공학석사학위논문

이기종 매니코어 시스템 자원 관리를 위한 스케쥴러 개발

Developing scheduler to manage resources for heterogeneous many core systems

2013년 2월

서울대학교 대학원

컴퓨터 공학부

최 국 태

이기종 매니코더 시스템 자원 관리를 위한 스케쥴러 개발

Developing scheduler to manage resources for heterogeneous many core systemes

지도교수 염 헌 영

이 논문을 공학석사학위논문으로 제출함

2012년 12월

서울대학교 대학원

컴퓨터 공학부

최 국 태

최국태의 석사학위논문을 인준함

2012년 12월

위 원 장 민 상 렬 (인)

부 위 원 장 염 헌 영 (인)

위 원 엄 현 상 (인)

초록

PC에서와 마찬가지로 최근의 스마트폰에서는 매니코어와 GPU를 포함하는 시스템들이 일반적이 되었다. 이러한 매니코어와 GPU를 포함하는 시스템의 성능을 높이기 위해서는 실행 중인 태스크들 간의 패러랠리즘을 높여야 한다. 패러랠리즘을 높이기 위해서는 애플리케이션 자체가 병렬로 실행될 수 있도록 프로그램되어야 하고, 운영체제가 실행 중인 태스크들 간에 시스템의 자원을 효율적으로 할당할 수 있어야 한다. 이러한 목적으로 시스템 스케쥴러의 로드 밸런싱 기능이 중요하게 되었다.

안드로이드 플랫폼은 리눅스의 커널을 사용하고 있으며 따라서 스케쥴러도 리눅스의 CFS를 그대로 사용하고 있다. CFS 스케쥴러는 로드 밸런싱을 위해 한 코어에서 다른 코어로 옮길 태스크 선택시에 pair-wise 밸런싱과, 태스크의 특성을 고려하지 않는 blind selection을 이용한다. 실험을 통하여 이러한 CFS의 로드 밸런싱 방식은 계속해서 태스크들을 코어 간에 옮기는 불필요한 오버헤드와 특정 코어에서 태스크들간의 성능 간섭을 일으켜서 모바일 기기의 성능을 떨어뜨릴 수 있음을 발견하였다.

이 논문에서는 리눅스의 cgroup 기능을 이용하여, 태스크의 CPU intensiveness와 같은 특성을 스케쥴러에게 전달할 수 있도록 하였다. 스케쥴러는 이런 특성을 이용하여 태스크의 특성에 따라 시스템 자원을 효율적으로 태스크에 할당함으로써 안드로이드 플랫폼의 애플리케이션 성능을 향상시킬 수 있다. 실험을 통하여 약 15% 정도의 성능 개선을 발견하였으며, 이 방식은 CPU intensiveness 특성 이외에 memory, block I/O, network bandwidth 같은 특성으로 확장 가능하다.

주요어 : 로드 밸런싱, cgroup, 안드로이드 플랫폼, 리눅스, 태스크 스케줄링

학 번 : 2011-20941

목차

표 목차

그림 목차

제 1 장 서론

2004년 이전까지 프로세서의 클럭 스피드는 “Moore’s Law"에 따라 약 18~24개월 주기로 2배씩 증가하였으며, 그에 따라 애플리케이션의 실행 속도도 증가하였다. 하지만 2004년 이후부터 소모 전력과 발열 등의 문제로 더 이상 프로세서의 클럭 스피드를 높이는 것이 힘들어지자, CPU 제조업체들은 동일한 칩 위에 많은 수의 코어를 집적하여 성능 향상을 꾀하고 있다. 이 외에도 그 동안은 PC의 그래픽 성능 향상을 목적으로 주로 사용되던 GPU들이 범용의 코프로세서로 사용됨으로써 컴퓨터의 실행속도는 동시에 수행되는 태스크들 간의 패러랠리즘을 얼마나 높이느냐에 따라 달라지게 되었다.

태스크들 간의 패러랠리즘을 높이기 위해서는 해당 애플리케이션 자체의 병렬화 처리가 잘 되어 있어야 할 뿐만 아니라, 운영체제 차원에서도 컴퓨터의 자원을 효율적으로 처리할 수 있어야 한다. 따라서 CPU 스케쥴러의 로드 밸런싱 기능이 중요하게 되었다.

이러한 현상은 PC 뿐만 아니라 모바일 환경으로도 확대되어서 최근의 스마트폰들은 듀얼코어, 쿼드코어를 탑재하는 것이 대세이며, 2013년 이후에는 8 코어 CPU를 탑재한 스마트폰의 등장이 예고되고 있는 실정이다. 또한 게임 등을 위한 그래픽 처리 성능을 높이기 위하여 GPU를 탑재하는 스마트폰이 늘어나고 있으며, 스마트폰 GPU에서의 GPGPU 지원도 곧 현실화 될 것으로 예상된다.

그러나 “스마트폰에서는 사용 가능한 전력량, 동시에 실행 가능한 애플리케이션 수의 제한 등의 문제로 고사양의 매니코어가 꼭 필요한가?” 라는 의문과, 스마트폰의 대표적 운영체제인 안드로이드가 매니코어 CPU

를 제대로 처리하지 못하고 있다는 비판이 있다. 하지만 고 해상도의 스마트폰과 테블릿의 등장, 많은 애플리케이션을 동시에 실행시키는 시나리오의 개발, 고 사양의 하드웨어 자원을 필요로 하는 모바일 게임 등의 등장으로, 이제 모바일에서도 이러한 메니코어 및 GPU의 지원은 거스를 수 없는 대세인 것으로 판단된다.

이 논문에서는 대표적인 모바일 운영체제인 안드로이드에서 매니코어 및 GPU와 같은 이기종 자원들을 효율적으로 사용하기 위한 스케쥴러를 개발하려고 한다.

안드로이드에서는 기반이 되는 리눅스 커널의 스케쥴러를 그대로 사용하고 있다. 한때 BFS로 스케쥴러를 변경하여 사용하기도 하였으나, 여러 가지 문제로 현재는 리눅스 커널 메인라인의 CFS(Completely Fair Scheduler)를 그대로 사용하고 있다. CFS 스케쥴러는 로드 밸런싱을 위해 이동한 태스크 선정시에, pair-wise 방식과 실질적으로 CPU-bound, I/O-bound 등의 태스크 특성이 무시되는 blind selection 방식을 사용하고 있어, 코어들간의 로드 밸런싱을 위한 태스크 이동이 계속하여 발생함으로 interactivity가 중요시되는 모바일 상황에서 치명적인 약점을 보이고 있다.

이 논문에서는 리눅스의 control group 기능을 이용하여, 스케쥴러에 각 태스크들의 CPU intensiveness 특성 정보를 제공함으로써, 로드 밸런싱 성능의 향상을 꾀하였다. 실험을 통하여 이러한 방식이 안드로이드 플랫폼 애플리케이션의 실행 속도를 약 15% 정도 향상 시킴을 확인하였다. 그리고 이러한 방식은 태스크들의 memory, block I/O, network bandwidth 와 같은 특성으로도 확대하여 적용할 수 있다는 것을 알게 되었다.

이 논문의 나머지 부분은 다음과 같이 구성된다. 2장은 안드로이드 리

눅스 커널 스케쥴러(CFS)의 로드 밸런싱 기법에 대해 살펴보고 3장은 리눅스의 cgroup 기능에 대해서 고찰하여 보고 안드로이드 플랫폼에서의 애플리케이션 현황 및 로드 밸런싱 기법을 알아보고, 문제 상황을 도출하고, 5장에서 이 논문에서 제안된 기법을 설명하고 6장에서 실험 결과를 보이며 마지막으로, 7장에서 논문의 결론을 맺는다.

제 2 장 배경

본 장에서는 안드로이드 플랫폼의 구성과, 안드로이드 플랫폼이 사용하고 있는 리눅스 커널 스케쥴러의 로드 밸런싱 기법에 대해서 고찰해 본다.

2.1 안드로이드 플랫폼

안드로이드는 구글에서 만들어 오픈 소스로 배포하는 스마트폰 및 태블릿PC와 같은 모바일 장치를 위한 소프트웨어 플랫폼이다. 이는 리눅스 커널, C/C++ Native 라이브러리, 안드로이드 런타임, 애플리케이션 프레임워크와 휴대폰 기본 애플리케이션으로 구성되어 있다.

- **커널**

안드로이드는 메인라인 리눅스 커널을 기반으로 모바일 상황에 맞춘 여러 가지 모듈을 추가/수정하여 사용하고 있다. 안드로이드에서 대체하여 사용하고 있는 부분들은 Binder(IPC 매커니즘), 파워 매니지머트, Ashmen(공유 메모리 관리), Low Memory Killer(Out of Memory Killer) 등과 여러 가지 디버깅 툴을 추가하였다. 스케쥴러도 메인라인 리눅스와는 달리 한때 BFS를 도입하여 사용하였으나, 여러 가지 문제로 현재는 메인라인 리눅스와 동일하게 CFS를 사용하고 있다. 안드로이드 Jellybean 버전은 리눅스 커널 3.0.31 버전을 바탕으로 하고 있다.

- **C/C++ Native 라이브러리**

안드로이드는 성능을 위해 C/C++ 언어로 작성된 라이브러리 집합을 제공한다. 이는 기존의 리눅스 관련 C 표준 라이브러리들과 안드로이드 전용으로 새로 작성되거나 외부 오픈소스에서 추가된 Surface Manager(UI Window 제어), 2D/3D 등의 그래픽 라이브러리, 멀티미디어 코덱, SQLite DB 엔진, webkit 엔진 등과, 사용하는 하드웨어 디바이스에 따라 달라지는 HAL(Hardware Abstraction Layer)로 구성되어 있다.

- **안드로이드 런타임**

안드로이드 런타임은 달빅 가상 머신과 자바 코어 라이브러리로 구성되어 있으며, 프로세스가 동작하면서 라이브러리를 호출할 때 사용된다. 달빅 가상 머신은 비표준 자바 가상 머신으로 작은 메모리에 최적화되어 있는 달빅 Excutable(DEX) 바이트 코드를 실행한다. 개별 프로세스는 독립적인 달빅 가상 머신을 할당받아 별개의 인스턴스로 동작하게 된다.

- **애플리케이션 프레임워크**

애플리케이션 프레임워크는 Java API 기반의 인터페이스를 제공하는 계층으로 Java로 이루어진 Binder 서비스와, Activity Manager, View Sytem, Window Manager, Package Manager 드의 매니저로 구성되이 있다.

● 애플리케이션

스마트폰에 기본적으로 필요한, 홈스크린, 컨택(주소록), 브라우저, 이메일, 전화 등의 애플리케이션이 탑재되어 있다.

그림 2. 안드로이드 플랫폼 구조

2.2 리눅스 스케줄러(CFS)의 로드 밸런싱

멀티코어 시스템의 실행 속도는 일반적으로 동시에 수행되는 태스크들 간의 패러랠리즘을 얼마나 높이느냐에 따라 좌우된다고 한다. 태스크들 간의 패러랠리즘을 높이기 위해서는 애플리케이션 자체가 병렬화를 효과

적으로 지원하도록 프로그래밍 되어야 하며, 운영체제 차원에서는 동시에 수행되는 태스크들이 여러 개의 코어 뿐만 아니라 메모리 등과 같은 자원을 효율적으로 사용할 수 있도록 분배해야 한다. 이에 따라 멀티코어 시스템에서는 운영체제 스케쥴러의 로드 밸런싱 기능이 중요하게 되었다.

스케쥴러의 로드 밸런싱 기능은 사용하는 Run-queue와 각 코어에 로드를 분배하는 방식에 따라 분류할 수 있다. 사용하는 Run-queue에 따라 분류하면, 각각의 코어가 개별 Run-queue를 가지고 있으면서, 싱글 코어 시스템과 동일하게 자신의 큐에 할당된 테스크들을 스케쥴하는 방식과, 글로벌 큐를 가지고 스케쥴러의 정책에 따라 모든 코어에 균등하게 태스크들을 분배하는 방식이 있다. 글로벌 큐를 사용할 경우 스케쥴러의 정책에 따라 모든 큐에 태스크들을 균등하게 분배할 수 있으나, 태스크들의 Processor Affinity를 보장할 수 없어서 각 코어의 캐쉬 사용의 이점이 무시된다. 개별 큐를 사용하는 방식은 태스크들의 Processor Affinity를 보장할 수 있으나, 개별 큐에 할당된 태스크들의 로드 간 차이가 발생할 경우 로드 밸런싱을 위해 태스크를 옮겨야하는 오버해드가 발생한다.

로드 밸런싱을 위해 다른 큐로 옮길 태스크를 선택하는 방식에 따라서도 모든 큐의 최종 balanced 포인트를 정하고 태스크를 옮기는 Poly-wise 방식과, 두 개 큐간의 로드 밸런싱 을 실시한 후 모든 큐로 확대해 나가는 Pair-wise 방식이 있우며, 옮길 태스크들의 특성(CPU-bound, I/O-bound 등)을 고려하여 선택하는 방식과 단지 옮길 태스크들의 개수만을 고려하는 방식으로 분류할 수 있다.

리눅스 커널의 3가지 대표적인 스케쥴러인 O(1), CFS, BFS의 로드 밸런싱 방식을 간단히 비교해 보면 다음과 같다. 먼저 O(1) 과 CFS는 개별

큐를 사용하고, BFS는 글로벌 큐를 사용한다. 따라서 BFS에서는 태스크들이 계속해서 코어들을 옮겨 다니게 되며, O(1)과 CFS에서는 태스크 migration 이 일어나는 특정 시간 동안 이외에는 동일한 코어에서 실행된다. 다른 큐로 옮길 큐를 선택하는 방식에 있어서는 O(1)의 경우 Poly-wise 방식을 사용하고, 스케쥴러 내부에서 정한 interactivity bonus/penalty에 따라 정해진 내부 priority를 고려하여 선택을 하고, CFS는 Pair-wise 방식을 사용하며, 사용자단에서 지정한 Priority에 따라 결정된 웨이트 값에 기반하여 선택한다.

현재 안드로이드가 사용하고 있는 CFS 스케쥴러의 로드 밸런싱 방식을 좀 더 자세히 알아보면 다음과 같다. CFS는 가중치 기반 로드 발런싱을 통해 Run-queue간 로드를 균등하게 배분한다. run-queue Q_k의 로드는 다음과 같이 정의된다.

$$L_k = \sum_{\tau_i \in S_k} W(\tau_i) \qquad (1)$$

여기서 S_k는 Q_k안에 속해 있는 태스크들의 집합이다.

CFS는 특정 주기 T마다 로드 밸런싱을 수행한다. CFS는 run-queue Q_k에 대한 마지막 로드 밸렁싱 time인 $t_{last,k}$를 유지하고 스케줄러 tick마다 $t_{current}$와 $t_{last,k}$를 비교함으로써 Q_k가 다시 로드 밸런싱을 해야 할지 결정한다. 만약 $t_{current}$가 $T+t_{last,k}$보다 크다면 이는 run-queue $Q_{busiest}$에서 Q_k로 태스크들을 이주시키는 로드 밸런싱을 발생 시킨다.

CFS는 다수의 태스크들을 이주 시킬 수 있으며 이주되는 로드의 양은 다음과 같이 정의된다.

$$L_{imbal} = \min(L_{busiest} - L_{avg}, L_{busiest} - L_{busiest\_load\_per\_task}, L_{avg} - L_k) \quad (2)$$

여기서 $L_{busiest}$는 $Q_{busiest}$의 로드이며 L_{avg}는 전체 시스템의 평균 로드 값이다. $L_{busiest_load_per_task}$는 $Q_{busiest}$의 로드를 $Q_{busiest}$안에 속해 있는 태스크의 수로 나눈 값이다. CFS는 다음과 같은 조건일 때 태스크를 이주 하지 않는다.

$$L_{imbal} < W(\tau_i)_{\tau_i \in S_{busieset}} / 2 \quad (3)$$

여기서 $S_{busiest}$는 $Q_{busiest}$안에 태스크들의 집합이다.

CFS는 run-queue가 비어 있을 때마다 추가로 로드 밸런싱을 발생시킨다. 이런 경우에 busiest run-queue에서 태스크들을 이주시키고 그 양은 (2)에 의해 정의된다. 이와 같은 방법으로 CFS는 두 개 코어 간의 로드 밸런싱을 실시하게 된다. 시스템의 코어의 개수가 두 개 이상일 경우는 이와 같은 두 개 코어 간의 로드 밸런싱을 나머지 코어들에 대해서는 연속적으로 수행하게 된다.

쿼드 코어 시스템을 예로 들면, 코어 A, B, C, D에서 코어 D의 로드가 다른 코어들에 비해 낮다고 판단이 되면, 우선 코어 A와 D가 간에 로드 밸런싱을 시도한다. 두 개 코어간 로드 밸런싱이 완료되면, 이제, 코어 B와 D 간의 로드 밸런싱이 시도되고, 이후 코어 C와 D 간의 로드 밸런싱이 시도된다. 이렇게 두 개 코어 간의 로드를 비교해서 밸런싱을 시도하였기 때문에 한 차례 로드 밸런싱이 완료되어도 모든 코어간 로드가 밸

런싱 상태에 이르렀다고는 할 수 없다. 따라서 이 시점에 다시 로드를 비교해서 동일한 형태의 밸런싱을 다시 시도하게 된다. 따라서 최초 로드 밸런싱 시도 시점에 4개 코어에 대한 글로벌 정보를 이용하여 최후 밸런싱 시점의 각 코어별 로드를 예측하여 각 코어에서 다른 코어로 옮겨야할 태스크를 결정하는 poly-wise 한 방식에 비하여, CFS의 pair-wise 한 방식의 로드 밸런싱은 전체 코어간 로드가 균등한 상태가 되기 위해 소요되는 시간이 길고, 더 많은 회수의 코어간 태스크 이동이 발생하게 되어 불필요한 오버해드를 발생시키게 된다.

한 개 코어에서 다른 코어로 옮길 태스크의 선정에는 위에서 태스크들의 웨이트 값을 이용한다고 하였다. 태스크의 웨이트는 해당 프로그램에 부여되는 priority 즉 리눅스의 nice 값에 따라 결정된다. 이 nice 값은 해당 프로그램 작성시 프로그래머가 부여하거나, 시스템의 방침에 따라 프로그램의 기능 및 실행 컨텍스트에 따라 부여되는 값이다. 일반적으로 유저 컨텍스트에서 실행되는 프로그램들은 nice 값으로 0을 갖는 경우가 많으며, 따라서 유저 컨택스트에서 실행되는 어플리케이션들의 웨이트 값은 크게 차이가 나지 않는다. 이와 같이 nice 값에 의하여 결정되는 웨이트로는 특히 유저 컨텍스트에서 실행되는 태스크들 간의 특성을 분별할 수 있는 기준이 되지 않는 경우가 많다. 이와 같이 CFS는 로드 밸런싱을 위해 옮겨야할 태스크의 선택에 있어서, 실질적으로는 blind-selection을 하게 되고, 이는 특정 코어에 CPU-bound 한 특성의 태스크들이 몰려서 해당 코어의 로드가 집중되거나, 태스크들 간의 성능 간섭 현상이 발생할 수 있는 원인이 된다.

CFS 이전 리눅스 스케쥴러인 O(1)에서는 애플리케이션의 nice 값 이외에도 해당 태스크의 실행 시점에 태스크의 interactivity를 판단하여 보너스와 패널티가 부여되고, 스케쥴러는 nice 값에 이와 같은 보너스/패널

티가 합쳐진 값을 해당 태스크의 최종 priority 로 판단하게 된다.

이와 같이 CFS의 pair-wise, blind-selection 형태의 로드 밸런싱 방법은 불필요한 오버헤드와 코어간 성능 간섭이 발생할 수 있지만, 주기적으로 코어들 간의 로드를 체크하고, 태스크들이 이동되기 때문에 특정 코어에 로드가 집중되는 최악의 밸런싱 상태에 머무르는 시간이 길지 않아서, 평균적으로는 시스템 전체 성능에 크게 문제가 되지 않는다고 판단되어진다.

제 3 장 Cgroup(Control Group) 기능

본 장에서는 리눅스의 Cgroup 기능에 대해서 살펴보도록 한다. Cgroup은 태스크들과 그 미래 자식 테스크들을 하이어라키 그룹으로 분류할 수 있는 매카니즘을 제공한다. 이를 통하여 사용자들은 시스템에서 실행중인 태스크들을 특정 그룹으로 묶고, 해당 그룹에 CPU, CPU 시간, 시스템 메모리, 네트웍 밴드위스와 같은 시스템 리소스를 할당할 수 있다.

Cgroup은 태스크 그룹들이 트리 형태를 이루는 하이어라키를 형성하며, 한 개의 그룹은 프로세스 하이어라키와 마찬가지로 부모 그룹의 특성을 상속받는다. Cgroup 하이어라키는 지원하는 서브 시스템에 따라 여러 개의 존재할 수 있다. 서브 시스템은 Cgroup의 태스크 그룹핑 기능을 이용하여, 각 그룹에 시스템 자원을 제한하거나 스케쥴링 하는 등 "리소스 컨트롤러" 역할을 한다. 현재 리눅스에서 지원하는 서브 시스템은 아래와 같다.

cpu : Cgroup 태스크들의 특정 CPU률 관리

cpuset : 개별 CPU들과 메모리 노드를 특정 Cgroup에 할당

cpuacct : 태스크들의 CPU 자원 사용 기록 리포트

memory : 태스크들이 사용할 시스템 메모리를 관리

blkio : HDD, SDD, USB 등과 같은 블록 장치의 입/출력 관리

net_cls : 특정 Cgroup에서 발생하는 네트웍 트래픽에 ID 부여

net_prio : 네트웍 인터페스간 Priority 관리

devices : Cgroup에 특정 장치 사용 권한을 관리

freezer : Cgroup 내 특정 테스크의 실행/정지 관리

ns : 네임 스페이스

Cgroup의 사용 예로 학생, 교수 등의 많은 사용자가 있는 대학 시스템의 대규모 서버를 들 수 있다. Cgroup 서브 시스템을 통해 이 서버의 시스템 자원을 아래와 같이 계획할 수 있다.

CPU : "Top cpuset" - CPUSet1, CPUSet2, CPUSet3
CPUSet1 : Professors
CPUSet2 : Students
CPUSet3 : System tasks

Memory : Processors - 50%, Students - 30%, System tasks - 20%

Disk : Processors - 50%, Students - 30%, System tasks - 20%

Network : WWW Browsing - 20%(Professors-15%, Students-5%)
Network File System - 60%
Others - 20%

Cgroup을 사용하기 위해서는 시스템 부팅 시점에 Cgroup 파일 시스템을 마운트하고, 이후 사용 시점에 각 그룹별로 해당 파일 시스템 디렉토리를 생성한다. 각 디렉토리에는 해당 그룹에 포함될 테스크 아이디들을 기입하고, 해당 그룹에 부여될 서브 시스템 어트리뷰트를 추가해 준다. Cgroup 사용을 위해서는 별도의 시스템 콜이 필요하지 않으며, 일반 파일 시스템 API들을 사용하여 구현할 수 있다.

안드로이드 플랫폼에서도 2.1(Eclair) 버전부터 커널에 디폴트로 Cgroup이 포함되었으며, 아래와 같이 부팅 시점(init.c)에 cgroup 파일 시스템을 마운트하고, /dev/cpuctl 이란 cgroup 파일 시스템의 루트 디렉토리를 생성해 두었다. 따라서 /dev/cpuctl 아래에 서브 디렉토리를 생성하면 자동으로 서브 그룹이 생성되고, 해당 디렉토리에 태스크의 아이디를 기

록함으로써, 해당 컨트롤 그룹에 태스크들을 추가할 수 있다. 그리고 안드로이드 플랫폼에서는 백그라운드에서 동작하는 쓰래드들의 CPU 사용 시간을 제한하기 위하여 이미 Cgroup을 사용하고 있다.

```
on init
sysclktz 0
loglevel 3
# setup the global environment
    export PATH /sbin:/system/sbin:/system/bin:/system/xbin
    export LD_LIBRARY_PATH /system/lib
    export ANDROID_BOOTLOGO 1
    export ANDROID_ROOT /system
    export ANDROID_ASSETS /system/app
    export ANDROID_DATA /data
    export EXTERNAL_STORAGE /sdcard
# Backward compatibility
    symlink /system/etc /etc
# create mountpoints and mount tmpfs on sqlite_stmt_journals
    mkdir /sdcard 0000 system system
    mkdir /system
    mkdir /data 0771 system system
    mkdir /cache 0770 system cache
    mkdir /sqlite_stmt_journals 01777 root root
    mount tmpfs tmpfs /sqlite_stmt_journals size=4m
    mount rootfs rootfs / ro remount
    write /proc/sys/kernel/panic_on_oops 1
    write /proc/sys/kernel/hung_task_timeout_secs 0
    write /proc/cpu/alignment 4
    write /proc/sys/kernel/sched_latency_ns 10000000
    write /proc/sys/kernel/sched_wakeup_granularity_ns 2000000
    write /proc/sys/kernel/sched_compat_yield 1
# Create cgroup mount points for process groups
    mkdir /dev/cpuctl
    mount cgroup none /dev/cpuctl cpu
    chown system system /dev/cpuctl
    chown system system /dev/cpuctl/tasks
    chmod 0777 /dev/cpuctl/tasks
.....
```

그림 3. 안드로이드 init.rc 코드

제 4 장 문제 정의

본 장에서는 안드로이드 플랫폼에서 애플리케이션들의 CPU 및 메모리 사용 현황과 운영체제의 로드 밸런싱 방법을 확인해 보고, 안드로이드 플랫폼에서의 로드 밸런싱의 문제점을 확인한 후 이 논문에서 접근하고자 하는 방법의 타당성을 검토해 본다.

4.1 안드로이드 애플리케이션 현황

우선 안드로이드 플랫폼에서의 애플리케이션들의 CPU 사용 현황을 확인해 보았다. 이 실험에는 Goolge의 레퍼런스폰인 Galaxy Nexus를 사용하였으며 안드로이드 플랫폼 버전은 4.1.1 (Linux Kernel 3.031) 이고 CPU는 ARMv7 1200MHz 2Core 이다.

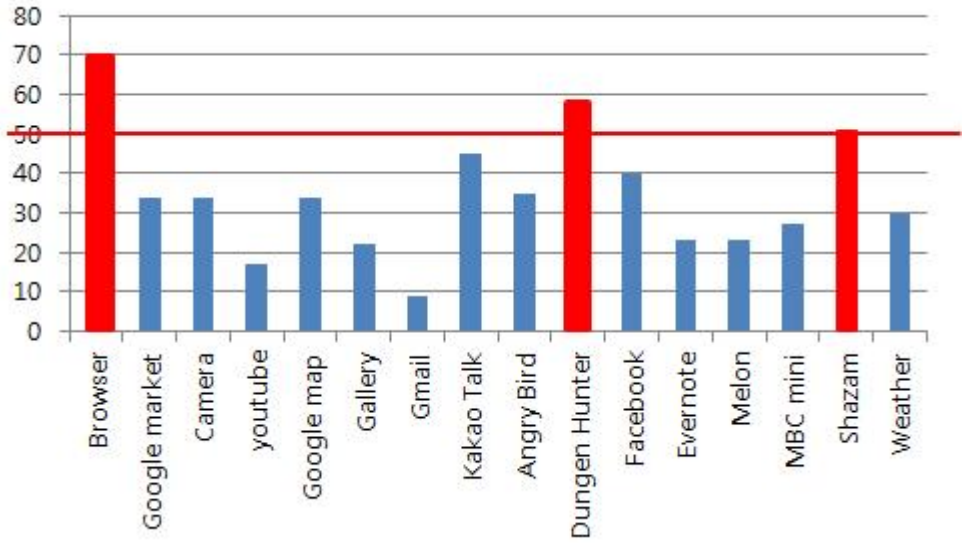


표 4 안드로이드 플랫폼에서 애플리케이션들의 CPU 사용 현황

실험은 안드로이드 플랫폼의 adb shell을 이용하여 top 명령어를 1초 간격으로 실행한 결과들이다. 안드로이드 플랫폼에서 주요 애플리케이션들의 CPU 사용 현황은 표 4.1과 같다. 애플리케이션 중에 CPU 사용률이 50%가 넘어 가는 애플리케이션은 브라우저, 아케이드 게임인 던전 헌트, 음악 매칭 서비스를 제공하는 샤잠과 같이 몇 개의 애플리케이션에 한정되어 있다. 브라우저의 경우 접속하는 사이트에 따라 결과가 달라지지만 모바일 사이트가 아닌 일반 사이트를 접속할 경우 대부분 50% 이상의 CPU 사용 현황을 보였으며, 최대 70~80%까지 올라가는 사이트들도 있었다. 게임의 경우는 화면 갱신 등과 같이 CPU-bound 작업들의 상당수가 GPU에 의해 처리됨으로 상대적으로 CPU 사용 현황은 예상과는 달리 브라우저 보다 높지 않았다. 그 외의 애플리케이션들은 샤잠과 같은 경우를 제외하고 대부분이 50% 이하로 CPU 사용 현황이 아주 높은 것은 아니었다.

다음은 일반적인 스마트폰 사용자들이 많이 이용할 수 있는 다중 애플리케이션 시나리오에서의 CPU 사용 현황을 확인해 보았다. 뮤직 플레이어를 실행하여 음악을 플레이하고, Google 마켓 애플리케이션을 실행하여 사이즈가 큰 애플리케이션을 다운로드 받도록 하였으며, 브라우져를 실행하여 amazon.com 과 같은 비교적 내용이 많고 복잡한 사이트에 접속을 시도하였다. 아래 그림 4.1에서 확인해 볼 수 있는 바와 같이 브라우저(com.android.chrome, com.android.chrome:sandboxedprocess)에서 CPU의 약 60% 정도를 사용하고, 뮤직 플레이어에서 음악을 실행하기 위한 미디어서버(/system/bin/mediaserver)와 애플리케이션 다운로드를 진행하는 와이파이 관련 프로세스(dhd_dpc)가 각각 2% 정도의 CPU를 사용하고 있다. 즉 전면에서 실행중인 애플리케이션 이외에 UI 없이 서비스 형태로 실행되는 애플리케이션들의 CPU 사용 현황은 극히 제한되

어 있으며, 그 외에 백그라운드에서 CPU를 점유하는 프로세스나 쓰레드는 거의 없다는 것을 확인할 수 있다. 즉 안드로이드 플랫폼에서는 전면에서 실행되는 애플리케이션 이외에는 CPU를 점유하는 프로세스가 거의 없거나 일부 서비스가 제한된 양의 CPU를 점유한다는 것을 모든 실험에서 확인할 수 있었다.

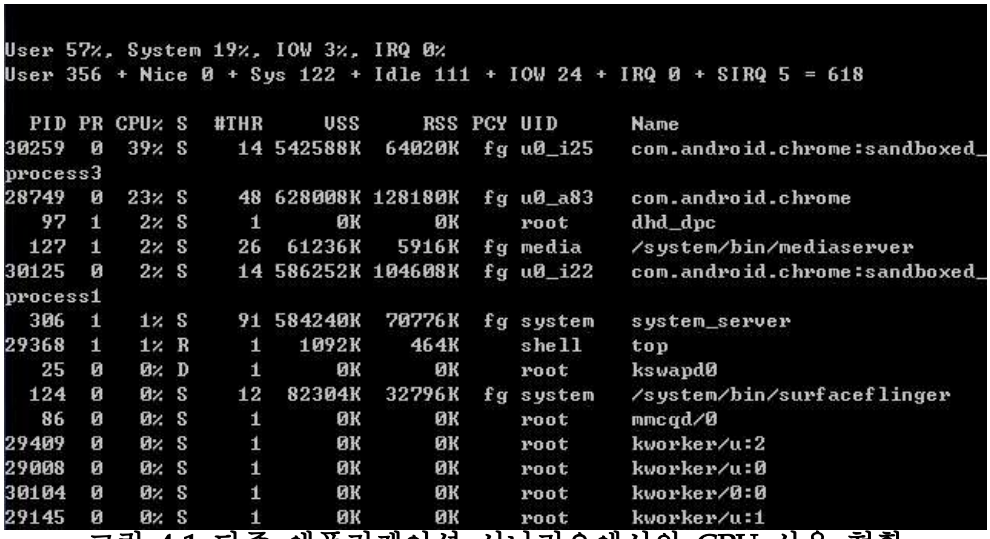


그림 4-1 다중 애플리케이션 시나리오에서의 CPU 사용 현황

4.2 안드로이드 플랫폼의 로드 밸런싱

안드로이드 플랫폼은 리눅스 커널의 CFS 스케쥴러를 그대로 사용한다. 따라서 스케쥴러의 로드 밸런싱도 리눅스와 동일하다. 이번 실험은 안드로이드 플랫폼에서 코어간 로드 밸런싱이 어떤 형태로 일어나는지 알아보았다. 우선 뮤직 플레이어로 음악을 플레이 시킨 상태에서 브라우져를 실행하여 무작위로 브라우징을 하면서 1초 간격으로 top 명령어를 실행하여 프로세스들이 실행되는 코어 현황을 500초간 확인 하는 실험을 5회 반복하였다. 브라우저는 이 전과 같이 2개의 프로세스 및 쓰래드

(com.android.chrome, com.android.chrome:sandboxedprocess)를 통해 실행되었고, 음악 플레이는 미디어 서버(/system/bin/mediaserver)를 통해 실행되었다. 500초간 평균적으로 브라우저는 약 60%의 CPU 사용률을 보여 주었다 이중 com.android.chrome는 약 35% com.android.chrome: sandboxedprocess 는 약 25%를 차지하였다. 그리고 미디어 서버는 약 4~5%의 사용률을 기록하였다. 이 기간 동안 com.android.chrome 프로세스는 약 170번, com.android.chrome:sandboxedprocess는 약 195번, 미디어 서버는 120번 정도 서로 다른 코어로 이동한 것으로 확인되었다. 심지어 아래 그림 4.2에서 보는 바와 같이 실행 중인 메인 프로세스 3개가 모두 동일한 코어에서 실행되는 경우도 65회 이상이었다.

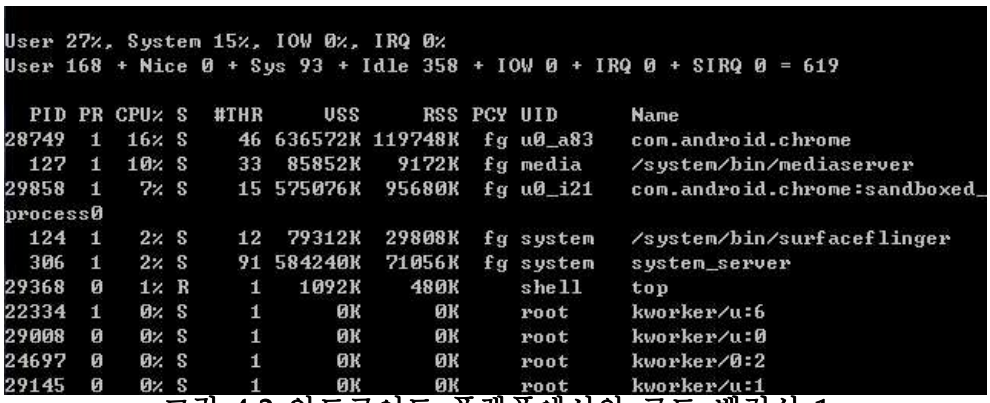


그림 4-2 안드로이드 플랫폼에서의 로드 밸런싱 1

다음으로 동일하게 뮤직 플레이어로 음악을 플레이한 상태에서 안드로이드 Quadrant 벤치마크 애플리케이션을 실행하였는 실험을 실시하였다. Quadrant 벤치마크 애플리케이션은 CPU, Memory, I/O, 2D & 3D Graphic 벤치마크 테스트를 동시에 실시하는 애플리케이션으로 CPU 테스트를 진행 중에는 CPU 사용 현황이 90% 이상으로 나타난다. 테스트는 약 60초간 진행이 되는데, 이 기간 동안에도 Quadrant 애플리케이션의 프로세스인 com.aurorasoftworks.quadrant.ui.stnadard 는 약 20회 이

상 실행 코어를 변경하였다. 그리고 아래 그림 4.2와 같이 메인으로 실행되는 2개 프로세스가 동일한 코어에서 실행되는 것도 15회 이상 발견되었다.

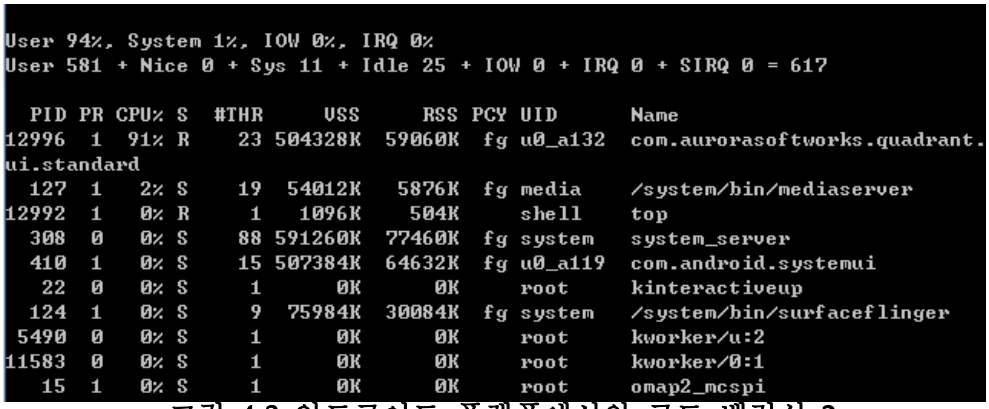


그림 4-3 안드로이드 플랫폼에서의 로드 밸런싱 2

이를 통하여 안드로이드 플랫폼에서는 로드 밸런싱이 리눅스와 동일하게 균형 상태를 이루어도 주기적으로 계속 실행이 되고, 코어를 변경하는 태스크의 선택도 해당 태스크의 특성이 반영되지 않은 채 무작위로 선택되어 진다는 것을 확인할 수 있었다.

4.3 문제 정의

안드로이드 플랫폼의 애플리케이션은 사용자들에게 보여지는 UI를 나타내는 여러 개의 Activity 들과 추가의 작업을 하는 서비스 쓰래드들로 구성된다. 대부분 한 개의 프로세스로 실행되는 경우가 많으며, 이전 실험의 크롬 브라우저와 같이 여러개의 프로세스로 나누어 실행시킬 수도 있다. 개별 Activity의 생명 주기는 아래 그림 4.3과 같으며, onCreate(), onStart(), onResume(), onPause(), onStop(), onDestroy() 를 반복해서 실

행하게 되는데, 이 중 사용자에게 보여지는 시기는 onResume()부터 onPause()가 발생하기 이전 시점까지이며, 그 이외에는 백그라운드 가게 되어 사용자에게 전혀 보이지 않거나 멈춘 상태가 된다. 사용자에게 보이지 않는 백그라운드 상태의 Activity는 CPU를 전혀 점유하지 않으며, 단지 서비스와 같은 형태를 이용하여 추가 작업만을 할 수 있다. 이전 실험에서 확인한 바와 같이 백그라운드에서 실행되는 서비스의 CPU 점유율은 5~10% 미만으로 극히 미미하다.

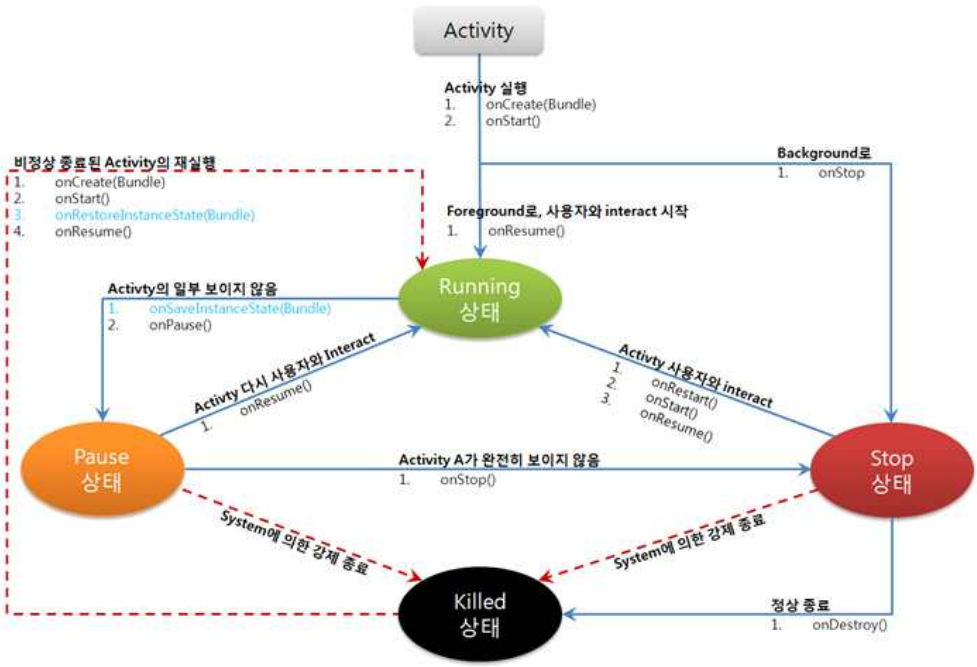


그림 4-4 안드로이드 플랫폼의 Activity 생명 주기

그런데 이전 실험에서 확인한 바와 같이 안드로이드 플랫폼의 스케쥴러는 멀티 코어 간 로드 밸런싱을 위하여 실행 중인 프로세스들이 구동되는 코어를 계속해서 변경을 하고 있으며, 다른 코어로 옮길 태스크의 선

택에 있어서도 해당 태스크의 특성을 고려하지 않아서, 많은 로드가 한 개 코어에 집중되는 경우도 많이 발생하였다.

따라서 실행되는 코어들 간에 성능 간섭이 발생할 수 있으며, 계속해서 코어를 변경함으로써 발생하는 오버헤드와 Processor Affinity 가 무시되는 현상도 발생하여 애플리케이션의 성능 저하를 유발할 수 있다.

제 5 장 Cgroup을 이용한 안드로이드 플랫폼 로드 밸런싱 개선

본 장에서는 cgroup을 이용한 안드로이드의 로드 밸런싱 개선과 전체 시스템에 대해 기술한다.

4장에서 안드로이드 플랫폼에서는 전면에서 실행 중인 1~2개의 프로세스에 CPU 점유가 집중한다고 하였다. 그리고 일반적인 사용자가 사용하는 애플리케이션 중에 CPU 점유도가 50% 이상인 것은 브라우저와 게임 정도이다. 따라서 이와 같은 몇 개 애플리케이션에 CPU 자원을 보장해 준다면 사용자의 체험 성능이 상승할 것으로 판단된다. 쿼드 코어 이상의 시스템에서는 CPU 점유도가 높은 애플리케이션에 개별 코어를 할당할 수 있음으로 두 개 이상의 애플리케이션이 동시에 전면에서 실행되는 시나리오에서도 실행에 무리가 없을 것으로 판단된다.

이에 이 논문에서는 CPU 점유도를 기준으로 30% 미만, 30~50% 사이, 50% 이상의 3가지 태스크로 분류를 하였다. 이 분류에 따라 50% 이상인 태스크들에 대해서는 cgroup의 cpuset 서브시스템을 이용하여 특정 코어를 할당하고, 30~50% 사이의 태스크들에 대해서는 cgroup의 cpu 서브시스템을 이용하여 실행 코어의 50%를 보장하도록 하였다.

전체 시스템은 크게 태스크들의 CPU 점유도를 체크하고 최대치를 기록하는 부분과, 해당 값을 기준으로 cgroup 하이어라키를 구성하는 두 개 부분으로 이루어 진다.

CPU 점유도 체크는 리눅스의 schedule() 함수 안에서 실시한다. current 태스크의 컨텍스트 스위칭이 발생하는 시점에 current 태스크의 CPU 점유도를 체크하고, 이전 최대치 값과 비교하여 높은 경우 변경하

게 된다. task_struck 구조체 내부에 해당 태스크의 CPU 점유도 최대치를 기록하는 필드를 추가하였다.

모바일 디바이스는 특성상 전력 소비에 제한이 많기 때문에 매니코어 시스템에서도 모든 코어를 전부 사용하지 않는 경우가 많다. 대기 상태에서는 1개 코어만을 사용하고, 시스템의 로드를 판단하여 다른 코어들을 차례로 enable 하고, 다시 시스의 로드가 줄어들면 코어를 disable 하는 형태이다. 최신의 모바일용 쿼드코어인 엔비디아社의 테그라 3 는 4개의 기본 코어와 이에 비해 전력 소모가 적고 성능이 조금 낮은 컴패년 코어 1개로 구성되어 있다. 대기 상태에서는 컴패년 코어만이 enable 되어 시스템 전체 전력 소모를 줄이고, 이후 사용자가 스마트폰을 사용하여 시스템 로드가 상승하게 되면, 로드에 따라 기본 코어들을 enable 한다.

이러한 기능은 리눅스의 hotplug를 이용하여 구현되는 경우가 많으며, hotplug는 workqueue에 해당 하드웨어 디바이스를 en/disable 하도록 하고 있다. 따라서 우리 시스템에서 cgroup 하이어라키를 구성하는 부분도 현재 enable 되어 있는 CPU를 확인한 후 실시할 수 있도록 workqueue로 구현하였다.

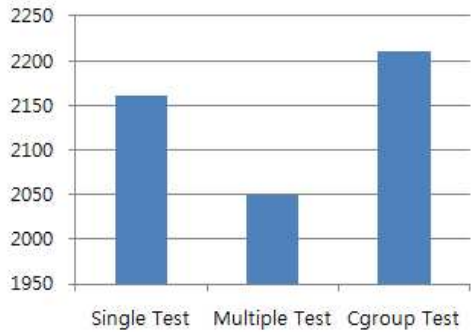
제 6 장 실험 및 검증

본 장은 제안된 Cgroup을 통한 안드로이드 플랫폼 로드 밸런싱 개선 기법을 검증하기 위한 실험 및 결과이다.

실험에 사용한 기기는 이전과 마찬가지로 Galaxy Nexus 모델을 이용하고, 안드로이드 플랫폼 버전은 4.1.1, 리눅스 커널 버전은 3.0.31 이고, CPU는 ARMv7 1200MHz 2 Core 를 사용하였다. 실험에 사용한 벤치마크 프로그램은 안드로이드 Quadrant 벤치마크 애플리케이션과, 브라우저의 JavaScript 성능 벤치마크 테스트인 SunSpider를 이용하였다. 실험은 최초 Quadrant 벤치마크 애플리케이션만 실행시켜서 테스트를 진행하는 경우와, 백그라운드에서 뮤직 플레이어를 통해 음악을 플레이하고, Google 마켓 어플에서 50MByte 이상급의 애플리케이션을 다운로드 받도록 실행 시킨 후에 테스트를 진행하는 경우, 그리고 동일한 케이스에 대해, Cgroup으로 Quadrant 애플리케이션만을 개별 CPU에 할당한 후 테스트를 진행하는 3가지 실험을 하였다. 브라우저를 통한 SunSpider 테스트에 대해서도 동일하게 3가지 실험을 실시하였다.

결과는 동일한 실험을 5번씩 진행하여 평균을 구하였다. 아래 표 6.1은 Quadrant 애플리케이션을 이용한 테스트 결과로서 Quadrant 애플리케이션 내부 기준에 따라 점수를 부여하는 것임으로 점수가 높을 수록 성능이 높다. CPU, Memory, I/O, Graphic 테스트를 동시에 실시하지만, CPU 테스트 만을 분리하여 실행할 수 없어, 모두 실시한 결과를 반영하였다. 표 6.2는 SunSpider를 이용한 테스트 결과로, SunSpider에서 마련한 JavaScript 함수를 모두 실행하는데 소요되는 시간을 ms 단위로 제공

한다. Single Test는 각각의 벤치마크 애플리케이션 만을 실행시킨 경우이고, Multiple Test는 백그라운드에서 다른 애플리케이션을 동시에 실행시킨 경우, 마지막 Cgroup 테스트는 Cgroup으로 테스트 애플리케이션에 개별 CPU를 부여한 케이스이다.



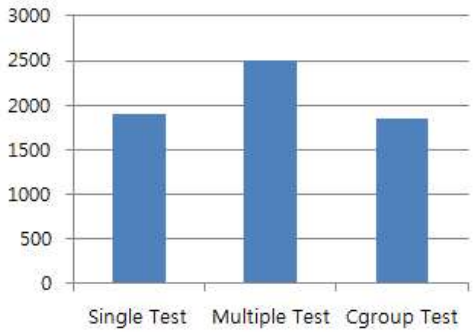


표 6-1 Quadrant 테스트 결과 **표 6-2 SunSpider 테스트 결과**

실험을 통해 Cgroup으로 테스트 애플리케이션에 개별 CPU를 할당한 케이스에서 Quandrant 테스트에서는 Single Test 대비 약 2%, Multiple Test 대비 약 7%의 성능 개선이 있었고, SunSpider 테스트에서는 Single Test 대비 약 2%, Multiple Test 대비 약 25%의 성능 개선이 있었다. Single Test 대비하여 약 2%의 개선은 테스트 애플리케이션들이 로드 밸런싱을 위한 실행 코어 변경으로 인한 오버헤드가 줄었기 때문으로 판단된다.

아래의 그림들은 Quadrant 테스트와 SunSpider 테스트 중에서 실례로 결과 화면을 캡춰한 사진이다.

그림 6-1 Quadrant 테스트 결과 화면

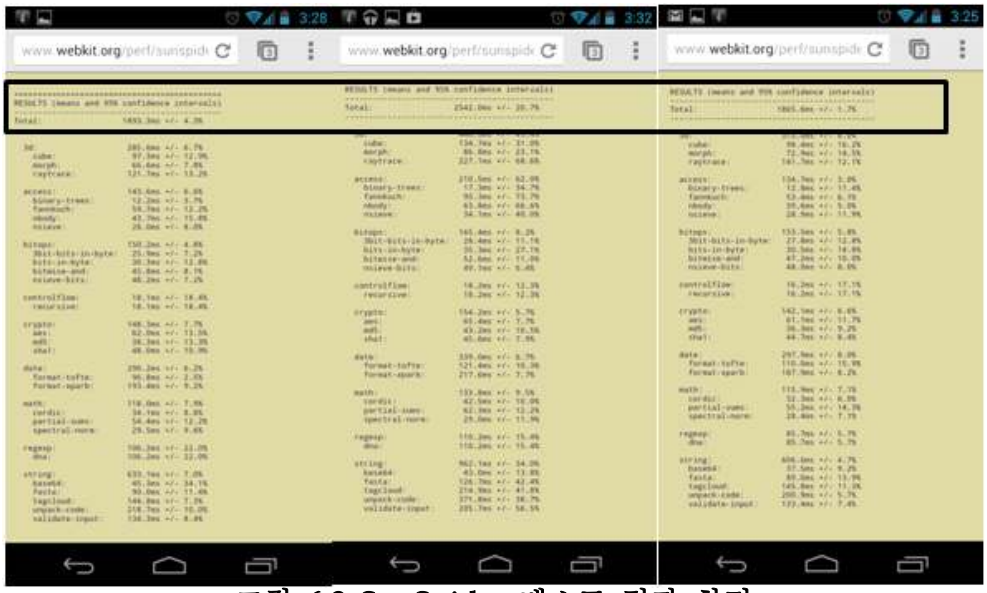


그림 6-2 SunSpider 테스트 결과 화면

제 7 장 결론

본 논문에서는 안드로이드 플랫폼에서 애플리케이션들의 CPU 사용 현황과 운영체제 스케쥴러의 로드 밸런싱 기법을 확인해 보고, 현재 스케쥴러 로드 밸런싱 기법의 문제점을 확인 하였다. 현재의 로드 밸런싱 기법은 로드 밸런싱이 불필요한 시점에서도 계속해서 로드 밸런싱을 시도하여 불필요한 오버헤드를 발생시키고, 로드 밸런싱시에 실행 코어를 변경할 태스크 선정 시에도 태스크의 특성이 고려되지 않아서, 특정 코어에 로드가 집중되는 현상이 자주 발생하였다.

이에 리눅스의 Cgroup 기능을 활용하여, CPU 사용률이 높은 애플리케이션들에 대해서는 별도의 CPU 리소스를 할당함으로써, 다중 애플리케이션 상황에서 CPU 사용률이 높은 애플리케이션이 필요한 CPU를 확보할 수 있도록 하였으며, 실행 중에 불필요한 코어 변동으로 발생하는 오버헤드를 줄이고, 실행 중인 애플리케이션 간의 성능 간섭을 줄일 수 있었다. Quadrunt 벤치마크 애플리케이션을 이용한 실험을 통해 평균 약 15% 이상의 성능 개선을 확인 할 수 있었다.

이 논문에서는 Cgroup의 cpu, cpuset 서브 시스템 만을 활용하였으나, 애플리케이션들의 메모리 사용 특성이나, 블록 장치, 네트웍 활용 특성 등에 대해서 그룹화가 가능한 특성을 도출한다면 memory, blkio, net_cls, 등의 서브 시스템을 확용하여 추가의 성능 개선이 가능할 것으로 판단된다.

참고 문헌

[1] http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt
http://www.kernel.org/doc/Documentation/cgroups/cpusets.txt
http://www.kernel.org/doc/Documentation/cgroups/memory.txt

[2] Lisa A. Torrey, Joyce Coleman, Barton P. Miller, “Comparing Interactive Scheduling in Linux” University of Wisconsin.

[3] Chandandeep Singh Pabla, “Completely Fair Scheduler” Linux Journal, vol 2009, August 2009.

[4] http://sites.google.com/site/io/anatomy--physiology-of-an-android

[5] Joseph T. Meehean, "Towards Transparent CPU Scheduling" Doctoral Dissertation, University of Wisconsin, August 2011.

[6] Aas, J. "Understanding the Linux 2.6. 8.1 CPU scheduler", Retrieved Oct, 2005, 16, pp. 1-38.

[7] Robert Love, "Linux Kernel Development 3 Process Management, 4 Process Scheduling", 3rd Edition, 2010

[8] Martin Prpic, Rudiger Landmann, Douglas Silas. "Red Hat

Enterprise Linux 6. Resource Management Guide", Red Hat, Inc, 2011

[9] http://www.zdnet.com/blog/hardware/intel-android-not-ready-for-multi-core-cpus/20746

[10]http://www.linux-magazine.com/Online/News/Con-Kolivas-Introduces-New-BFS-Scheduler

[11] 고현철, 유형목 “안드로이드의 모든 것 분석과 포팅”, 한빛 미디어 2011년 6월

[12] http://.www.nvidia.com/object/tegra-3-processor.html

[13] Juan A. Colmenares, Sarah Bird, Henry Cook, Paul Pearce, David Zhu, Jonh Shalf, Steven Hofmeyr, Krste Asanovic, John Kubiatowicz "Resource Management in the Tessellation Manycore OS", USENIX, 2010

[14] http://tessellation.cs.berkeley.edu/#Home;;id.2

[15] http://developer.android.com

[16] OmidReza Kiyarazm, M-Hossein Moeinzad도, Sarah Sharifian-R, "A new method for scheduling load balancing in multi-processor systems based on PSO", ISMS, 2011

ABSTRACT

Manycore and GPU systems are common in recent smartphones as PC. We need to increase parallelism of working tasks to enhance the performance of manycore and GPU systems. To increase parallelism, application should be programed to work in parallel and OS of the system should assign system resources efficiently among competing tasks. So load-balancing functionality of scheduler became important for that purpose.

Android platform uses Linux kernel and so it uses Linux CFS scheduler. CFS scheduler uses pair-wise balancing and somewhat blind selection to choose a task to move for load-balancing. We found that CFS's these load-balancing methods are not good for mobile devices' performance because it lead to unnecessary overhead for moving tasks among cores continuously and cause performance interference among tasks in a core.

In this paper, we use used cgroup functionality of Linux to give task's characteristics like CPU intensiveness to scheduler. With that, scheduler could assign proper amount of resource to the task and that performance of foreground application of Android could be enhanced. We found about 15% performance enhancement with this method. And we could extend this way to include other characteristics of task like memory, block I/O, network bandwidth and etc.

Keywords: load-balancing, cgroup, Android platform, Linux, task scheduling

Student Number: 2011-20941