Journal of The Korea Society of Computer and Information
Vol. 19, No. 5, May 2014
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2014.19.5.001

멀티코어 프로세서상의 실시간 태스크들을 위한 중복 실행에 기반한 저전력 결함포용 스케줄링

이 관 우*

Energy-Efficient Fault-Tolerant Scheduling based on Duplicated Executions for Real-Time Tasks on Multicore Processors

Kwan-woo Lee*

요 약

제시된 기법은 실시간 태스크들의 데드라인들을 만족하고 또한 기본-백업 태스크 모델을 사용하여 영구 결함을 포용하면서 멀티코어 프로세서의 에너지 소모량을 최소화하도록 태스크들을 스케줄링한다. 기존의 방법들이 기본 태스크와 백업 태스크의 중복 수행 시간을 최소화하도록 태스크들을 스케줄링했지만, 제시된 기법에서는 코어 속도를 최대한 줄이기 위해서 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화하여 에너지 소모량을 감소시켰다. 제시된 기법이 에너지 소모량을 최소화시킴을 수학적으로 분석하였고, 또한 성능평가 실험을 통해서 제시된 기법이 기존 방법의 에너지 소모량을 최대 77%까지 감소시킴을 보였다.

▸ Keywords : 저전력 스케줄링, 실시간 태스크, 결함 포용, 멀티코어 프로세서

Abstract

The proposed scheme schedules given real-time tasks so that energy consumption of multicore processors would be minimized while meeting tasks' deadline and tolerating a permanent fault based on the primary-backup task model. Whereas the previous methods minimize the overlapped time of a primary task and its backup task, the proposed scheme maximizes the overlapped time so as to decrease the core speed as much as possible. It is analytically verified that the proposed scheme minimizes the energy consumption. Also, the proposed scheme saves up to 77% energy

•제1저자 : 이관우 •교신저자 : 이관우
•투고일 : 2014. 1. 28, 심사일 : 2014. 2. 25, 게재확정일 : 2014. 3. 22.
* 한성대학교 정보시스템공학과(Dept. of Information System Engineering, Hansung University)
※본 연구는 한성대학교 교내연구비 지원 과제임.

consumption of the previous method through experimental performance evaluation.

▸ Keywords : Energy-efficient scheduling, Real-time task, Fault tolerance, Multicore processor

I. 서 론

결함 포용 기법은 결함이 발생하여도 시스템에 장애가 발생하지 않도록 결함으로 인한 파급 영향을 차단하는 방법으로 오랫동안 많은 연구가 진행되어 왔다. 최근 들어 실시간 태스크들의 데드라인 제약과 무선 컴퓨팅 기기의 한정된 배터리 에너지 자원 제약을 모두 고려한 결함 포용 스케줄링(fault-tolerant scheduling) 기법 연구들[1-8]이 활발히 진행되고 있다.

과거에는 사용하지 않는 프로세싱 코어(processing core)의 전원을 동적으로 소등시키는 DPM(Dynamic Power Management) 기법[9]을 적용하여 에너지 소모량을 줄였다면, 최근에는 프로세싱 코어의 연산 속도를 동적으로 조절하는 DVFS(Dynamic Voltage and Frequency Scaling) 기법[1,2,3]을 적용하여 DPM 기법보다 에너지 소모 감소 효율을 향상시켰다. DVFS 기법이 적용되어 현재 실용화된 CPU로는 Intel Pentium M 765, Intel PXA255, AMD Athlon64 4000+등이 있다[3]. 본 논문에서는 멀티코어 프로세서(multicore processor) 상에서 주어진 실시간 태스크들의 데드라인들을 만족하고 또한 기본-백업 태스크 모델(primary-backup task model)을[4,5] 사용하여 영구 결함을 포용하면서 DVFS 기법을 적용하여 에너지 소모량을 최소화하는 스케줄링 기법을 다룬다.

실시간 태스크들에 대한 저전력 결함 포용 기법은 일시 결함(transient fault)을 포용하는 기법[1-3]과 영구 결함(permanent fault)을 포용하는 기법[4-8]으로 분류된다. 일시 결함은 동일 프로세싱 코어에서 주어진 태스크의 반복 수행이라는 시간 중복 접근(time redundancy approach)으로 해결할 수 있지만, 영구 결함은 다른 프로세싱 코어에서 백업 태스크라는 기본 태스크의 복사본 태스크를 추가로 실행하는 하드웨어 중복 접근(hardware redundancy approach)으로만 해결할 수 있다. 영구 결함을 포용하는 방법은 임시 결함도 같이 포용할 수 있으므로, 본 논문에서 제시된 기법은 백업 태스크를 다른 프로세싱 코어에서 추가로 실행하는 기본-백업 태스크 모델을 적용하여 설계되었다.

기본-백업 태스크 모델에 기반을 둔 저전력 결함 포용에 관련된 기존 스케줄링 방법들[4-8]은, 기본 태스크와 백업 태스크의 중복 수행 시간을 최소화하여 에너지 소모량을 감소시켰다. 기본-백업 태스크 모델에서는 기본 태스크가 성공적으로 완료되면 백업 태스크는 더 이상 실행될 필요가 없으므로, 백업 태스크는 중복 수행 기간 동안만 실행된다. 일반적으로 결함이 발생할 확률은 매우 낮아서 결함이 발생하지 않는 상황에서의 에너지 소모량이 절대적 비중을 차지하므로, 기존 방법들은 백업 태스크에 최대 코어 속도를 적용하여 실행 시간을 최소화하고 기본 태스크는 백업 태스크와 중복 수행이 발생하지 않는 한도에서 가장 낮은 코어 속도를 적용하여 에너지 소모량을 줄였다. 본 논문에서는 기본 태스크와 백업 태스크의 데드라인을 만족하는 한도에서 최대한 낮은 코어 속도를 적용하기 위해서 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화하면 오히려 에너지 소모 감소 비율이 향상될 수 있음을 수학적 분석을 통하여 처음으로 입증하였다. 또한 이러한 수학적 분석 결과를 기반으로 에너지 소모량을 최소화하는 스케줄링 기법을 제시하였다.

제시된 스케줄링 기법은 시스템 부하량이 일정 수준 이하이면 기존의 방법과 동일하게 기본 태스크와 백업 태스크의 중복 수행 시간을 최소화하도록 코어 속도를 적용하고, 시스템 부하량이 일정 수준 이상이 되면 기존의 방법과는 상이하게 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화하여 데드라인을 만족하는 최소 코어 속도를 적용한다. 성능 평가 실험을 통하여, 제시된 기법이 기존 스케줄링 기법의 에너지 소모량을 최대 77%까지 감소됨을 알 수 있었다. 본 연구의 기여점으로 1) 기존의 저전력 결함포용 스케줄링 기법과 비교하여, 제시된 기법은 에너지 소모 비율 개선이라는 장점을 들 수 있다. 2) 스케줄러 구현이 용이하며 기존의 방법에서 요구되는 결함 탐지 절차를 제거했다는 점이다.

본 논문의 나머지 부분은 다음과 같이 구성된다. II장은 관련된 기존 연구들을 조사하여 정리한 후 본 연구의 차별성을 열거한다. III장에서는 제시된 기법이 고려하는 시스템 환경

에 대해서 설명한다. IV장은 제안된 스케줄링 기법의 동작 과정을 상세히 설명하며 에너지 감소 비율 향상을 수학적 분석을 통하여 입증한다. 그리고 V장에서는 제안된 기법과 기존 방법의 에너지 소모 성능을 비교하는 실험을 다룬다. 마지막으로 VI장에서는 본 논문의 내용을 요약하고 정리한다.

II. 관련 연구

영구 결함보다는 일시 결함의 발생 비율이 높기 때문에, 기존 관련 연구들의 대부분은 일시 결함을 포용하면서 프로세서의 전력 소모를 최소화하는 스케줄링 문제를 오랜 기간 집중적으로 다루었다[1-3]. 상대적으로 소외되었던 영구 결함에 대한 저전력 결함 포용 스케줄링 연구는 일부 연구들[4-8]에서만 다루어졌다.

멀티코어 프로세서 상에서 영구 결함을 포용하는 기존 저전력 결함 포용 기법들의 대부분[4,7,8]은, 프로세싱 코어들이 동일 순간에도 서로 다른 코어 속도로 동작할 수 있는 독립형 멀티코어 프로세서에 적합하도록 설계되었다. 그러나 동일 순간에는 코어들의 속도가 일치하는 연관형 멀티코어 프로세서[10,11]가 현재 실용화되어 사용되고 있는 DVFS 기법을 지원하는 멀티코어 프로세서의 대부분을 차지한다. 독립형 멀티코어 프로세서에 적합하도록 설계된 기존 방법은 연관형 멀티코어 프로세서에 적용될 수 없다. 그러나 본 논문에서 제시된 기법은 연관형 멀티코어 프로세서에 적합하도록 설계되었다.

기존 연구 [5]에서도 연관형 멀티코어 프로세서에 적용될 수 있는 저전력 결함 포용 방법 기법을 다루었지만, 주기적(periodic) 실시간 태스크가 아닌 비주기적(aperiodic) 실시간 태스크들에 대한 스케줄링 문제만을 다루었다. 반면 제시된 기법은 주기적 실시간 태스크들에 대한 스케줄링 문제를 다룬다.

기존 연구 [6]의 방법은 제시된 기법이 다루는 문제에 적용할 수 있다. 따라서 본 논문의 성능 평가에서는 제시된 기법과 기존 연구 [6]에서 소개된 방법의 에너지 소모 감소 비율을 비교한다. 기존 연구 [6]의 방법에서는 백업 태스크는 최대 코어 속도를 적용하여 실행 시간을 최소화하고 기본 태스크는 중복 수행이 발생하지 않는 한도에서 가장 낮은 코어 속도를 적용하여 전력 소모량을 극대화하지 못했다.

아래의 표 1은 제시된 기법과 관련된 기존 연구들의 장단점들을 정리하여 보여주고 있다.

표 1. 관련 연구들의 장단점 정리

관련연구	장점	단점
[1], [2], [3]	일시 결함을 포용하면서 전력 소모 줄임	영구 결함을 포용하지 못함
[4], [7], [8]	독립형 멀티코어 프로세서 상에서 영구 결함을 포용하면서 전력 소모 줄임	연관형 멀티코어 프로세서에 적용할 수 없음
[5]	연관형 멀티코어 프로세서 상에서 비주기적 실시간 작업들의 영구 결함을 포용하면서 전력 소모 줄임	주기적 실시간 작업들에게 적용할 수 없음
[6]	연관형 멀티코어 프로세서 상에서 주기적 실시간 작업들의 영구 결함을 포용하면서 전력 소모 줄임	전력 소모를 극대화하지 못함

III. 시스템 환경

멀티코어 프로세서는 DVFS 기법[1,2,3]을 제공하며, 동종의 프로세싱 코어들이 N개가 존재한다. DVFS 기법에서는 코어들의 속도를 언제든지 변경할 수 있고, 코어의 에너지 소모량은 코어 속도의 세제곱에 비례한다. 본 논문에서는 동일 순간에도 각 코어가 다른 속도를 가질 수 있는 독립형 멀티코어 프로세서는 고려하지 않고, 동일 순간에는 코어들의 속도가 같은 연관형 멀티코어 프로세서만을 고려한다. 태스크를 실행하고 있지 않는 코어는 휴면 상태(idle status)가 되므로 동일하게 적용되는 코어 속도와는 무관하게 에너지 소모량은 없다[10,11]. 최대 코어 속도를 $S_{\max}$으로 표기하고, 표현의 간결성을 위해 $S_{\max} = 1.0$이 되도록 최대 코어 속도를 정규화한다. 그러면 최대 속도보다 감속된 코어 속도 S는 $0 \leq S < S_{\max}$의 관계식을 만족한다.

영구 결함은 일시 결함(transient fault)를 포함하므로, 본 논문에서는 영구 결함(permanent fault)을 고려한다. 각 결함은 서로 독립적으로 발생하며 단일 프로세싱 코어의 장애(failure)만을 유발시킨다.

서로 독립적으로 수행되는 M개의 주기적 실시간 태스크들의 주어지고, 이들 M개의 태스크들을 $T_1, \ldots, T_M$으로 표기한다. 영구 결함을 포용하는 기본-백업 태스크 모델에서는, 주어진 주기적 실시간 태스크 T_m을 기본 태스크(primary task)으로 명하고 기본 태스크의 실행에 장애가 발생할 경우를 대비하여 다른 프로세싱 코어에서 대신 실행을 수행하도록 준비된 태스크를 백업 태스크(backup task)으로 명한다. 기

본 태스크 T_m에 대한 백업 태스크는 B_m으로 표기한다. 주기적 실시간 태스크들의 도착 주기가 실행을 완료해야 하는 데드라인이 되며, T_m의 데드라인을 D_m로 표기한다. 그리고 최대 코어 속도 $S_{\max}$를 적용할 때 T_m의 실행 시간을 W_m로 표시한다. $S_{\max}$보다 낮은 코어 속도 S로 T_m과 B_m을 실행하면, T_m과 B_m의 실행 시간은 W_m/S가 된다. B_m은 동일한 D_m과 W_m의 값을 가진다. 데드라인 D_m 이내에 T_m과 B_m의 실행을 완료할 수 있어야만, T_m의 실행 완료 실패를 B_m의 실행으로 포용할 수 있다.

일반적으로 결함이 발생하는 경우가 극히 드물어서 결함이 발생하는 않은 상황이 절대적 비율을 차지하므로, 본 논문에서는 결함이 발생하지 않은 상황에서 실시간 태스크의 데드라인과 결함 포용 조건을 만족하면서 프로세서의 에너지 소모량을 최소화하는 문제를 다룬다. 태스크를 실행하기 위해서 단위 시간당 프로세서가 소모하는 전력량(power consumption amount)에 태스크의 실행 시간을 곱한 값이 에너지 소모량(energy consumption amount)이 된다. 즉 에너지 소모량은 태스크를 실행하는 동안에 단위 시간당 프로세서가 소모하는 전력량 값과 태스크 실행 시간 값의 영향을 동시에 받는다.

IV. 제안된 스케줄링 기법

1. 단일 기본 태스크와 백업 태스크의 스케줄링

기존의 방법에서는 기본 태스크와 백업 태스크의 중복 수행 시간을 최소화하고, 이후에 기본 태스크에 적용되는 코어 속도를 최대한 낮춘다. 반면 제시된 기법은 특정 조건을 만족하면 기본 태스크와 백업 태스크의 중복 수행 시간은 고려하지 않고, 데드라인을 만족하는 한도에서 코어 속도를 최소화하여 기본 태스크와 백업 태스크에 적용한다. 데드라인을 만족하는 최소 코어 속도를 적용하면 기본 태스크와 백업 태스크의 중복 수행 시간은 최대화된다.

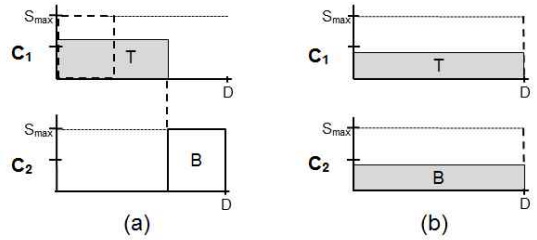

그림 1. $U_m \leq 0.5$ 경우의 스케줄링 비교
Fig. 1. Scheduling comparison when $U_m \leq 0.5$

그림 1(a)는 $U_m \leq 0.5$일 때, 기존의 방법에서 백업 태스크는 최대 코어 속도를 적용하여 실행 시간을 최소화하고 기본 태스크는 중복 수행이 발생하지 않는 한도에서 가장 낮은 코어 속도를 적용하는 경우를 보여주고 있다. 그림 1(a)에서 상단의 점선 사각형은 기본 태스크에 최대 코어 속도를 적용할 때의 시행 시간을 나타낸다. 그림 1(b)는 데드라인을 만족하는 한도에서 최소 코어 속도를 적용하여 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화한 경우를 보여주고 있다. 주어진 태스크의 데드라인 대비 실행 시간 비율에 따라 그림 1(a)의 경우가 그림 1(b)의 경우보다 에너지 소모량이 많을 수도 있고 오히려 적을 수도 있다. 본 논문에서는 에너지 소모량을 최소화하기 위해서 그림 1(b)의 경우가 그림 1(a)의 경우보다 에너지 소모량이 적은 환경 조건을 찾는다.

T_m의 데드라인 대비 실행 시간 비율을 '태스크 이용률(task utilization)'이라고 명하고 U_m으로 표기한다. 즉, $U_m = W_m/D_m$이다. B_m은 T_m과 동일한 U_m 값을 가진다. 제시된 기법은 $U_m \geq (1-\frac{1}{2^{1/2}})$이면 데드라인을 만족하는 한도에서 최저 코어 속도를 적용하여 기본 태스크 T_m과 백업 태스크 B_m의 중복 수행 시간을 최대화하고 방법을 사용하고, $U_m < (1-\frac{1}{2^{1/2}})$이면 T_m과 B_m의 중복 수행 시간을 최소화하는 기존 방법을 사용한다. 그 이유를 다음과 같이 $U_m \leq 0.5$인 환경과 $U_m > 0.5$인 환경으로 분리하여 설명한다.

□ Reason 1. $U_m \leq 0.5$인 환경에서, 그림 1(b)와 같이 T_m과 B_m의 중복 수행 시간을 최대화하는 경우의 에너지 소모량을 먼저 계산한다. 최대 코어 속도가 $S_{\max} = 1.0$이므로, 이 경우의 코어 속도는 $S = U_m$가 된다. 따라서 두 개의 코어가 D_m의 시간동안 소모하는 에너지양은 $2 \cdot S^3 \cdot D_m = 2 \cdot (U_m)^3 \cdot D_m$이다. 다음으로 그림 1(a)와 같이 중복 수행이 발생하지 않는 한도에서 가장 낮은 코어 속도를 T_m에 적용하는 경우의 에너지 소모량은 다음과 같다. $U_m \leq 0.5$일 때, 기본 태스크 T_m의 코어 속도는

$S=\frac{U_m}{(1-U_m)}$이다. T_m을 수행하는 코어가 $D_m \cdot (1-U_m)$의 시간동안 소모하는 에너지양은 $S^3 \cdot D_m \cdot (1-U_m) = \frac{(U_m)^3}{(1-U_m)^3} \cdot D_m \cdot (1-U_m)$이다. 즉, $2 \cdot (U_m)^3 \cdot D_m < \frac{(U_m)^3}{(1-U_m)^2} \cdot D_m$의 관계식을 만족하는 U_m는 $U_m > (1-\frac{1}{2^{1/2}}) \approx 0.29$이다.

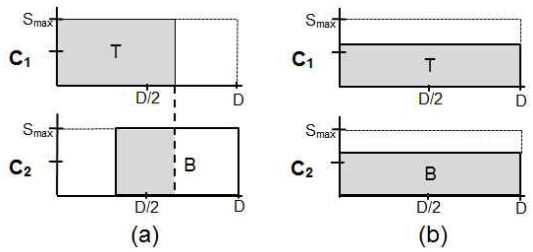

그림 2. $U_m > 0.5$ 경우의 스케줄링 비교
Fig. 2. Scheduling comparison when $U_m > 0.5$

□ **Reason 2.** $U_m > 0.5$인 환경에서, 그림 2(a)에서 보여주듯이 T_m과 B_m의 중복 수행 시간을 최소화하는 기존 방법과 그림 2(b)에서 보여주듯이 기본 태스크 T_m과 백업 태스크 B_m의 중복 수행 시간을 최대화하는 방법의 에너지 소량을 비교하면 다음과 같다. 그림 2(a)에서 보여주는 중복 수행 시간을 최소화하는 경우의 에너지 소모량을 먼저 계산한다. $U_m > 0.5$일 때, 기본 태스크 T_m의 코어 속도는 $S_{\max}=1$이고, $U_m \cdot D_m$의 시간동안 소모하는 에너지양은 $(S_{\max})^3 \cdot U_m \cdot D_m = U_m \cdot D_m$이다. 백업 태스크 B_m의 코어 속도도 $S_{\max}=1$이고, 중복 시간은 $(U_m-\frac{1}{2}) \cdot 2 \cdot D_m$이며 중복 시간동안 소모하는 에너지양은 $(S_{\max})^3 \cdot (2U_m-1) \cdot D_m = (2U_m-1) \cdot D_m$이다. 따라서 T_m와 B_m의 전체 에너지 소모량은 $(3U_m-1) \cdot D_m$이다. 다음으로 그림 2(b)와 T_m과 B_m의 중복 수행 시간을 최대화하는 경우의 에너지 소모량을 계산한다. 코어 속도는 $S=U_m$이므로, 두 개의 코어가 D_m의 시간동안 소모하는 에너지양은 $2 \cdot S^3 \cdot D_m = 2 \cdot (U_m)^3 \cdot D_m$이다.

그림 2(a)의 에너지 소모량 $(3U_m-1) \cdot D_m$과 그림 2(b)의 에너지 소모량 $2 \cdot (U_m)^3 \cdot D_m$을 비교하기 위해서, $0.5 < U_m < 1$ 값에 대해서 $2 \cdot (U_m)^3-(3U_m-1) < 0$임을 보여 $(3U_m-1) > 2 \cdot (U_m)^3$임을 증명한다. 3차 함수 $F(U_m) = 2 \cdot (U_m)^3-(3U_m-1)$을 미분한 $6(U_m)^2-3=0$의 수식을 만족하는 U_m의 값은 $\pm\frac{1}{\sqrt{2}}$이다. 이는 3차 함수 $F(U_m)$는 $-\frac{1}{\sqrt{2}}$까지는 계속 증가하다가 이후에 $+\frac{1}{\sqrt{2}}$까지는 계속 감소하고, 이후에는 다시 계속 증가한다는 것을 의미한다. 따라서 $0.5 < U_m < 1$ 값에 대해서 3차 함수 $F(U_m)$는 계속 증가하는 것을 의미하며, $F(0.5) < 0$이고 $F(1)=0$이므로 $0.5 < U_m < 1$ 값에 대해서 $F(U) < 0$ 이다. 따라서 $\frac{1}{2} < U_m < 1$ 값에 대해서 $2 \cdot (U_m)^3-(3U_m-1) < 0$ 이므로 $(3U_m-1) > 2 \cdot (U_m)^3$이다. 즉, $0.5 < U_m < 1.0$인 환경에서는 데드라인을 만족하는 한도에서 최소 코어 속도를 적용하여 T_m과 B_m의 중복 수행 시간을 최대화하는 방법이 T_m과 B_m의 중복 수행 시간을 최소화하는 기존 방법보다 항상 에너지 소모량이 적다.

$U_m \le 0.5$인 환경에 대한 분석 결과와 $U_m > 0.5$인 환경에 대한 분석 결과를 종합하면, $U_m \ge (1-\frac{1}{2^{1/2}}) \approx 0.29$이면 데드라인을 만족하는 한도에서 최저 코어 속도를 적용하여 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화하는 방법이 기존 방법과 비교하여 에너지 소모량을 감소시키거나 에너지 소모량이 동일하다.

최대한 낮은 코어 속도 $S_{low}=U_m$을 적용하지 않고 적당히 낮은 코어 속도 S_{mid} $(S_{low} < S_{mid} < S_{\max})$을 적용하여 T_m과 B_m의 중복 시간 길이를 변경하는 여러 가지 방법들에 대해서도 그림 2의 $U_m > 0.5$ 환경에 대한 수학적 분석을 유사하게 적용할 수 있다. 본 논문에서 사용하는 최대 코어 속도 $S_{\max}=1.0$는 상대적 속도이므로 S_{mid}가 상대적 최대 코어 속도가 되도록 $S_{mid}=1.0$ 환경으로 변경시킬 수 있고, $S_{mid}=1.0$ 변경 이후의 태스크 이용률을 U_m^t $(> U_m)$이라고 하면 $U_m^t > 0.5$인 경우로 전환된다. 전환된 경우에 대

해서 그림 2의 $U_m > 0.5$ 환경에 대한 수학적 분석을 동일하게 적용하면, 최대한 낮은 코어 속도 $S_{low} = U_m^t$ 을 적용하여 중복 수행 시간을 최대화하는 것이 S_{mid}를 적용하여 감소된 중복 수행 시간의 경우보다 에너지 소모량이 적다. 즉, 기본 태스크와 백업 태스크의 중복 수행이 존재한다면, 기본 태스크의 코어 속도와 백업 태스크의 코어 속도가 동일한 연관형 멀티코어 프로세서 환경에서는 데드라인을 만족하는 한도에서 최저 코어 속도를 적용하여 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화하는 것이 항상 에너지 소모량을 최소화한다. 그리고 기본 태스크와 백업 태스크의 중복 수행이 존재하지 않는다면, 백업 태스크는 최대 코어 속도를 적용하여 실행 시간을 최소화하고 기본 태스크는 중복 수행이 발생하지 않는 한도에서 가장 낮은 코어 속도를 적용하는 것이 항상 에너지 소모량을 최소화한다. 결론적으로 연관형 멀티코어 프로세서 환경에서는, $U_m \geq (1-\frac{1}{2^{1/2}})$이면 데드라인을 만족하는 한도에서 최저 코어 속도를 적용하여 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화하고, $U_m < (1-\frac{1}{2^{1/2}})$이면 백업 태스크는 최대 코어 속도를 적용하여 실행 시간을 최소화하고 기본 태스크는 중복 수행이 발생하지 않는 한도에서 가장 낮은 코어 속도를 적용하는 것이 항상 에너지 소모량을 최소화한다.

2. 다중 태스크들의 스케줄링

앞의 VI장 1절에서는 단일 기본 태스크와 이에 상응하는 백업 태스크를 두 개의 코어에서 각각 실행할 경우의 에너지 소모량을 최소화하는 방법을 다루었다. 나머지 해결해야 하는 문제는 각 코어에 배치된 태스크가 다수인 경우에 하나의 코어 상에서 다수의 태스크들을 실행시키는 방법과, 주어진 기본 태스크들과 백업 태스크들을 코어들에게 배치시키는 방법을 결정하여야 한다.

다수의 태스크들을 하나의 코어 상에서 실행시키는 방법을 위해, n번째 코어에 배치된 태스크들의 태스크 이용률 합을 '코어 이용률(core utilization)'이라고 명하고 CU_n으로 표기한다. 만약 $T_1, \ldots, T_x$가 n번째 코어에 배치되었다면, $CU_n = \sum_{i=1}^{x} U_i$이다. 제시된 기법에서는 VI장 1절과 유사하게 $CU_n < (1-\frac{1}{2^{1/2}}) \approx 0.29$이면 기존 방법[6]과 동일하게 기본 태스크와 백업 태스크의 중복 수행 시간이 없는 한도에서 기본 태스크를 실행하는 코어 속도를 최소화하고, $CU_n \geq (1-\frac{1}{2^{1/2}})$이면 데드라인을 만족하는 한도에서 최대한 낮은 코어 속도를 적용하여 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화한다.

주어진 기본 태스크들과 백업 태스크들을 코어들에게 배치할 때, 각 코어에는 기본 태스크들만 배치되거나 백업 태스크들만 배치하여 기본 태스크들과 백업 태스크들이 혼합되어 동일한 코어에 배치되는 경우는 발생하지 않도록 한다. 기존 방법에서는 기본 태스크들만 배치된 코어는 EDF(Earliest Deadline First) 규칙[6]에 기반을 두어 선행 데드라인을 가진 태스크를 선순위로 실행 순서를 배정하고 각 태스크에 대해서 최대한 빠른 시기에 수행을 시작한다. 그리고 백업 태스크들만 배치된 코어는 EDL(Earliest Deadline Late) 규칙[6]에 기반을 두어 선행 데드라인을 가진 태스크를 후순위로 실행 순서를 배정하고 각 태스크에 대해서 데드라인을 만족하는 한도에서 최대한 늦은 시기에 수행을 시작한다. 반면 제시된 기법에서는 $CU_n \geq (1-\frac{1}{2^{1/2}})$이면 기본 태스크들과 백업 태스크들에 대해서 동일하게 EDF 규칙에 기반을 두어 각 태스크의 실행 순서를 배정한다.

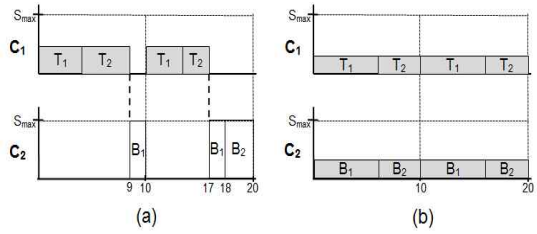

그림 3. 단일 코어에서 다중 태스크들의 스케줄링
Fig. 3. Scheduling of multiple tasks on a core

그림 3은 제시된 기법에서 다중 태스크들을 수행하는 예시를 보여주고 있다. 그림 3의 (a)는 $CU_n < (1-\frac{1}{2^{1/2}})$에 해당하는 경우로, 제시된 기법과 기존 방법의 동작 과정이 동일하다. 기본 태스크와 백업 태스크의 중복 수행은 발생하지 않고, 백업 태스크는 최대 코어 속도 $S_{\max}$를 적용하며 기본 태스크는 중복 수행이 발생하지 않는 한도에서 가장 낮은 코어 속도를 적용한다. 기본 태스크는 EDF 규칙에 기반을 두어 실행 순서를 배정하고 최대한 빠른 시기에 실행을 시작하며, 백업 태스크는 EDL 규칙에 기반을 두어 실행 순서를 배정하고 데드라인을 만족하는 한도에서 최대한 느린 시기에 실

행을 시작한다. 그림 3의 (b)는 $CU_n \geq (1-\frac{1}{2^{1/2}})$에 해당하는 경우로, 동작 과정이 기존 방법과 달라진다. 배치된 태스크들의 데드라인을 만족하는 한도에서 최대한 낮은 코어 속도를 적용하며 EDF 규칙에 기반을 두어 선행 데드라인을 가진 태스크를 선순위로 실행 순서를 배정한다.

마지막으로 주어진 M개의 기본 태스크와 M개의 백업 태스크를 N개의 코어들에게 배치하는 방법은 다음과 같다. 먼저 M개의 기본 태스크들을 $N/2$개의 코어들에게 배치하고, M개의 백업 태스크들을 기본 태스크와 동일한 과정으로 나머지 $N/2$개의 코어들에게 배치한다. M개의 기본 태스크들을 $N/2$개의 코어들에게 배치할 때, 배치 이후의 코어 이용률 값들이 최대한 균등화되도록 배치하는 것이 에너지 소모량을 감소시킨다[12]. 그러나 코어 이용률 값들을 최대한 균등화시키는 배치 결정을 찾는 문제는 NP-hard의 계산 복잡도를 가지므로[12], 제시된 기법에서는 휴리스틱(heuristic) 배치 방법들 중에서 균등화에 가장 우수한 성능을 보이는 WFD(Worst-Fit Decreasing) 휴리스틱 방법[12]을 사용한다. WFD 휴리스틱 방법은 주어진 태스크들을 태스크 이용률 값에 기반을 두어 내림차순으로 정렬하고, 주어진 코어들의 현재 코어 이용률 값을 비교하여 가장 낮은 코어 이용률 값을 가진 코어에게 각 태스크를 순차적으로 배치한다.

본 논문에서 고려하는 연관형 멀티코어 프로세서 환경에서는, 수행해야 할 태스크를 가진 코어들은 동일 순간에는 같은 코어 속도로 동작한다. 따라서 배치가 완료된 이후에는 먼저 각 코어에 적용될 코어 속도를 계산하고, 계산된 코어 속도들 중에서 가장 높은 코어 속도를 선택하여 모든 코어들에게 적용한다. 그렇지 않으면 일부 태스크들이 데드라인 이내에 수행을 완료하지 못하는 문제가 발생한다.

제시된 스케줄링 기법은 에너지 소모량 감소 이외에도 스케줄러 구현 간결성 및 기존의 방법에서 요구되던 결함탐지 절차 제거라는 장점을 추가로 내포하고 있다. 그림 3(b)의 방법에서는 데드라인을 만족하는 최저 코어 속도는 코어 이용률과 동일하므로 코어 속도를 결정하는 계산 과정이 매우 단순하여 스케줄링 기능을 전담하는 태스크 스케줄러(task scheduler) 구현이 용이하다. 반면 그림 3(a)의 방법에서는 기본 태스크들에 대해서 백업 태스크와 중복이 발생하지 않는 한도에서 최저 코어 속도를 계산해야 하는데, 코어 이용율 값이 커질수록 이 과정은 복잡해지고 특히 코어 이용율 값이 0.5 이상 되면 이 계산 과정은 매우 복잡하여 태스크 스케줄러 구현 난이도가 증가하게 된다. 스케줄러 구현 난이도가 증가하게 되면 스케줄러 구현 비용이 증가하고 또한 스케줄러가 예상하지 못한 불완전한 동작을 수행할 가능성이 커진다. 그리고 그림 3(a)의 방법에서는 기본 태스크 실행이 정상적으로 완료되면 백업 태스크 실행을 취소하고 기본 태스크 실행이 실패하면 백업 태스크 실행을 호출하는 기본 태스크 결함 탐지 과정[4,8]이 추가로 필요한 반면, 그림 3(b)의 방법에서는 기본 태스크 실행 완료 여부와 상관없이 백업 태스크를 항상 실행하므로 기본 태스크에 대한 결함 탐지 절차를 제거할 수 있다. 기존의 기본-백업 태스크 모델에 기반을 둔 연구들[4-8]에서는 기본 태스크 결함 탐지 절차에 소요되는 시간을 기본 태스크의 계산시간에 포함된다고 가정하였다.

V. 성능 평가

성능 평가를 위하여 제시된 기법과 기존 방법[6]의 에너지 소모량을 비교하였다. 제시된 기법과 기존 방법은 기본 태스크와 백업 태스크를 실행한 코어를 WDF 휴리스틱 기법을 적용하여 배치하는 과정이 동일하고, 또한 코어 이용률이 0.29 미만이면 각 코어가 배치된 태스크들을 수행하는 과정도 동일하다. 다른 점은, 코어 이용률이 0.29 이상이면 제시된 기법은 VI장 2절에서 설명하였듯이 배치된 태스크들의 데드라인을 만족하는 한도에서 최대한 낮은 코어 속도를 적용하여 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화한다. 반면, 기존 방법은 코어 이용률이 0.29 이상이어도 기본 태스크와 백업 태스크의 중복 수행 시간을 최소화하기 위해서, 백업 태스크는 최대 코어 속도를 적용하고 기본 태스크는 중복 수행이 발생하지 않는 한도에서 가장 낮은 코어 속도를 적용한다. 제시된 기법과 기본 방법 모두에서 배치가 완료된 이후에 먼저 각 코어에 적용될 코어 속도를 계산하고, 계산된 코어 속도들 중에서 가장 높은 코어 속도를 선택하여 수행해야 할 태스크를 가진 코어들에게 적용하였다.

성능 실험에서 사용 가능한 프로세싱 코어들은 8개로 주어지고, 주기적 실시간 태스크들은 16개로 고정하여 16개의 기본 태스크들과 16개의 백업 태스크들이 주어진다. 평가 지표로는 '기존 방법의 에너지 소모량' 대비 '제시된 방법의 에너지 소모량' 비율을 '상대적 에너지 소모율'이라고 정의하여 사용한다. 상대적 에너지 소모량을 평가 지표로 사용하면 코어 속도의 실제 값은 성능에 영향을 미치지 않으므로, 최대 코어 속도 $S_{\max}$는 1.0의 값을 가지도록 임의로 설정하였다. 최대 속도보다 감속된 코어 속도 S는 $0 \leq S \leq S_{\max}$ 사이에서 임의의 모든 값들을 가질 수 있고, 코어 속도 S의 단위 시간당 에너지 소모량은 코어 속도의 세제곱(S^3)에 비례한다. 그

리고 태스크들의 데드라인과 계산량은 다음과 같이 인위적으로 생성하였다. 각 태스크의 데드라인 값은 0.01초와 1초 사이에서 균등분포를 따르도록 선택하였고, 최대 코어 속도 $S_{\max}=1.0$를 적용할 때의 실행 시간 W_m은 0.001초와 데드라인 사이에서 정규 분포 또는 지수 분포를 따르도록 선택하였다.

동일한 실험 조건에 대해서 10만개의 태스크 집합들을 생성하여 성능을 측정하고, 측정된 값들의 평균을 표시하였다. Unix 운영체제 환경에서 C 프로그램을 작성하여 성능 실험을 실시하였고, SimPack 툴[2][13]이 제공하는 라이브러리 함수를 사용하여 태스크들의 데드라인과 계산량 값들의 확률적 분포를 조절하였다.

그림 4는 태스크들의 실행 시간을 정규분포를 사용하여 선택한 경우에 제시된 방법의 상대적 에너지 소모율을 보여주고 있다. x축의 '시스템 부하'는 코어 개수 대비 태스크 이용률 전체 합의 비율($\frac{\sum_{m=1}^{M} 2 \cdot U_m}{N}$)을 나타낸다. 다시 말해서 시스템 부하는 태스크 배치가 완료된 이후의 전체 코어들의 평균 코어 이용률 값($\frac{\sum_{n=1}^{N} CU_n}{N}$)을 백분율로 표시한 것이다. 'No-OH'는 기존 방법에서 기본 태스크에 대한 결함탐지 절차 부하가 전혀 없다고 가정한 경우에 대한 성능이고, '5%-OH'는 결함탐지 절차 부하량을 기본 태스크 계산량의 5%로 적용하는 경우에 대한 성능이며, '10%-OH'는 결함탐지 절차 부하량을 기본 태스크 계산량의 10%로 적용하는 경우에 대한 성능이다.

시스템 부하가 25%이하일 때는, 제시된 방법과 기존 방법의 동작과정이 동일하여 에너지 소모량이 같다. 시스템 부하가 30% 이상이 되면, 제시된 기법의 상대적 에너지 소모율은 점진적으로 줄어들다가 시스템 부하의 값이 50% 또는 55% 이후에 상대적 에너지 소모비율이 다시 점진적으로 증가한다. 결함탐지 절차 부하가 없는 경우, 시스템 부하가 55%일 때 상대적 에너지 소모율이 약 30%이므로 제시된 기법이 기존 방법의 에너지 소모량을 약 70% 감소시켰다. 결함탐지 절차 부하가 5%인 경우에 대해서는 시스템 부하가 50%일 때 상대적 에너지 소모율이 약 25%이고, 결함탐지 절차 부하가 10%인 경우에 대해서는 시스템 부하가 50%일 때 상대적 에너지 소모율이 약 23%이다.

2) 컴퓨터 시뮬레이션에 필요한 큐잉(queuing) 함수들을 C나 C++로 구현해 놓은 라이브러리 패키지.

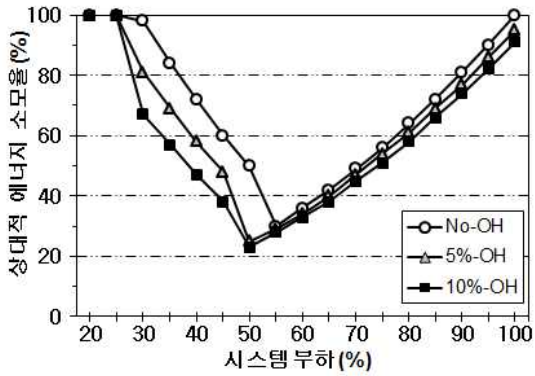

그림 4. 정규분포에 대한 성능 평가
Fig. 4. Evaluation of normal distribution

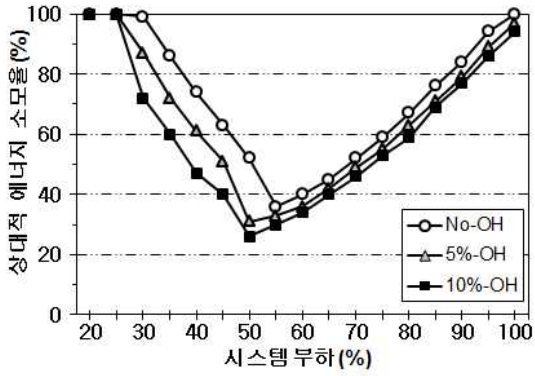

그림 5. 지수 분포에 대한 성능 평가
Fig. 5. Evaluation of exponential distribution

그림 5는 태스크들의 실행 시간을 지수분포를 사용하여 선택한 경우의 상대적 에너지 소모율로, 그림 4와 매우 유사한 성능 분포를 보였지만 미세한 차이를 보인다. 그 이유는 지수분포가 정규분포에 비해서 상대적으로 태스크 이용률 값들의 편차가 크고, 이로 인해서 WDF 휴리스틱에 기반하여 태스크들을 코어들에게 배치한 이후에 코어 이용률 값들의 편차가 상대적으로 커지기 때문이다.

V. 결 론

본 논문에서는 멀티코어 프로세서 상에서 실시간 태스크들의 데드라인을 만족하고 또한 기본-백업 태스크 모델을 사용하여 영구 결함을 포용하면서 에너지 소모량을 최소화하는 스케줄링 기법을 제시하였다. 제시된 기법은 시스템이 특정 조

건을 만족하면 데드라인을 만족하는 한도에서 최대한 낮은 코어 속도를 적용하여 기본 태스크와 백업 태스크의 중복 수행 시간을 최대화함으로써 에너지 소모 감소 비율을 극대화하였다. 수학적 분석을 통하여 제시된 기법이 연관형 멀티코어 프로세서 환경에서 에너지 소모량을 항상 최소화함을 입증하였고, 성능평가 실험을 통해서 제시된 기법이 기존 방법의 에너지 소모량을 최대 77%까지 감소시킴을 확인하였다.

성능 평가의 신뢰도를 높이기 위해서 제시된 스케줄링 기법을 실제 프로세서 환경에서 구현하고자 하였으나, 장비 구축의 높은 난이도로 인해서 실제 프로세서 환경과 흡사한 시뮬레이션에 기반한 성능 평가 실험으로 대체하였다. 향후 연구에서는 동일 순간에 코어들의 다른 속도 사용을 허용하는 독립형 멀티코어 프로세서 환경에서 기본 태스크와 백업 태스크의 에너지 소모량을 최소화하도록 기본 태스크와 백업 태스크의 중복 수행 길이를 결정하는 분석 연구도 수행할 예정이다.

참고문헌

〔1〕 R. Melhem, D. Mosse and E. Elnozahy, "The interplay of power management and fault recovery in real-time systems," IEEE Trans. Computers, Vol. 53, No. 2, pp. 217-231, 2004.

〔2〕 T. Wei, P. Mishra, K. Wu and H. Liang, "Fixed-priority allocation and scheduling for energy-efficient fault tolerance in hard real-time multiprocessor systems," IEEE Trans. Parallel Distrib. Syst., Vol. 19, No. 1, pp. 1511-1525, 2008.

〔3〕 Y. Liu, H. Liang and K. Wu, "Scheduling for energy efficiency and fault tolerance in hard real-time systems," Design, Automation and Test in Europe Conference and Exhibition, pp. 1444-1449, 2010.

〔4〕 R. Al-Omari, A.K. Somani and G. Manimaran, "An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems," Journal of Parallel and Distributed Computing, Vol. 65, No. 5, pp. 595-608, 2005.

〔5〕 M.K. Tavana, M. Salehi and A. Ejlali, "Feedback-based energy management in a standby-sparing scheme for hard real-time systems," IEEE Real-Time Systems Symp., pp. 349-356, 2011.

〔6〕 M.A. Haque, H. Aydin and D. Zhu, "Energy-aware standby-sparing technique for periodic real-time applications," IEEE Int'l Conf. Computer Design, pp. 190-197, 2011.

〔7〕 Y. Guo, D. Zhu and H. Aydin, "Efficient power management schemes for dual-processor fault-tolerant systems," Int'l Workshop Highly-Reliable Power-Efficient Embedded Designs, pp. 23-27, 2013.

〔8〕 Y. Guo, D. Zhu, H. Aydin and L.T. Yang, "Energy-efficient scheduling of primary/backup tasks in multiprocessor real-time systems," Tech. Report CS-TR-2013-016, Available at http://venom.cs.utsa.edu/dmz/techrep/2013/CS-TR-2013-016.pdf, Univ. of Texas at San Antonio, 2013.

〔9〕 L. Benini, A. Bogliolo and G. Micheli, "A survey of design techniques for system-level dynamic power management," IEEE Trans. VLSI Syst., Vol. 8, No. 3, pp. 299-316, 2000.

〔10〕 H. Pack, J. Yeo and W. Lee, "Energy-efficient multi-core scheduling for real-time video processing," Journal of the Korea Society of Computer and Information, Vol. 16, No. 6, pp. 11-20, 2011.

〔11〕 W.Y. Lee, "Power-efficient scheduling of periodic real-time tasks on lightly loaded multicore processors," Journal of the Korea Society of Computer and Information, Vol. 17, No. 8, pp. 11-19, 2012.

〔12〕 H. Aydin and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," Int'l Parallel Distributed Processing Symp., 2003.

〔13〕 P. A. Fishwick, "SimPack: getting started with simulation programming in C and C++," Winter Simulation Conference, pp. 154-162, 1992.

저 자 소 개

이 관 우

1994: POSTECH
컴퓨터공학 공학사.
1996: POSTECH
컴퓨터공학과 공학석사.
2003: POSTECH
컴퓨터공학과 공학박사
현 재: 한성대학교
정보시스템공학과 부교수
관심분야: 실시간 시스템,
소프트웨어공학
Email : kwlee@hansung.ac.kr