Dynamic Power Management)기반의 전력 제어 기법을 사용한다. 그러나 DPM 기법은 센서 노드에 입력되는 작업부하 환경에 적합한 프로세서의 공급 전압 및 동작 주파수를 동적으로 제어하지 못한다 [7]. 그리고 TinyOS가 제공하는 비선점형 스케줄링 기법은 즉시 실행이 필요한 태스크가 있음에도 불구하고 자신의 우선순위 보다 낮은 태스크가 획득한 CPU 사용권한을 선점할 수 없기 때문에 실시간 태스크의 속성을 보장하는 것이 어렵다 [8].

이에 본 논문에서는 제한된 에너지 자원을 사용하는 센서 노드 플랫폼에서 실시간 태스크와 비실시간 태스크의 실행이 요구되는 경우, 효율적인 전력 소비를 제공하면서 실시간 태스크의 마감시한 보장과 비실시간 태스크의 향상된 평균 응답 시간을 제공하는 PS-EDF (Postponable execution Start time based Earliest Deadline First) 기반 DVS (Dynamic Voltage Scaling) 혼합 태스크 스케줄링 기법을 제안하였다.

Ⅱ. 관련 연구

이 장에서는 저전력 실시간 센서 노드 플랫폼과 관련된 기존 연구를 분석하였다. 비선점형 태스크 스케줄링 정책을 사용하는 TinyOS에서 실시간성을 제공하는 기법으로는 비선점형 우선순위 기반의 사용자 태스크 스케줄링 기법 [9], 비선점형 EDF 기반의 사용자 태스크 스케줄링 기법 [10], 그리고 선점형 5-단계 우선순위 기반의 사용자 태스크 스케줄링 기법 [11]이 있다. 참조논문 [9]와 [11]은 TinyOS에 선점형 EDF 사용자 태스크 스케줄링 기법을 적용시키기 위하여 태스크간 문맥전환이 가능하도록 수정하였다. 그러나 이러한 기법에서는 기존의 비선점형 TinyOS 구조를 선점형 스케줄링 기반의 센서 노드 플랫폼 구조로 변형시키는 문제가 발생한다. 참조논문 [10]은 비선점형 태스크 스케줄링 정책을 사용하는 TinyOS 환경에서 실행되는 사용자 태스크에게 마감시한 기반의 우선순위를 부여할 수 있도록 하였다. 그러나 비선점형 태스크 스케줄링을 사용하기 때문에 즉시 실행이 필요한 태스크임에도 불구하고 자신의 우선순위보다 낮은 태스크가 획득한 CPU 사용권한을 선점할 수 없다.

일반적으로 실시간 센서 노드 플랫폼을 구현하기 위해서는 선점형 태스크 스케줄링 기법이 요구된다. 선점형 태스크 스케줄링 기법을 적용한 플랫폼으로는 MANTIS, DCOS, 그리고 AvrX 등이 있으며, 이 중에서 MANTIS가 가장 많이 알려져 있다. 여러 센서 노드 플랫폼의 전력 소비를 측정한 기존 연구 [12]에 따르면, 연속적으로 데이터를 이웃 노드에게 전송하는 실험에서는 TinyOS가 MANTIS보다 1.7% 정도의 배터리 향상을 보여 주었다. 참조 논문 [13]에서는 MANTIS에 수면 상태로 전환시킬 수 있는 기능을 추가하여 MANTIS의 전력 소모량과 TinyOS의 전력 소모량이 유사함을 보여 주었다.

센서 노드의 저전력 관리 기법으로 앞서 살펴본 DPM 기법 외에 SVS (Static Voltage Scaling) 기법, 그리고 DVS

기법이 있다. SVS 기법에서는 태스크의 최악 실행시간을 고려하여 한 번 결정된 프로세서의 동작 주파수를 계속해서 사용한다. 따라서 태스크의 실제 실행시간이 최악 실행시간보다 더 짧은 경우가 발생하더라도 동작 주파수를 재조정하지 못하여 센서 노드의 전력소비를 효율적으로 관리할 수 없는 문제점이 있다 [14]. DVS 기법은 프로세서에 입력되는 부하에 따라 프로세서에 공급되는 동작 주파수를 동적으로 조절하여 태스크의 마감시한을 만족하는 범위 내에서 프로세서의 공급 전압을 낮추어 센서 노드 플랫폼의 전력 소비를 감소시켜 효율적인 전력 소비를 제공할 수 있다 [14-15].

참조 논문 [16]에서는 정수선형계획법 (Integer Linear Programming) 기반의 DVS 기법을 사용하였다. 그러나 실시간 태스크의 실제 실행시간이 최악 실행시간보다 짧은 경우가 발생하는 실행환경을 고려하지 않았다. 참조 논문 [17]에서는 DVS 기반의 실시간 태스크 스케줄링 기법인 cc-EDF (cycle-conserving Earliest Deadline First) 기법을 제안하였다. cc-EDF 기법은 태스크의 이른 완료로 발생된 실행 여유 시간을 대기 중인 태스크들에게 균등하게 배분한다. 그리고 태스크의 실시간 속성을 위배하지 않는 범위 내에서 프로세서의 공급 전압을 줄여 전력 소비를 감소시킨다.

Ⅲ. PS-EDF 기반 DVS 태스크 스케줄링 기법

실시간 태스크의 속성을 기술하기 위하여 사용된 수식 기호는 다음과 같다. i번째 태스크 T_i는 요청시간 Tr_i, 실제 실행시간 Te_i, 최악 실행시간 Tc_i, 실행주기 Tp_i, 그리고 절대적 마감시한 Td_i를 속성으로 가지며, T_i (Tr_i, Tc_i, Te_i, Tp_i, Td_i)로 구성된다. 태스크 T_i의 우선순위는 마감시한 Td_i에 의해 결정되며, Td_i가 작을수록 우선순위가 높아진다. 본 논문에서 제안한 PS-EDF 기반 DVS 태스크 스케줄링 기법은 4개의 스케줄링 의사결정 규칙을 사용하고 있다. 그리고 실시간 태스크의 우선순위는 $T_1, T_2, T_3, ..., T_N$과 같이 내림차순으로 서열화된다. 그리고 N은 태스크의 전체 수를 나타내며, 태스크를 구별하기 위하여 변수 i, j, 그리고 k와 같은 아래 첨자가 사용된다. 먼저 스케줄링 의사결정 규칙 1을 살펴보면 다음과 같다.

스케줄링 의사결정 규칙 **1**: 태스크의 실시간 속성인 마감시한을 고려하는 태스크가 N개로 구성된 태스크 집합이 아래의 수식 (1)을 만족하면 제안한 저전력 실시간 센서 노드 플랫폼에서 스케줄링이 가능하다.

$$\sum_{i=1}^{N} \frac{Tc_i}{Td_i} \le 1 \tag{1}$$

태스크 T_i의 최악 실행시간 Tc_i 대신 시점 t 에서 실제 실행시간 Te_i를 고려하는 경우, 수식 (1)은 수식 (2)로 유도된다. 수식 (2)에서 $Re_j(t)$와 $Rd_j(t)$는 t 시점 이전에 실행 중인 태스크 중에서 태스크 T_i의 마감시한보다 짧거나 같은 태스크 T_j의 남은 실행시간 및 마감시한을 나타낸다. $Te_k(t)$와 $Td_k(t)$는 t 시점에서 실행이 요청된 태스크 중에서 태스크 T_i의 마감시한 보다 짧거나 동일한 태스크 T_k의 실제 실행시간과 마감시한을 의미한다. 수식 (2)는 이러한 태스크들의 실행완료에 요구되는 프로세서 이용률 $TU(t)$가 1 이하임을 나타낸다.

$$TU(t) = \sum_{\{Tr_j < t\} \wedge \{Td_j \le Td_i\}}^{i} \frac{Re_j(t)}{Rd_j(t)} + \sum_{\{Tr_k = t\} \wedge \{Td_k \le Td_i\}}^{i} \frac{Te_k(t)}{Td_k(t)} \le 1 \tag{2}$$

시점 t에서 비실시간 태스크가 사용할 수 있는 실시간 태스크의 실행 여유시간을 기술한 스케줄링 의사결정 규칙 2를 살펴보면 다음과 같다.

스케줄링 의사결정 규칙 **2**: 실시간 태스크의 마감시한을 초과하지 않는 범위 내에서 최대 지연 가능한 시간을 의미하는 실행여유 시간에 비실시간 태스크를 스케줄링하여 실시간 태스크의 마감시한을 보장함과 동시에 비실시간 태스크의 응답시간을 향상시킨다. 시점 t에서 태스크 T_i의 실행여유 시간 $EA_i(t)$는 수식 (3)으로 유도되며, 시점 t에서 비실시간 태스크에게 할당 가능한 실행여유 시간은 모든 실시간 태스크의 실행 여유시간 중 가장 최소값이 된다. 수식 (3)에서는 태스크 T_i의 실행시간과 태스크 T_i보다 우선순위가 높으면서 현재 실행 중인 태스크 T_j의 남은 실행시간 $Re_j(t)$, 그리고 태스크 T_i보다 우선순위가 높으면서 실행 대기 중인 태스크 T_k의 실행시간을 고려한다.

$$EA_i(t) = Td_i - \sum_{j=1}^{i} Re_j(t) - \sum_{k=1}^{i} Tc_k \cdot \left\lceil \frac{Tp_i}{Tp_k} \right\rceil \quad (3)$$

스케줄링 의사결정 규칙 3에서는 스케줄링 의사결정 규칙 2에서 기술한 실행여유 시간의 선점 조건을 기술하였다.

스케줄링 의사결정 규칙 **3**: 비실시간 태스크의 응답시간을 향상시키는 규칙은 다음과 같다. 실행대기 중인 비실시간 태스크들 중에서 가장 짧은 대기시간 및 실행시간을 가진 비실시간 태스크를 발생된 실행여유 시간에 먼저 실행시킨다. 시점 t에서 2개의 비실시간 태스크 T_i와 T_j가 있으며, $(Tw_i(t)/Te_i) < (Tw_j(t)/Te_j)$ 조건을 만족한다고 가정한다. $Tw_i(t)$는 시점 t에서 비실시간 태스크의 실행요청 시간에 대한 상대적인 대기시간을 나타내며, $Tw_i(t) = Tr_i/t$ 로 정의된다. 태스크 T_i가 태스크 T_j보다 먼저 실행되는 경우, 전체 응답시간은 $(t + Tc_i)Tw_i(t) + (t + Tc_i + Tc_j)Tw_j(t)$가 된다. 태스크 T_j가 태스크 T_i보다 먼저 실행되는 경우, 전체 응답시간은 $(t + Tc_j)Tw_j(t) + (t + Tc_i + Tc_j)Tw_i(t)$가 된다. 이는 태스크 T_i가 태스크 T_j보다 먼저 실행되는 경우보다 응답시간이 짧다. 따라서 가장 짧은 대기시간 및 실행시간을 가진 비실시간 태스크를 실행여유시간에 먼저 실행시키면, 비실시간 태스크의 응답시간을 향상시킬 수 있다.

한편, 일반적으로 실시간 태스크의 실제 실행시간은 최악 실행시간보다 작은 경우가 대부분이며, 참조 논문 [18]에서는 다양한 실험을 통하여 최악 실행시간과 실제 실행시간의 차이가 최대 10배였음을 보여 주었다. 이러한 실행시간의 차이로 인해 작업 부하량의 변동에 따른 부가적인 실행 여유 시간이 발생한다. DVS 기법을 제공하는 CMOS (Complementary Metal-Oxide Semiconductor) 회로의 동작 주파수를 줄이면 프로세서는 더 낮은 전압에서 동작할 수 있다. CMOS 회로에 공급되는 전압이 V_{dd}이고, 회로의 커패시턴스 부하가 C_{cf}일 때, 전력 소모량 P_d는 $P_d \approx C_{cf} \times V_{dd}^2 \times F$와 같이 유도할 수 있다. F는 동작 주파수를 나타내며, 프로세서의 주파수 속도와 공급 전압은 서로 비례한다. 따라서 주파수 속도가 $1/N$ 로 감소하면 전력 소모량은 $1/N^2$로 감소하게 된다.

스케줄링 의사결정 규칙 **4**: 프로세서의 동작 주파수 및 공급 전압을 동적으로 결정할 수 있는 DVS 기법 및 최악 실행 시간과 실제 실행 간의 차이로 인해 발생되는 부가적인 실행 여유 시간 $DVS\_EA_i(t)$ 는 수식 (4)와 같다.

$$DVS\_EA_i(t) = \quad (4)$$
$$Td_i - \left(\sum_{Td_j \le Td_i}^{N} \frac{Re_j(t)}{f_j} - \sum_{Td_k \le Td_i}^{i} \frac{Te_k}{f_k} \cdot \left\lceil \frac{Tp_i}{Tp_k} \right\rceil \right),$$
$$where\ 0 < f_j \text{ and } f_k \le 1, f_j = \frac{Re_j(t)}{Rd_j(t)}, \text{ and } f_k = \frac{Re_k(t)}{Rd_k(t)}$$
$$\min DVS\_EA_{BCET} = \max(\min\{DVS\_EA_i(t) | i = 1 \le i \le N\}, 0)$$

태스크의 마감시한을 보장하기 위해 동작 주파수 조정 계수 f_j와 f_k는 태스크 T_j와 T_k의 남은 마감시한 대비 남은 실행시간 비율인 Re_j/Rd_j와 Re_k/Rd_k로 설정된다. 시점 t에서 실시간 태스크 T_i의 $DVS_EA_i(t)$를 구하기 위해서는 태스크 T_i의 실행시간과 태스크 T_i보다 우선순위가 높으면서 실행 중인 태스크 T_j의 남은 실행시간 $Re_j(t)$와 태스크 T_i보다 우선순위가 높으면서 실행 대기 중인 태스크 T_k가 T_i의 마감시한 Td_i내에 실행 가능한 시간을 고려한다. 그리고 수식 (4)에서는 시점 t에서 실시간 태스크 T_i가 최적의 실행시간 (BCET: Best-Case Execution Time)으로 실행이 완료되어 발생될 수 있는 최소 실행여유 시간 $\min DVS_EA_{BCET}$를 비실시간 태스크에게 할당한다.

표 1은 스케줄링 의사결정 규칙들을 사용한 PS-EDF 기반 DVS 태스크 스케줄링 기법의 동작 과정을 기술하였다. 표 1에서 PS_i (Postponable execution Start time)는 지연 가능한 실행 시간을 나타내며, PS_i는 Rd_i에서 Re_i만큼 감소한 값이다. PS_i 시간 동안은 태스크 T_i가 자신의 마감시한을 초과하지 않는 범위 내에서 실행을 최대한 지연시킬 수 있다. 만약 태스크 T_i보다 우선순위가 높은 새로운 태스크 T_j가 생성되는 경우, 태스크 T_i의 실행은 PS_i 시간만큼 최대한 지연시킨 후에 실행하여도 태스크 T_j의 마감시한을 만족시킬 수 있다. 그러나 태스크 T_i보다 우선순위가 높은 태스크 T_j의 남은 마감시한인 Rd_j가 태스크 T_i의 PS_i보다 큰 경우에 태스크 T_j가 최악 실행시간으로 수행된다면, 태스크 T_i의 PS_i를 보장하지 못한다. 따라서 Rd_j가 PS_i보다 큰 경우에는 Rd_j 값을 PS_i값으로 변경한 후에 태스크 T_j의 마감시한을 보장할 수 있는 동작 주파수 f_j를 계산한다. 표 1에서 실시간 태스크의 실행요청을 담당하는 Real-time_Task_Release()는 태스크 T_i의 마감시한 Td_i를 기준으로 하여 실시간 태스크 실행대기 관리큐 *Real-time_Task_Ready_Q*를 재정렬한다.

표 1. PS-EDF 기반 DVS 스케줄링 기법
Table. 1 PS-EDF based DVS Scheduling Technique

```
Calculate_PS()
{
 foreach entry task Ti in Real-time_Task_Ready_Q
  Rdi ← min (Rdi, PSi+1); PSi ← Rdi - Rei;
 endforeach
}
Upon Real-time_Task_Release()
{
/* Insert a task Ti at queue Real-time_Task_Ready_Q
   according to a sequence of ordered deadlines, Td1
   ≤ Td2 ≤ Tdi, ... , ≤ Tdn in such a way that the
   resulting sequence of size n+1 is also ordered; */
 Real-time_Task_Ready_Q ← Ti;
 Rdi ← Tdi; Rei ← Tci; PSi ← Rdi - Rei;
 Calculate_PS();
}
Upon Non-real-time_Task_Release()
{
 Non-real-time_Task_Ready_Q ← Tk;
}
Upon Real-time_Task_Completion()
{
 Remove Ti from queue Real-time_Task_Ready_Q;
 if ((Non-real-time_Task_Ready_Q ≠ ∅) ∧ (Tei <
  Tci)) then select a task Tk from the queue
  Non-real-time_Task_Ready_Q by Scheduling
  Decision 3 and run task Tk during
  minDVS_EABCET;
}
Upon Clock_Trigger()
{
 Rei ← Rei - fi for task Ti;
 foreach entry task Tj in Real-time_Task_Ready_Q
  Rdi ← Rdi - 1; PSi ← Rdi - Rei;
 endforeach
}
Select_Frequency()
{
 Microprocessor's operating frequency scaling factor
 fi ← Rei / Rdi, where Rei and Rdi of task Ti;
}
Upon Task_Q_Empty()
{
 if (Real-time_Task_Ready_Q = ∅ ∧
 Non-real-time_Task_Ready_Q = ∅) then
 Microprocessor's state ← Sleep_State;
}
Real-time_Task_Scheduler()
{
 Ti ← Select a task at the head of queue
 Real-time_Task_Ready_Q;
 Select_Frequency(); Run task Ti;
}
```

Non-real-time_Task_Release()는 비실시간 태스크 T_k의 실행요청이 발생하는 경우, 해당 비실시간 태스크를 *Non-real-time_Task_Ready_Q*에 저장하여 관리한다. Calculate_PS()는 태스크 T_i의 마감시한 Rd_i와 태스크 T_i보다 우선순위가 낮은 태스크 T_{i+1}의 PS_{i+1}값 중에서 최소값을 태스크 T_i의 마감시한 Rd_i에 할당한다. Clock_Trigger()는 매 클록이 발생할 때마다 태스크 T_i의 남은 실행시간을 동작 주파수의 조정 계수 f_i만큼 감소시킨다. 그리고 태스크 T_i의 남은 마감시한 Rd_i와 지연 가능한 시간 PS_i를 재계산한다. Select_ Frequency()는 태스크 T_i의 남은 마감시한 Rd_i와 남은 실행시간 Re_i를 기반으로 하여 동작 주파수 조정계수 f_i를 설정한다.

Real-time_Task_Scheduler()에서는 우선순위가 가장 높은 태스크 T_i를 선택하고, 동작 주파수 조정 계수를 설정하여 T_i를 실행시킨다. 그리고 Real-time_Task_Completion()은 실시간 태스크 T_i의 최악 실행시간 Tc_i와 실제 실행시간 Te_i간의 차이로 인해 발생된 실행 여유 시간 $minDVS_EA_{BCET}$에 실행대기 중인 비실시간 태스크를 실행시킨다. 그리고 태스크 T_i가 최악의 실행시간보다 일찍 완료되는 경우, 발생된 실행여유 시간을 실행 대기 중인 태스크 T_{i+1}에 할당하여 프로세서의 동작 주파수 조정계수를 1보다 작은 값으로 설정한다. 이를 통해 센서 노드 플랫폼의 전력소비를 줄일 수 있다. Task_Q_Empty()는 처리해야 할 태스크가 없으면, 프로세서를 수면 상태로 전환시킨다.

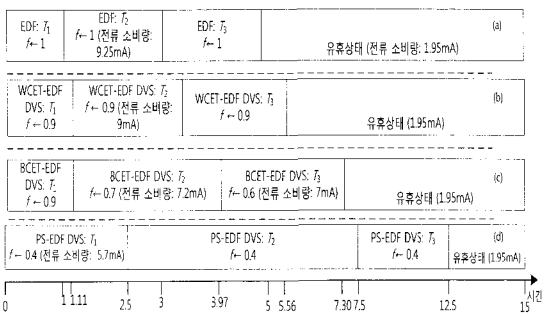

그림 1. 실시간 태스크 스케줄링의 동작 결과
Fig. 1 Operation results of task scheduling techniques

PS-EDF 기반 DVS 스케줄링 기법과 EDF 기법, WCET-EDF 기반 DVS 기법, 그리고 BCET-EDF 기반 DVS 기법의 상세 동작 과정은 그림 1과 같다. 그림 1에

서 사용되는 태스크는 T_1 (Tr_1 = 0, Tc_1 = 2, Te_1 = 1, Tp_1 = 15, Td_1 = 5), T_2 (Tr_2 = 0, Tc_2 = 3, Te_2 = 2, Tp_2 = 15, Td_2 = 10), 그리고 T_3 (Tr_3 = 0, Tc_3 = 3, Te_3 = 2, Tp_3 = 15, Td_3 = 15)으로 구성된다. 동작 실험 환경은 ATmega128 프로세서를 탑재한 Chipcon사의 CC2420DB 센서 노드에서 수행하였다. ATmega128L의 동작 주파수 조정 계수 f가 1일 때 소비되는 단위 전류량은 9.25mA, f가 0.7이면 7.2mA, f가 0.4이면 5.7mA, 그리고 프로세서가 유휴 상태이면 1.95mA가 소요되었다. 나머지 전력 소비량은 그림 1에 기술하였다. 동작 주파수의 조정계수 변화에 따른 전류 소비 증가량은 $\sum_{i=1}^{N} Te_i \cdot f_i^2 \cdot I_{f_i}$와 같이 계산된다. 태스크 T_i의 전류 증가량은 동작 주파수 조정계수 f_i와 태스크의 실제 실행시간 Te_i, 그리고 동작 주파수 조정 계수 f_i에 따른 단위 전류량 I_{f_i}로 구성된다. CMOS기반의 프로세서가 소비하는 전력량은 동작 주파수의 제곱에 비례하며, 시간에 따른 전류 소비량에 비례한다.

그림 1에서 EDF, WCET-EDF 및 BCET-EDF 기반 DVS 태스크 스케줄링 기법 그리고 PS-EDF 기반 DVS 태스크 스케줄링 기법이 소비하는 전류 증가량은 350mA. 332mA, 238mA, 그리고 153mA이다. PS-EDF 기반 DVS 태스크 스케줄링 기법의 전력 소비량이 상대적으로 적음을 확인하였다. 그림 1-(b)에서 보여준 WCET-EDF 기반 DVS 태스크 스케줄링 기법은 SVS 태스크 스케줄링 기법과 유사하게 태스크의 최악 실행시간을 기준으로 하여 설정된다. 태스크 T_1, T_2, T_3의 마감시한 대비 최악 실행시간을 고려하여 설정된 동작 주파수 조정계수는 0.9 (= 2/5 + 3/10 + 3/15)로 설정되어 그림 1-(b)와 같이 동작한다.

그림 1-(c)에서 보여준 BCET-EDF 기반의 DVS 태스크 스케줄링 기법은 참조 논문 [19]에서 제안한 기법과 유사하게 동작한다. 태스크의 초기 실행에서는 최악 실행시간을 고려하여 설정된 동작 주파수로 실행된다. 이후에 태스크가 최악 실행시간보다 이른 시간에 완료되면, 발생된 실행여유 시간을 다음 태스크의 실행에 요구되는 동작 주파수 설정에 이용한다. 발생된 실행여유 시간은 모든 태스크에게 균등하게 분배한다. 시각 t = 0에서 모든 태스크의 최악 실행시간을 고려하여 동작 주파수의 조정계수는 0.9 (= 2/5 + 3/10 + 3/15)로 설정된다. 시각 t = 1.11에서 태스크 T_1은 최악 실행시간보다 일찍 완료되며 실제 실행시간은 1.11이 된다. 동작 주파수의 조정계수를 재계산하기 위하여 차후에 다시 실행이 되는 태스크 T_1의 실제 실행시간을 1로 가정하고, 태스크 T_2의 실행에 사용되는 동작 주파수의 조정계수는 0.7 (= 1/5 + 3/10 + 3/15)로 설정된다. 시각 t = 3.97에서 태스크 T_3의 실행에 사용되는 동작 주파수의 조정계수는 0.6 (= 1/5 + 2/10 + 3/15)으로 설정된다.

그림 1의 (d)에서 보여주는 PS-EDF 기반 DVS 태스크 스케줄링 기법의 동작 과정을 살펴보면 다음과 같다. 시각 t = 0에서 태스크 T_1이 생성되면 태스크 T_1의 남은 마감시한 Rd_1은 5, 남은 실행시간 Re_1은 2, 그리고 실행 여유시간 PS_1은 3 (= 5 - 2)으로 설정된다. 태스크 T_1의 실행에 필요한 프로세서의 동작 주파수 조정계수는 0.4 (= 2/5)로 설정된다. 시각 t = 2.5 에서 태스크 T_1의 실행이 완료되고, 태스크 T_2의 Rd_2는 7.5 (= 10 - 2.5), Re_2는 3, 그리고 PS_2는 4.5 (= 7.5 - 3)로 설정된다. 그리고 동작 주파수의 조정계수는 0.4 (= 3/7.5)로 설정된 후에 태스크 T_2가 실행된다. 시각 t = 7.5에서 태스크 T_2의 실행이 완료되고, 태스크 T_3의 Rd_3는 7.5 (= 15 - 7.5), Re_3는 3, 그리고 PS_3는 4.5 (= 7.5 - 3)로 설정된다. 그리고 동작 주파수의 조정계수는 0.4 (= 3/7.5)로 설정되며 시각 t = 12.5에서 태스크 T_3의 실행이 완료되어, 모든 실시간 태스크의 마감시한을 보장함과 동시에 센서 노드 플랫폼의 전력 소비를 감소시킨다.

Ⅳ. 성능 평가 모델

PS-EDF 기반 DVS 태스크 스케줄링 기법의 성능은 TinyOS 1.x와 2.x 태스크 스케줄링 기법들의 성능과 비교하였다. 그러나 기존 비선점형 TinyOS 1.x와 2.x에서 제공하는 스케줄링 기법의 성능을 분석한 결과, 실시간성 부분에서 좋은 성능을 제공하지 못하였다. 이에 기존 비선점형 TinyOS 구조의 변경 없이 태스크의 실시간 속성을 최대한 만족하면서 비실시간 태스크에게 빠른 응답시간을 제공하는 태스크 클러스터링 기반 스케줄링 기법을 구현하였다.

비선점형 TinyOS 환경에서 실행되는 태스크의 동작과 기존의 일반적인 운영체제에서 실행되는 태스크의 동작 방식은 다음과 같은 두 가지 측면에서 차이점이 있다. 첫째, 비선점형 선입선출 방식의 태스크 스케줄링을 사용하는 TinyOS에서 실행 시간이 긴 태스크에 의해 나

머지 태스크들의 실행이 지연되는 문제를 해결하고자 지연 처리 호출 (DPC: Deferred Procedure Call) 기법을 사용한다. TinyOS의 지연 처리 호출 기법은 실행시간이 긴 태스크를 여러 개의 서브태스크로 나눈 후 각 서브 태스크의 실행 시간을 짧게 하여 프로세서의 선점을 양보할 수 있는 기회를 자주 발생시켜 실행 준비 중인 태스크들의 지연 대기 시간을 단축시키고자 한다. 둘째, 센서 노드의 I/O 작업을 포함하여 TinyOS 운영체제를 구성하고 있는 대부분의 주요 기능들 또한 지연 처리 호출 기법이 적용된 다수의 서브 태스크 단위로 구현된다. 따라서 TinyOS를 탑재한 센서 노드에서 사용자가 요청한 작업은 최소 한 개 이상의 사용자 태스크와 최소 한 개 이상의 TinyOS 플랫폼 태스크로 구성되는 경우가 대부분이다. TinyOS의 스케줄링 단위는 일련의 다수 태스크로 구성된 센서 노드의 작업이 아닌 센서 노드 작업을 구성하는 개별 태스크이다. 그리고 TinyOS 플랫폼 태스크들은 다른 태스크들에 의해 비실시간 및 독립적으로 랜덤하게 호출 및 실행되는 특징이 있다.

TinyOS 2.x [10]의 비선점형 EDF 기반 태스크 스케줄링 기법은 태스크의 마감시한을 매개변수로 입력 받도록 사용자 태스크 생성 인터페이스를 수정하였다. 그러나 TinyOS에서 센서 노드의 I/O 장치들을 제어하는 기존의 TinyOS 플랫폼 태스크들은 이미 매개변수가 없는 void형 기반의 태스크 생성 인터페이스를 사용하여 구현되어 있기 때문에 TinyOS 플랫폼 태스크의 마감시한을 설정할 수가 없다. 따라서 TinyOS 플랫폼 태스크는 호출하는 사용자 태스크의 마감시한을 고려하지 않고 독립적으로 실행되기 때문에 요청한 센서 노드 작업의 실시간 속성이 상실되는 현상이 발생한다. 그림 2는 센서 노드 작업의 실시간 속성이 상실되는 현상이 TinyOS 태스크 스케줄링 기법에 미치는 영향을 보여준다.

그림 2는 3개의 태스크 T_1과 T_2, 그리고 T_3이 TinyOS 1.x의 비선점형 선입선출 기반 스케줄링 기법과 TinyOS 2.x에서 제공하는 비선점형 EDF 기반의 태스크 스케줄링 기법에 따른 동작 결과를 보여준다. 태스크 T_i는 사용자 태스크와 플랫폼 태스크로 표현되는 서브태스크 $SubT_{i,1}, SubT_{i,2}, \ldots, SubT_{i,j}$ 로 구성되며, $T_i = (Tr_i, \{SubT_{i,j}, SubTc_{i,j}\}, Tc_i, Te_i, Tp_i, Td_i)$로 표현된다. 서브태스크 $SubT_{i,j}$는 최악 실행시간 $SubTc_{i,j}$를 가지며, 개별 서브태스크 $SubTc_{i,j}$의 총합은 Tc_i값과 동일하다.

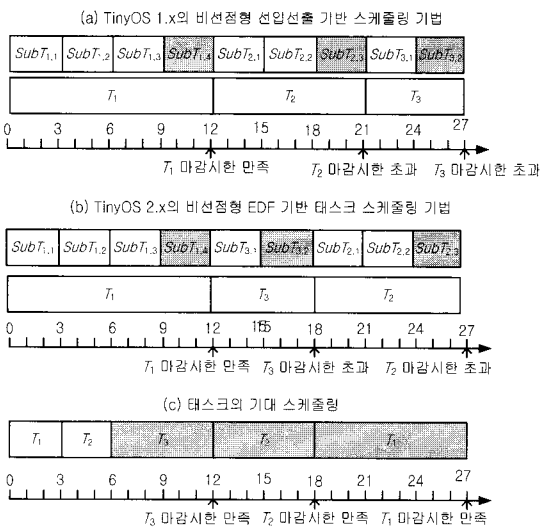

그림 2. 비선점형 TinyOS 태스크 스케줄링 기법에 따른 동작 결과
Fig. 2 Operation results of non-preemptive TinyOS task scheduling techniques

T_1과 T_2, 그리고 T_3은 실행 완료를 위해 각각 4개, 3개, 그리고 2개의 서브태스크 실행이 요구된다. 태스크는 T_1 ($Tr_1 = 0$, $\{SubT_{1,1}, SubTc_{1,1} = 3, SubT_{1,2}, SubTc_{1,2} = 3, SubT_{1,3}, SubTc_{1,3} = 3, SubT_{1,4}, SubTc_{1,4} = 3\}$, $Tc_1 = 12$, $Te_1 = 12$, $Tp_1 = 28$, $Td_1 = 28$)과 T_2 ($Tr_2 = 3$, $\{SubT_{2,1}, SubTc_{2,1} = 3, SubT_{2,2}, SubTc_{2,2} = 3, SubT_{2,3}, SubTc_{2,3} = 3\}$, $Tc_2 = 9$, $Te_2 = 9$, $Tp_2 = 28$, $Td_2 = 19$), 그리고 T_3 ($Tr_3 = 6$, $\{SubT_{3,1}, SubTc_{3,1} = 3, SubT_{3,2}, SubTc_{3,2} = 3\}$, $Tc_3 = 6$, $Te_3 = 6$, $Tp_3 = 28$, $Td_3 = 13$)로 구성된다. 그림 2-(a)에서 보는 바와 같이 TinyOS 1.x에서 사용하고 있는 비선점형 선입선출 방식의 태스크 스케줄링 기법은 태스크의 마감시한에 대한 고려 없이 태스크가 도착한 순서대로 스케줄링한다. 따라서 태스크 T_2와 T_3는 마감시한을 초과하게 된다.

그림 2-(b)의 TinyOS 2.x에서 사용하고 있는 비선점형 EDF 기반 사용자 태스크 스케줄링 기법에서는 사용자 태스크의 마감시한을 기반으로 하여 마감시한이 짧은 사용자 태스크의 서브 태스크를 군집화하여 하나의 태스크 단위로 스케줄링하기 때문에 태스크 T_1의 실행 지연은 태스크 T_2와 T_3에게도 영향을 주게 되어 결국 태스크 T_2와 T_3 모두 마감시한을 초과하게 된다.

이러한 실시간 속성의 상실 현상을 해결하기 위해서는 TinyOS 플랫폼 태스크의 생성 인터페이스를 수

정함과 동시에 선점형 기반의 TinyOS 환경으로 수정해야 한다. 그러나 이러한 수정 요구는 모든 TinyOS 플랫폼 태스크의 재작성이 요구되며, 또한 비선점형 TinyOS 실행환경의 주요 정책 효과 및 목적이 없어지게 된다. 이에 기존 TinyOS 플랫폼 태스크의 구조와 태스크들의 지연 처리 호출하는 방식을 그대로 사용하면서, 사용자 태스크가 호출하는 TinyOS 플랫폼 태스크에게 사용자 태스크의 실시간 속성을 상속하여 태스크 클러스터링으로 결합시키는 태스크 클러스터링 기반의 비선점형 실시간 스케줄링 기법을 구현하였다.

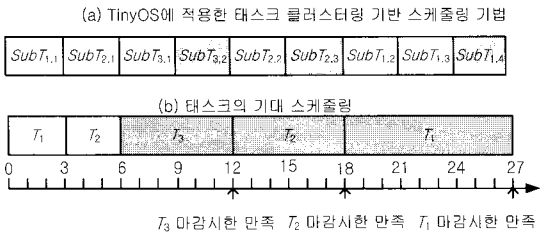

그림 3 태스크 클러스터링 기반 비선점형 실시간 스케줄링 기법에 따른 동작 결과
Fig. 3 Operation results of the task-clustering based non-preemptive real-time scheduling technique

그림 3은 그림 2에서 사용한 태스크 T_1과 T_2, 그리고 T_3을 태스크 클러스터링 기반의 비선점형 실시간 스케줄링 기법으로 동작시킨 결과를 보여준다. 그림 3-(a)에서는 센서 노드 작업의 마감시한을 태스크 내의 서브 태스크에게 상속시켜 마감시한을 보장하도록 하였다. 그림 3-(a)의 동작 결과는 주어진 태스크 집합의 실시간 속성을 만족시키는 그림 3-(b)의 기대 스케줄링 결과와 동일하다. 그림 3에서 보여준 태스크 T_1과 T_2, 그리고 T_3이 스케줄링되는 상세 동작 과정은 그림 4와 같다.

표 2는 태스크 클러스터링 기반 비선점형 실시간 태스크 스케줄링 기법의 알고리즘을 나타낸다. 이 알고리즘에서 사용되는 자료 구조를 살펴보면 다음과 같다. *CL_Head_Task_List*는 실행 대기 중인 태스크를 관리하는 큐이며, 태스크의 마감시한을 기준으로 오름차순으로 태스크를 정렬한다. *CL_Sub_ Task_Ready_Q*는 *CL_Head_Task_List*에서 실행 대기 중인 태스크의 서브태스크를 관리하는 큐 구조이다.

*CL_Sub_Task_Ready_Q*에서 사용하는 *Old_Sub_Task _Ptr* 포인터는 가장 최근에 선점당한 서브태스크의 식별자를 가리킨다. *Current_Sub_ Task _Ptr* 포인터는 현재 실행 중인 서브태스크의 식별자를 가리킨다. 그림 4에서 기술한 *Highest_Priority_ Task_Ptr* 포인터는 현재 시점에서 마감시한이 가장 촉박한 태스크의 식별자를 가리킨다.

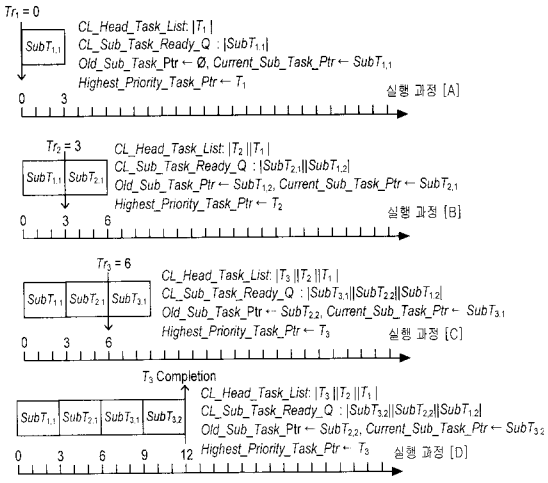

그림 4. 태스크 클러스터링 기반 비선점형 실시간 태스크 스케줄링 기법의 동작 과정
Fig. 4 Operation procedures of the task-clustering based non-preemptive real-time task scheduling technique

그림 4의 실행과정 [A]인 시각 0에서 태스크 T_1의 서브태스크 $SubT_{1,1}$이 호출된다. 이때 서브태스크 $SubT_{1,1}$의 실행 정보인 식별자가 *CL_Sub_Task_ Ready_Q*에 저장되며 (표 2의 CL_Sub_Task_ Release() 동작 과정), *CL_Head_Task_List*에 클러스터링 태스크 T_1의 실행 정보인 마감시한과 식별자가 저장된다 (표 2의 CL_Head_Task_Release() 동작 과정). 그리고 서브태스크 $SubT_{1,1}$이 실행된다 (표 2의 CL_Task_Scheduler() 동작 과정). 그림 4의 실행과정 [B]인 시각 3에서 서브태스크 $SubT_{1,1}$의 실행이 끝나면, 태스크 T_2의 서브태스크 $SubT_{2,1}$와 태스크 T_1의 서브태스크 $SubT_{1,2}$가 실행 가능하다.

표 2 태스크 클러스터링 기반의 비선점형 실시간 태스크 스케줄링 알고리즘
Table. 2 Task-Clustering based Non-preemptive Real-time Task Scheduling Technique

```
Upon CL_Head_Task_Release()
{
 CL_Head_Task_List ← Ti;
 Insert a task Ti at queue CL_Head_Task_List
 according to a sequence of ordered deadlines, Td1
 ≤ Td2 ≤ Td3 , ... , ≤ Tdk in such a way that the
 resulting sequence of size k+1 is also ordered;
 Highest_Priority_Task_Ptr ← Select a task at the head of
 CL_Head_Task_List;
}

Upon CL_Sub_Task_Release()
{
 CL_Sub_Task_Ready_Q ← SubTi,j;
 Insert a subtask SubTi,j at a queue
 CL_Sub_Task_Ready_Q accroding to the following
 criteria. The primary criteria is a sequence of
 ordered deadlines, Td1 ≤ Td2 ≤ Td3 , ... , ≤ Tdk.
 The secondary criteria is a FIFO-based sequence of
 calling subtasks when subtasks belong to the same
 task cluster, where SubTi,j ∈ Ti. In such a way, the
 resulting sequence of size k+1 is also ordered;
}
Upon CL_Head_Task_Completion()
{
 Remove Ti from CL_Head_Task_List;
}
Upon CL_Sub_Task_Completion()
{
 Remove SubTi,j from CL_Sub_Task_Ready_Q;
}
CL_Task_Scheduler()
{
 foreach entry task in CL_Sub_Task_Ready_Q
 if (CL_Head_Task_List = ∅ ∧ CL_Sub_Task_Ready_Q =
    ∅) then Microprocessor's state ← Sleep_State;
 else
    Old_Sub_Task_Ptr ← Current_Sub_Task_Ptr;
    Current_Sub_Task_Ptr ← Select a task at the head
    of CL_Sub_Task_Ready_Q;
    Run (Current_Sub_Task_Ptr);
 endif
 endforeach
}
```

이때 태스크 T_1의 마감시한과 태스크 T_2의 마감시한을 비교한 후에 마감시한이 더 촉박한 서브태스크 $SubT_{2,1}$가 *CL_Sub_Task_Ready_Q*의 맨 선두에 저장된다 (표 2의 CL_Sub_Task_Release() 동작 과정). *Old_Sub_Task_Ptr* 포인터는 선점당한 서브태스크 $SubT_{1,2}$를 가리킨다. 실행과정 [C]와 [D]는 이와 같은 유사한 동작 과정을 보여준다. 그리고 태스크 및 서브 태스크의 실행이 완료되면 *CL_Head_Task_List*와 *CL_Sub_Task_Ready_Q*에서 해당 태스크의 실행 정보를 삭제한다 (표 2의 CL_Head_Task_Completion() 및 CL_Sub_Task_Completion() 동작 과정).

V. 실험 및 성능 분석

제안한 기법의 성능 분석을 위하여 Chipcon사에서 출시한 2대의 CC2420DB 센서 노드를 사용하였다. 성능평가에 사용되는 태스크들은 실행동안 가감 산술연산 과정을 반복적으로 수행한 후 해당 결과를 이웃 센서 노드에게 전송한다. 성능 평가를 위하여 사용한 태스크 집합의 프로세서 이용률은 0.1부터 1까지 0.1 단위로 증가하도록 하였다. 실시간 태스크 T_i는 (Tr_i, Tc_i, Tp_i, Td_i)로 구성된다. 태스크 집합의 처리기 이용률이 각각 0.3, 0.6. 0.9인 경우에 대하여 10ms를 기본 단위로 하여 표현되는 실시간 태스크의 구성은 다음과 같다. 태스크의 프로세서 이용률이 0.3인 경우, 태스크는 T_1 (0, 1, 5, 5), T_2 (0, 1, 20, 20), 그리고 T_3 (0, 3, 60, 60)로 구성된다. 프로세서 이용률이 0.6인 경우, 태스크는 T_1 (0, 1, 5, 5), T_2 (0, 4, 20, 20), T_3 (0, 3, 30, 30), 그리고 T_4 (0, 5, 50, 50)로 구성된다. 프로세서 이용률이 0.9인 경우, 태스크는 T_1 (0, 2, 5, 5), T_2 (0, 4, 20, 20), T_3 (0, 3, 30, 30), T_4 (0, 4, 40, 40), 그리고 T_5 (0, 5, 50, 50)로 구성된다. 한편, 1개의 비실시간 태스크는 실행시간이 10ms와 200ms사이에 랜덤하게 선택되며, 평균 도착율은 초당 0.05 및 0.03 중에서 선택적으로 실행된다.

그림 5는 태스크의 이용률에 따른 마감시한 만족율을 보여준다.

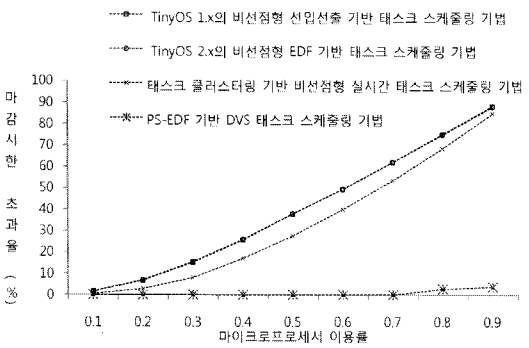

그림 5. 태스크 스케줄링 기법에 따른 마감시한 초과율
Fig. 5 Deadline miss ratio of task scheduling techniques

그림 5의 실험에서 비실시간 태스크의 도착율은 0.05이며, 가로 축은 실시간 태스크의 프로세서 이용률을 나타낸다. 비실시간 태스크의 도착율이 0.05인 경우는 1초에 1/20 비율로 비실시간 태스크가 도착함을 나타낸다. 따라서 비실시간 태스크의 실행으로 프로세서의 이용률이 증가하여 이용률이 1을 초과하는 경우가 발생할 수 있기에 태스크의 마감시한이 초과되는 경우가 발생한다. 그림 5에서 보는 바와 같이, TinyOS 1.x의 비선점형 선입선출 기반 태스크 스케줄링 기법의 실시간 성능을 개선하기 위하여 제안된 TinyOS 2.x의 비선점형 EDF 기반 태스크 스케줄링 기법이 마감시한 보장율을 개선하지 못하고 있다. 이에 대한 이유는 IV 장의 그림 2에 대한 설명에서 기술하였다. 제안한 태스크 클러스터링 기반의 비선점형 태스크 스케줄링 기법을 TinyOS에 탑재하여 성능 분석을 한 결과, 기존 TinyOS에서 사용되는 태스크 스케줄링 기법보다 나은 성능을 보여주었다. 그러나 비선점형 태스크 스케줄링을 기법을 사용하는 TinyOS에서 마감시한 기반의 실시간 태스크 스케줄링을 제공하는 것은 NP-Hard 문제이기에, 마감시한 보장을 요구하는 태스크에게는 선점형 태스크 스케줄링 기법을 적용해야 한다. 따라서 III장에서 제안한 선점형 태스크 스케줄링 기법인 PS-EDF 기반 DVS 태스크 스케줄링 기법은 매우 낮은 태스크의 마감시한 초과율을 보여주었다.

그림 6은 실시간 태스크의 마감시한을 고려한 비실시간 태스크의 평균 응답시간 성능을 보여준다. 성능 분석 대상인 비실시간 태스크 스케줄링 기법은 다음과 같다.

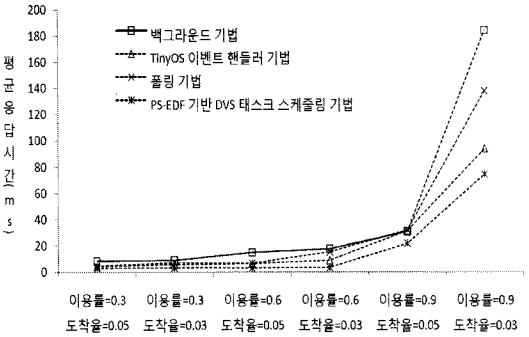

그림 6. 비실시간 태스크의 평균 응답 시간
Fig. 6 Average response time of aperiodic tasks

TinyOS에서는 비실시간 태스크의 응답시간을 향상시키기 위하여 비실시간 작업을 이벤트 핸들러 형태로 구현하여 비실시간 작업에 대한 빠른 응답시간을 제공하도록 하였다. TinyOS에서는 태스크보다 이벤트 핸들러의 실행 우선순위가 높기 때문에 이벤트 핸들러 형태의 구현 방식이 비동기적으로 발생하는 비실시간 작업에게 빠른 응답시간을 제공할 수 있다. 그리고 폴링 기법과 백그라운드 기법을 사용하여 비실시간 태스크를 실행시켰다. 폴링 기법은 주기적으로 실행되는 폴링 태스크가 비실시간 태스크의 실행 요구가 있는지를 주기적으로 폴링을 하여 비실시간 태스크의 실행 요구가 있는 경우, 주기적으로 실행되는 폴링 태스크에 할당된 실행시간 동안 요청된 비실시간 태스크를 실행시킨다. 그림 6의 실험에서 폴링을 주기적으로 실행하는 폴링 태스크의 우선순위를 가장 높게 설정하여 비실시간 태스크의 응답시간을 감소시키고자 하였다. 백그라운드 기법에서는 실행 준비 상태의 실시간 태스크가 없을 경우에 비실시간 태스크를 실행시킨다. 본 논문에서 제안한 PS-EDF 기반 DVS 기법이 폴링 기법과 백그라운드 기법, 그리고 TinyOS의 이벤트 핸들러 기법보다 평균 응답시간이 낮게 나타났다. 또한 실시간 태스크의 이용률이 증가할수록 비실시간 태스크의 응답시간이 증가하는 것을 알 수 있다. 이는 실시간 태스크의 마감시한을 보장하기 위하여 비실시간 태스크의 처리 지연이 발생하기 때문이다.

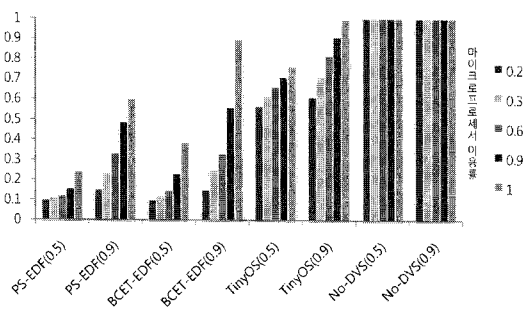

그림 7. 스케줄링 기법에 따른 전력 소모량
Fig. 7 Power consumption of task scheduling techniques

그림 7은 프로세서 이용률과 최악 실행시간 대비 실제 실행시간 비율에 따른 스케줄링 기법의 전력 소모량을 정규화시킨 결과를 보여준다. 개별 태스크의 실제 실행시간에 대한 변화는 가우시안 확률 분포를 따르며, 최악 실행시간 대비 실제 실행시간의 평균 비율은 0.5와 0.9의 값을 가지도록 하였다. 그림 7의 성능 분석에 사용된 기법은 제안한 PS-EDF 기반의 DVS기법, BCET-EDF 기반의 DVS 기법, TinyOS의 비선점형 EDF 기반 태스크 스케줄링 기법, 그리고 DVS를 적용하지 않은 선점형 EDF 태스크 스케줄링 기법인 No-DVS 태스크 스케줄링 기법이다. 실험을 수행한 결과, PS-EDF 기반의 DVS 태스크 스케줄링 기법이 가장 적은 전력 소비량을 보여 주었다. 동작 주파수의 조정 계수를 1로 설정하여 프로세서가 항상 최고 속도로 동작하는 No-DVS 기법에서는 프로세서의 이용률 및 실제 실행시간 비율과 상관없이 전력 소모량의 최대값 1을 유지하고 있다. 처리할 태스크가 없는 경우에 프로세서를 수면 상태로 전환하여 센서 노드의 전력 소모량을 줄이는 TinyOS인 경우, BCET-EDF와 PS-EDF 기반 DVS 태스크 스케줄링 기법보다 전력 소모량이 많았다. TinyOS는 태스크의 실행 여유시간을 활용하지 못하여 실제 실행시간 비율 0.5와 0.9에서 성능 차이가 크게 나지 않는 것을 확인하였다.

VI. 결 론

본 논문에서는 제한된 전원 공급을 사용하는 센서 노드 플랫폼에서 실시간 태스크와 비실시간 태스크의 실행이 요구되는 경우, 실시간 태스크의 마감시한 보장 및 비실시간 태스크의 향상된 평균 응답 시간과 센서 노드의 전력 소비를 줄이는 PS-EDF 기반 DVS 태스크 스케줄링 기법을 제안하였다. 동작 시험을 수행한 결과, PS-EDF 기반의 DVS 스케줄링 기법은 효율적인 전력 소비를 제공함과 동시에 실시간 태스크의 마감시한 보장과 비실시간 태스크의 빠른 응답 시간을 제공하였다.

참고문헌

[1] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," Annual ACM/IEEE Design Automation Conference, pp. 275-280, June 2004.

[2] S. Martin, K. Flautner, T. Mudge, and D. Blaauw, "Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessor under dynamic workloads," IEEE/ACM International Conference on Computer-Aided Design, pp. 721-725, November 2002.

[3] P. Levis and S. Madden, "The emergence of networking abstractions and techniques in TinyOS," USENIX and ACM Symposium on Networked Systems Design and Implementation, pp. 1-14, March 2004.

[4] H. Abrach and S. Bhati. "MANTIS - system support for MultiModAl NeTworks of In-situ Sensors," ACM International Workshop on Wireless Sensor Networks and Applications, pp. 50-59, September, 2003.

[5] A. Dunkels, B. Gronvall, and T. Voigt, "CONTIKI - a lghtweight and flexible operating system for tiny networked sensors," Annual IEEE International Conference on Local Computer Networks, pp. 455-462, November 2004.

[6] P. Ganesan and A.G. Dean, "Enhancing the AvrX kernel with efficient secure communication using software thread integration," Real-Time and Embedded Technology and Applications Symposium, pp. 265-275, May 2004.

[7] V. Raghunathan, C. Schurgers, S. Park, and M.B. Srivastava, "Energy-aware wireless microsensor networks," IEEE Signal Processing Magazine, Vol. 1, No. 2, pp. 40-50, March 2002.

[8] K. Jeffay and C.U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," IEEE Real-Time Systems Symposium, pp. 129-139, December 1991.

[9] V. Subramonian, H-M. Huang, S. Data, and C. Lu, "Priority scheduling in TinyOS - A case study," Technical Report WUCSE-2003-74, Washington University - St. Louis, December 2002.

[10] P. Levis and C. Sharp, "Schedulers and tasks," TinyOS 2.x Extension Proposal 106.

[11] C. Duffy, U. Roedig, J. Herbert, and C. Screenan, "Adding preemption to TinyOS," Workshop on Embedded Network Sensors, pp. 88-92, June 2007.

[12] M. Healy, T. Newe, and E. Lewis, "Power management in operating systems for wireless sensor nodes," IEEE Sensor Applications Symposium, pp. 1-6, February 2007.

[13] C. Duffy, U. Roedig, J. Herbert, and C. Sreenan, "Improving the energy efficiency of the MANTIS kernel," European Conference on Wireless Sensor Networks, pp. 261-276, January 2007.

[14] H. Aydin, R. G. Melhem, D. Mosse, and P. Mejia-Alvarez, "Power aware scheduling for periodic real-time tasks," IEEE Transaction on Computers, Vol. 53, No. 5, pp. 584 - 600, May 2004.

[15] M. Qiu, L. Yang, Z. Shao, and E. Sha, "Dynamic and leakage energy minimization with soft real-time loop scheduling and voltage assignment," IEEE Trans. on VLSI Systems, Vol. 18, No. 3, March 2010.

[16] Z. Cao, B. Foo, L. He, and M. Schaar, "Optimality and improvement of dynamic voltage scaling algorithms for multimedia applications," IEEE Transaction on Circuits and Systems, Vol 57, No. 3, pp. 681-690, March 2010.

[17] P. Pillai and K.G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," ACM symposium on Operating Systems Principles, pp. 89-102, October 2001.

[18] R. Ernst and W. Ye, "Embedded program timing analysis based on path clustering and architecture classification," IEEE/ACM International Conference on Computer-Aided Design, pp. 598-694, 1997.

저자소개

탁성우(Sungwoo Tak)

1995년 2월 부산대학교 컴퓨터공학과 학사
1997년 2월 부산대학교 컴퓨터공학과 석사
2003년 2월 미국미주리주립대학교 Computer Science 박사
2004년~현재 부산대학교 정보컴퓨터공학부 부교수
※관심분야 : 유무선 네트워크