어플리케이션 실행 특성 분석을 통한 모바일 시스템 성능 최적화 연구

조중석, 최창문, 정인상, 조두산, 정유진

순천대학교 전자공학과
e-mail : dscho@scnu.ac.kr

A Study of mobile system performance optimization through analysis of application execution characteristics

Jungseok Cho, Chang-mun Choi, In-sang Jeong, Doosan Cho, Youjin Jung
Dept. of Electronic engineering, Sunchon University

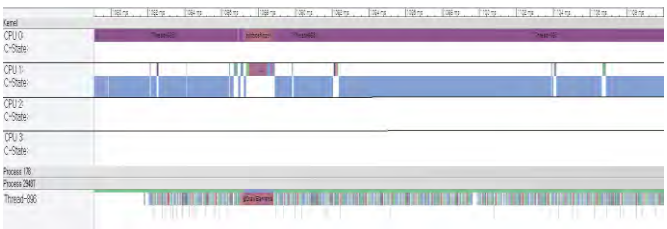
요 약

모바일 디바이스의 보급으로 사람들의 생활에 많은 변화를 가져왔으며, 이러한 변화에 따라 점차 수요에 따른 모바일 콘텐츠 시장 또한 확산 되었다. 사람들의 수요에 의해 모바일 애플리케이션은 문서작성, 게임, 사진, 은행 업무, 영화, 벨소리 뿐 아니라 HD 비디오 재생, 스트리밍 AV 서비스 등 하드웨어적 고성능을 요구하는 애플리케이션까지 등장하게 되었다. 이러한 추세에 더불어 모바일 디바이스는 멀티코어의 성능에 이르는 디바이스까지 출시 되었다. 하지만 멀티코어의 효율성은 스케쥴러가 코어에 작업을 할당하는 방법에 따라 달라진다. 이종 멀티 코어 플랫폼에서 애플리케이션의 실행 시간은 실행되는 코어에 의존한다. 본 논문에서는 프로파일에 의해 각 태스크의 실행 시간을 분석하여 태스크 스케쥴링 기법을 제안한다.

1. 서론

현제 세계 모바일 기기 이용자 수는 약 32 억 명에 달하며, 지난 4 년 동안에만 10 억 명의 이용자가 증가 했다, 이러한 증가 추세는 2017 년까지 모바일 기기 이용자 수는 약 7 억명 이상이 증가하여 2018 년에는 세계 모바일 인구가 40 억 명을 돌파할 것으로 예측된다 [1]. 이러한 추세와 더불어 모바일 애플리케이션 시장의 확산 또한 빠르게 일어나고 있는 것이 현실이다. 모바일 애플리케이션의 시장의 확장에 대한 기여는 모바일 디바이스의 보급에 의함을 알 수 있다. 모바일 디바이스, 스마트폰과 스마트패드 등은 소형 컴퓨터로써 프로세서와 메모리가 탑재된다. 모바일 프로세서는 배터리로 구동되는 모바일 임베디드 컴퓨팅으로 특화 설계된 CPU 를 의미하며, 현재 스마트기기에 사용되는 SoC 형태의 AP 가 모바일 프로세서의 대명사가 된다. 데스크탑 CPU 제조사들은 2006 년 이후 폭증하는 성능 요구와 전력 소모 문제를 해결하기 위해 멀티코어 아키텍처로 전환 하였다. 이와 마찬가지로 모바일 CPU 역시 모바일 기기에 대한 성능 요구량이 증가하면서 같은 도전에 직면 하였다. 사람들은 모바일 기기에 PC 와 같은 수준의 기능과 성능을 요구하면서 배터리 수명 시간 연장을 원하지만, 배터리 수명 증가 속도는 매우 느리므로 결국 고성능일지라도 프로세서의 전력 소모를 감소시키거나 증가를 최소한으로 억제해야 하며, HD 비디오 재생, 스트리밍 AV 서비스, 멀티태스킹, 웹브라우징, 3D 게임 등 싱글코어 프로세서로는 감당하기 어려운 성능이 요구된다. 그렇기 때문에 현재에 이르러서는 옥타코어의 성능을 가지는 디바이스들이 보급되고 있는 추세이다 [2][3].

하지만 이러한 디바이스 성능 향상에도 불구하고 애플리케이션의 실행에 있어서 자원 관리 및 분배에는 효율적이지 못한 것이 현실이다. 아래 그림 1 을 보면 애플리케이션 실행 시에 Thread 의 대부분을 CPU 0 에서 처리를 하는 것을 알 수 있다. 이는 성능 감소와 에너지 효율 측면에서도 비효율적이다. 따라서 모바일 디바이스의 성능 개선 및 애플리케이션의 효율적인 자원 분배와 최적화를 위해서 애플리케이션 실행 특성의 분석이 필요하다.



(그림 1) 애플리케이션 실행 동안 CPU 상태

2. 관련연구

2.1 부화 균등화

부화 균등화는 프로세서에 부하의 재분배를 통해 병렬 및 분산 시스템의 성능을 향상시키는 과정이다. 부하 균등화의 주요 목표는 처리량을 최대화하기 위해 프로세서에 작업을 분산, 안정성을 유지하고 무결성을 가져야한다.

부하 균등화는 정적 부하 균등화와 동적 부하 균등화로 구분할 수 있다. 정적 부화 균등화는 프로세서의 실행의 시작 부분에 결정된다 [4][5]. 그런 다음 성능에 따라 작업 부하는 마스터 프로세서에 의해 시작에 분포한다. 동적 부하 균등화는 런타임 프로세서에 분산된다는 점에서 정적 알고리즘과 다르다. 마스터 프로세서는 수집된 새로운 정보에 기초하여 슬레이브 프로세서에 새로운 프로세스를 할당한다 [6][7].

이러한 부하 균등화 알고리즘은 분산 시스템에 초점을 맞추고 SoC 시스템에 고려한 알고리즘이 없다. SoC 의 중요한 특성은 분산시스템의 속성과 다르다. 주요 요인은 자원의 수에 따른 자원 분석 오버헤드의 고려가 필요하다.

분산 시스템에 대한 부하 균등화 알고리즘은 일반적으로 가벼운 부하, 일반적인 부하, 무거운 부하 세 가지 부하 상태의 통계를 사용한다 [8]. 그러나 SoC 에서 단독으로 세 개의 상태 시스템 부하 메커니즘은 최적이 아니다. SoC 는 보통 다른 성능 및 리소스의 성능의 다양한 자원의 성능을 가지고 있기 때문에 자원의 부하 상태로 변화한다. 따라서, SoC 에 대한 로드 밸런싱 알고리즘은 최적의 시스템 성능을 위해 성능과 자원의 부하 상태를 고려해야 한다 [9].

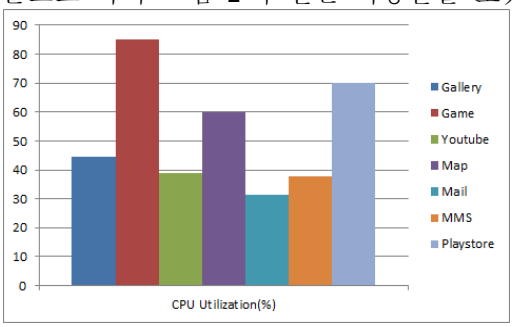
2.2 멀티프로세서에서 작업 분배

MPSoc 시스템에서는 작업의 크기에 따라 적절히 프로세서 코어에 작업을 할당하여 병렬처리 함으로써 시스템 처리 속도를 높이고자 하는 목적이 있다. 멀티프로세서에서 사용되었던 기법으로는 대표적으로 리스트 스케쥴링 기법 [10]과 클러스터링 기법 [11]이 있다. 리스트 스케쥴링 기법은 태스크들에 우선순위를 부여하여 그 일련의 순서대로 스케쥴링하는 방식이다. 클러스터링 기법은 최소 스케쥴 시간이 될 수 있도록 태스크 서로간의 통신비용이 많은 태스크들을 하나의 그룹으로 묶어서 이 그룹 단위로 프로세서 코어에 할당하는 방식이다. 멀티프로세서의 스케쥴링 기법들은 우선 높은 통신비용을 해결하는데 초점을 맞추어 설계되었다. 따라서 통신비용이 상대적으로 낮은 모바일 시스템의 시스템 온 칩에서는 이러한 기술을 그대로 사용하는데 효율적인 측면에서 어려움이 있다.

리눅스 기반인 안드로이드 에서도 프로세서 자원을 효율적으로 활용하여 사용자 응답성을 높이기 위한 스케쥴링 기법이 사용되고 있다. 하지만 그럼에도 불구하고 사용자 응답성 측면에서 부정적인 평가가 보고되었다 [12]. CFS 같은 경우 태스크들에게 공평한 CPU 할당 시간을 주도록 설계되어 있다. 하지만 리눅스 서버 환경에 특화되어 있어 대화형 애플리케이션 환경에서는 효율적인 작업 할당이 되지 않는 것으로 알려져 있다 [13]. 현재의 안드로이드 버전까지도 탑재된 MPSoC 의 프로세싱 코어들의 효율적인 사용을 위한 스케쥴링 기법이 적용되지 못하고 있는 실정이다.

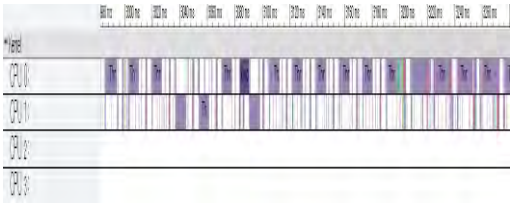
3. 애플리케이션 실행 분석

애플리케이션의 실행 분석 기준은 CPU 사용률을 기준으로 한다. ADB(Android Debug Bridge)를 통하여 디바이스에서 실행되는 애플리케이션을 분석 한다. 애플리케이션은 사용 빈도가 높은 앨범, 게임, 유투브, 지도, 메일, MMS, Playstore 로 하였다. 각각 실행시간은 10 분으로 하여 그림 2 와 같은 사용률을 보였다.

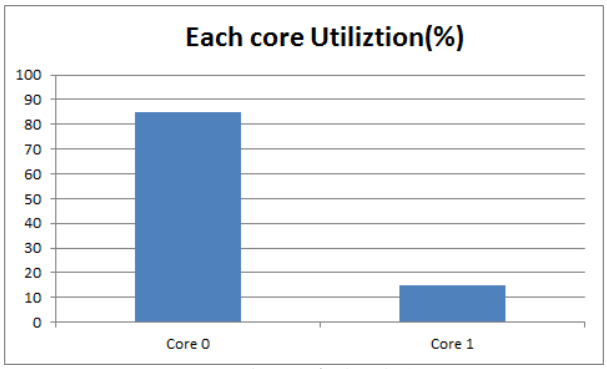


(그림 2) 애플리케이션 CPU 평균 아용률(%)

각각의 애플리케이션의 각 CPU 사용률을 측정하기 Systrace 를 이용하여 분석한다. Systrace 분석은 애플리케이션 중 CPU 사용률이 가장 높은 Game 을 기준으로 분석한다. 아래 그림 3 과 그림 4 에 그 결과가 나타난다.



(그림 3) 게임에 대한 Systrace 분석

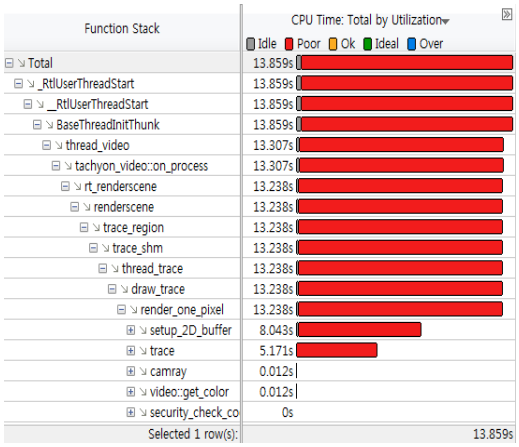


(그림 4) 각 코어의 사용률(%)

위 그림의 결과를 바탕으로 4 개의 코어를 사용하는 디바이스에서 애플리케이션 실행 시 사용되는 코어는 2 개가 된 다는 것을 알 수 있다.

각 코어별 사용률에 영향을 미치는 요소들을 분석하기 위해 동적 코드 분석 도구를 사용하여 실행 가능한 이진 프로그램에 대한 동적 분석을 실행한다.

동적 코드 분석 도구는 동적 컴파일을 통한 동적 코드 변환 기법[14]을 사용한다. 프로그램 실행을 통하여 핫스파스 하드웨어 이벤트를 측정하여 실제적으로 프로그램이 가지는 스레드와 함수, 명령어들이 하드웨어에 미치는 영향들을 분석한다. 아래 그림 5 에 분석에 대한 결과를 나타내었다.



(그림 5) 프로그램 분석에 대한 결과

4. 알고리즘

위 측정 결과를 토대로, 효과적인 CPU 사용을 위한 알고리즘을 제시한다. 애플리케이션의 분석을 통하여, 애플리케이션이 동작하는 각 함수들의 실행시간 및 CPU 사용이 되는 각 코어들과 사용되지 않는 코어들에 대해 파악할 수 있었다. 이를 바탕으로 효과적인 CPU 사용을 위한 알고리즘을 제시한다. 코어의 할당은 실행 프로그램의 함수 실행 시간을 기준으로 한다.

PE={1, 2, 3, ..., n}

F={1, 2, 3,...,k}

Load(PEi)=

normalized_execution_time(allocated(f,PEi))/speed(PEi)

$$AveLoad = \frac{\sum_{i=1}^{n} Load(PE_i)}{n}$$

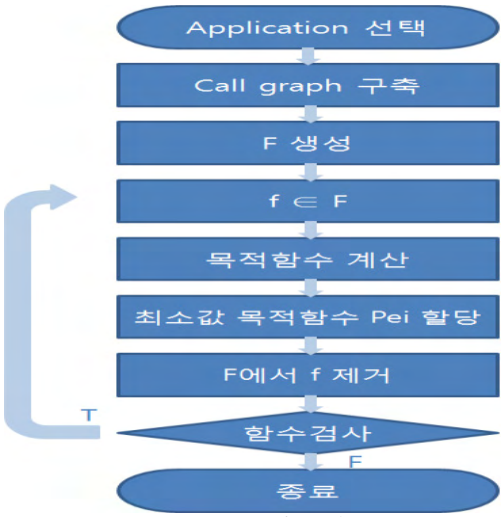
Objective function =

$$Minimize\left(\sqrt{\frac{\sum_{i=1}^{n}(Load(PE_i) - AveLoad)^2}{n}}\right)$$

PE는 MPSoC 프로세서 코어 집합이며, F는 애플리케이션 함수 집합, Load(PEi)는 할당된 함수 집합 f ∈ F to PEi를 나타낸다. 애플리케이션의 작업량을 정규화하기 위하여 일반 PC에서 동일한 환경을 구성하고 함수단위로 실행시간을 반복 측정하여 그 평균값을 함수의 작업량으로 정의 하였다. 제안하는 기법의 최종 목적은 코어 별로 작업량을 균등하게 하여 시스템 성능을 최적으로 하는데 있다. 이를 달성하기 위하여 목적 함수는 코어들에 할당된 작업량의 표준 편차를 최소화하도록 설계하였다. 알고리즘 동작은 아래 그림 6과 같다.

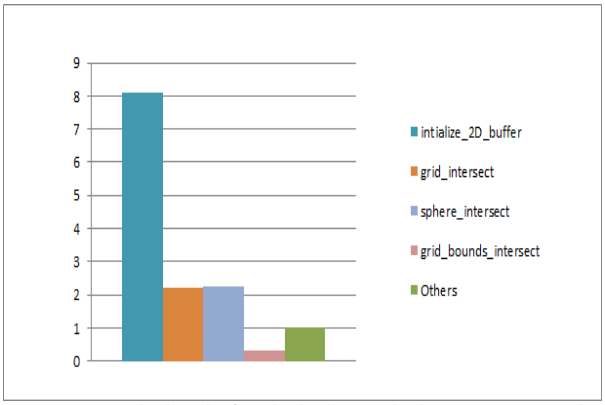
5. 실험결과

제시하는 알고리즘을 검증하기 위해, Microsoft Visual Studio 2012 와 Intel Vtune Amplifier XE 2013 버전을 사용하였다. 모바일 디바이스 상의 애플리케이션의 디버깅 제한으로, PC 에서 실험 환경을 구성하여 측정하였다.



(그림 6) 알고리즘

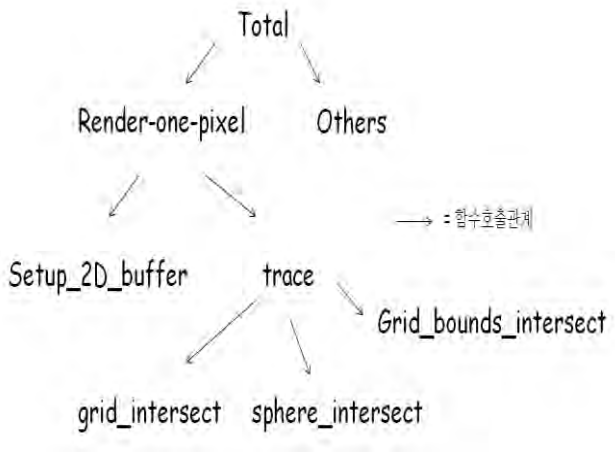
알고리즘을 적용시키기 위해 데이터 버퍼에 이미지를 그리는 프로그램과 사진 보기 프로그램, 사진 편집 프로그램, 시스템 UIAClient 프로그램, Sudoku 프로그램 총 5 개의 프로그램에서 실행되는 함수들의 CPU 시간을 측정한다. 아래 그림 7 은 데이터 버퍼 이미지 그리기 프로그램에 대한 각 함수들의 CPU 사용시간을 나타낸다.



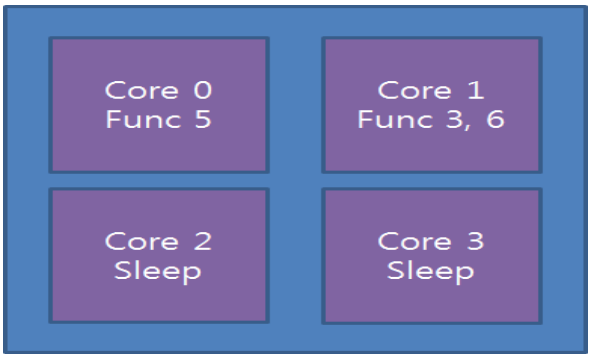
(그림 7) 데이터 버퍼 이미지 그리기 프로그램 각 함수들의 CPU 사용시간

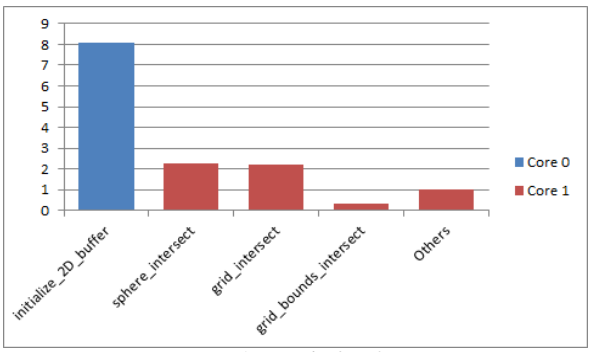
위 결과를 토대로 함수의 호출 관계를 분석하여 알고리즘에 적용시킨다. 함수의 호출 관계는 그림 8 에 나타난다. 호출 관계의 분석 후, 제시하는 알고리즘을 적용하면 그림 9 과 같이 된다.

제시하는 알고리즘을 통하여 각 함수에 대해 코어 분할을 할시 코어 0 과 코어 1 의 사용시간이 각각 8.079s, 5.834s 가 된다. 즉, CPU 사용시간이 8.079s 가 된다는 것이다. 분할하기 전 CPU 시간 13.913s 보다 약 41%의 시간 이득이 있다는 것을 확인할 수 있다.

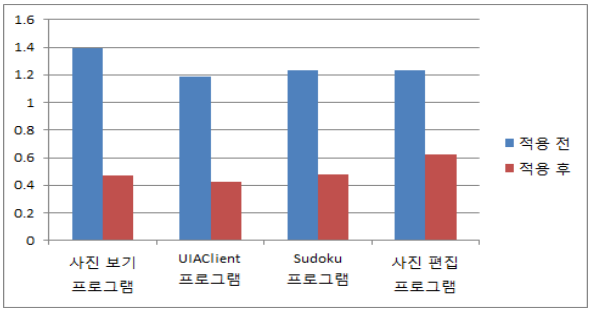


(그림 8) 함수 호출 그래프





(그림 9) 각 코어별 함수 분할



(그림 10) 알고리즘 적용 전 후 CPU 사용 시간 비교

위 방법을 토대로, 사진보기 프로그램, UIAClient 프로그램, Sudoku 프로그램, 사진 편집 프로그램에 각각 적용시켰을 시 그림 9와 같이 된다.각각의 프로그램은 각각 약 66%, 64%, 61%, 49%의 CPU 사용 시간 감소를 가져온다.

6. 결론

본 논문에서, 사용자 상호작용이 높은 7 개의 애플리케이션의 실행 특성을 분석을 통하여 모바일 디바이스 상에서 애플리케이션의 비효율적인 실행을 검증하였고, 효율적인 자원 분배 및 성능 향상을 위한 알고리즘을 제시하였다. 제시하는 알고리즘으로 5 개의 실행 가능한 프로그램으로 실험을 통하여 각 프로그램의 평균 56%의 이득을 얻을 수 있음을 확인하였다. 본 논문에서 제시한 알고리즘을 통하여 모바일 디바이스의 성능 향상과 전력효율의 개선이 기대된다.

Acknowledgement

이 논문은 2010 년도 정부 (교육부)의 재원으로 한국연구재단 기초연구사업(2010-0024529), 2014 년도 정부 (교육부)의 재원으로 한국과학창의재단 (대학생 창의융합형 연구과제 지원사업)의 지원을 받아 수행된 연구임.

참고문헌

[1] KOCCA 한국콘텐츠진흥원 “세계 모바일 애플리케이션 시장 현황 및 전망”
[2] 방송통신위원회 “신규모바일 기기 정보보호 연구” 2011. 12
[3] KETI 한국산업기술평가관리원 “모바일 CPU 기술 동향 및 산업 전망”
[4] Derek L. Eager, Edward D. Lazowska , John Zahorjan, “Adaptive load sharing in homogeneous distributed systems”, IEEE Transactions on Software Engineering, v.12 n.5, p.662-675, May 1986.
[5] R. Motwani and P. Raghavan, “Randomized algorithms”, ACM Computing Surveys (CSUR), 28(1):33-37, 1996
[6] S. Malik, “Dynamic Load balancing in a Network of Workstation”, 95.515 Research Report, 19 November, 2000.
[7] Y.Wang and R. Morris, "Load balancing in distributed systems," IEEE Trans. Computing. C-34, no. 3, pp. 204-217, Mar. 1985.
[8] Gil-Haeng Lee, “An Adaptive Load Balancing Algorithm Using Simple Prediction Mechanism” Database and Export Systems Applications, 1998, pp. 496–501.
[9] Shinwon Lee, Meka, V., Mingu Jeon, Nagoo Sung, Jeongnam Youn “Dynamic load balancing algorithm for system on chip”, SoC Design Conference (ISOCC), 2012 International, 2012,, pp. 208 – 211
[10] Hesham El-Rewini and T.G. Lewis, “Scheduling parallel program tasks onto arbitrary target machines”, Journal of parallel and distributed computing, Feb. 1990, vol9, No2, pp.138-153.
[11] A. Gerasoulis and T. Yang, “On the granularity and clustering of distributed acyclic task graphs”, IEEE Trans. parallel and distributed systems, Vol. 4, No. 6, pp. 686-701, June 1993.
[12] http://techland.time.com/2011/12/07/is-android-oomed- to-lag-more-than-ios/
[13] L.A. Torrey, J. Coleman, and B. Miller, “A comparison of interactivity in the linux 2.6 scheduler and an MLFQ scheduler,” Software: Practice and Experience, vol. 37, no.4, pp.347-364, 2007.
[14] Hesham El-Rewini and T.G. Lewis, “Scheduling arallel program tasks onto arbitrary target machines”, Journal of parallel and distributed computing, Feb. 1990, vol9, No2, pp.138-153.