(KCC2012 우수논문)

# 안드로이드 기반 스마트폰의 사용자 응답성 향상을 위한 프레임워크 지원 우선순위 부스트 기법과 로드 밸런싱 기법*

# (Framework-assisted Priority boosting and Load balancing for Improving Interactivity of Android Smartphones)

손 용 석† 허 승 주‡ 유 종 훈§ 홍 성 수**
(Yongseok Son) (Sungju Huh) (Jonghun Yoo) (Seongsoo Hong)

**요 약** 최근 안드로이드 플랫폼을 탑재한 스마트폰이 널리 보급되면서 안드로이드 플랫폼에 대한 관심은 더욱 커지고 있다. 하지만 안드로이드 스마트폰은 종종 양질의 사용자 응답성을 제공하지 못하는 것으로 알려져 있다. 이는 안드로이드 상에서 대화형 태스크가 다른 태스크와 구별되지 않고 동일한 우선순위로 스케줄링 되기 때문에 사용자 입력을 처리하는 동안 여러 번의 선점을 당해 긴 응답시간을 초래할 수 있기 때문이다. 이 논문은 안드로이드 스마트폰의 사용자 응답성 향상을 위해 프레임워크 지원 우선순위 부스트 기법과 로드 밸런싱 기법을 제시한다. 프레임워크 지원 우선순위기법은 프레임워크 레벨에서 대화형 태스크를 식별하고 이를 커널에게 전달하며, 커널 레벨에서는 식별된 태스크의 우선순위를 선별적으로 부스트 시킴으로써 사용자 입력을 처리할 만큼 충분한 시간을 보장해 준다. 로드 밸런싱 기법은 부스트 된 태스크를 여전히 방해하는 태스크들을 다른 실행 큐로 이주시킴으로써, 대화형 태스크의 응답시간을 최소화 한다. 실험 결과 대화형 태스크의 응답시간이 우선순위 부스트 기법을 통해서는 기존 시스템보다 최대 22% 단축됨을 보였고 로드 밸런싱 기법을 통해서는 최대 43.31% 단축됨을 보여 제안된 기법의 효용성을 입증하였다.

**키워드** : 사용자 응답성, 안드로이드 스마트폰, 리눅스, 태스크 스케줄링

***Abstract*** Smartphones on Android platform recently have been come into wide use. However, it is often reported that Android smartphones cannot provide enough interactivity because Android cannot distinguish interactive tasks and non-interactive tasks and they are scheduled with the same priority and preempted. Thus, it occurs poor response time. This paper proposes a framework assisted priority boosting and load balancing for improving interactivity of Android smartphones. The framework assisted priority boosting technique distinguishes the interactive task in the framework level and send the task ID to the kernel. The kernel ensures enough time to process user input by boosting the priority of distinguished task. The load balancing technique minimizes response time of boosted task by migrating tasks disturbing boosted task to other run-queue. The experiment results demonstrate the priority boosting technique reduces response time up to 22% and the load balancing technique along with priority boosting reduces response time up to 43.31% compared to the previous techniques.

**Key words:** Interactivity, Android smartphones, Linux, Task scheduling

* 본 논문은 KCC2012에서 '안드로이드 기반 스마트폰의 사용자 응답성 향상을 위한 프레임워크 지원 우선순위 부스트 기법'의 제목으로 발표된 논문을 확장한 것임
†비 회 원: 서울대학교 융합과학기술대학원 지능형 융합시스템학과
ysson@redwood.snu.ac.kr
‡학생회원: 서울대학교 융합과학기술대학원 지능형 융합시스템학과
sjhuh@redwood.snu.ac.kr
§학생회원: 서울대학교 전기컴퓨터공학부
jhyoo@redwood.snu.ac.kr
**종신회원: 서울대학교 전기컴퓨터공학부 교수
sshong@redwood.snu.ac.kr

논문접수 : 2012년 X월 X일
심사완료 : 2012년 X월 X일


정보과학회논문지: XXXX 제XX권 제X호(2012.XX)

## 1. 서 론

안드로이드는 현재 가장 널리 사용되는 스마트폰 소프트웨어 플랫폼임에도 불구하고 여전히 사용자 응답성 측면에서 부정적인 평가가 보고되고 있다 [1]. 대화형 응용태스크의 성능은 사용자 입력을 처리하여 화면에 그 결과를 표시하는 응답시간으로 정량화될 수 있다. 이 시간 동안 커널은 입력 장치로부터 발생한 인터럽트를 처리하고, 응용 태스크는 이를 전달받아 최종적으로 출력 장치에 결과를 표시한다. 스마트폰 사용자는 이 과정이 매우 짧은 시간 내에 완료될 것을 예상한다.

그런데 수많은 태스크가 동시에 수행되는 안드로이드 스마트폰에서 짧은 응답 시간을 보장하는 것은 매우 어려운 문제이다. 대화형 태스크는 CPU를 차지하기 위해 통상 수십 개의 다른 태스크들과

경쟁해야 하며 이로 인한 간섭이 저조한 사용자 응답성의 주요 원인으로 지적되어 왔다.

안드로이드에서 태스크들은 리눅스의 기본 스케줄러인 Completely Fair Scheduler(CFS)에 의해 스케줄링 된다. CFS는 주목적은 각 태스크에게 공정한 CPU 시간을 할당하는 것이다. 이를 위해 CFS는 각 태스크에게 가중치에 비례하는 타임 슬라이스(time slice)를 할당하고 이 시간 동안 선점 없이 수행될 수 있도록 보장한다 [2].

CFS는 다수 사용자가 컴퓨팅 자원을 공평하게 나누어 사용해야 하는 서버 환경에서 특히 유용하다. 그러나 CFS는 사용자 응답성 측면에서는 만족스러운 성능을 발휘하지 못하는 것으로 알려져 있다 [3][4]. 이는 CFS가 대화형 태스크를 식별하지 않아 대화형 태스크에게 다른 태스크에 비해 CPU 자원을 보다 유리하게 할당해 주는 조치를 취할 수 없기 때문이다. 그 결과 대화형 태스크는 사용자 입력 처리 중 타임 슬라이스를 소진하게 될 때마다 다른 태스크에게 선점 당하여 긴 응답시간을 초래한다.

본 논문에서는 이 문제를 해결하기 위해 프레임워크 지원 우선순위 부스트 기법과 로드 밸런싱 기법을 제시한다. 우선순위 부스트 기법은 프레임워크 레벨에서 대화형 태스크를 식별하고 이를 커널에게 전달하며, 커널 레벨에서 식별된 태스크의 우선순위를 임시적으로 높여 줌으로써 사용자 입력을 처리하기 위해 충분한 타임 슬라이스를 할당 받을 수 있도록 한다. 로드 밸런싱 기법은 부스트 된 태스크를 여전히 방해하는 태스크들을 다른 실행 큐로 이주시킴으로써 대화형 태스크의 응답시간을 최소화 한다. 우리는 제안된 기법들을 안드로이드 Froyo 기반 개발 보드 상에 구현하였다. 실험 결과 우선순위 부스트 기법을 통해서는 대화형 태스크의 응답시간이 기존 시스템에 비해 최대 22% 단축됨을 보였고, 로드 밸런싱 기법을 통해서는 최대 43.31% 단축됨을 보여 제안된 기법들의 효용성이 입증되었다.

이 논문의 나머지 부분은 다음과 같이 구성된다. 2장에서는 관련 연구를 소개하고 3장에서는 CFS를 분석한다. 4장은 안드로이드의 입력 이벤트 처리과정을 분석하고 5장에서 긴 선점 지연시간 문제를 정의한다. 6장에서 제안된 기법들을 설명하고 7장에서 실험 결과를 보이며 8장에서 논문의 결론을 맺는다.

## 2. 관련 연구

안드로이드 시스템의 사용자 응답성을 향상시키기 위한 여러 선행연구들이 있다. 선행 연구는 두 가지로 구분 될 수 있다. (1)커널 레벨에서 사용자 응답성 개선 (2)프레임워크 레벨에서 사용자 응답성 개선

리눅스 커널 개발자들은 태스크 스케줄링 알고리즘을 개선시켜 리눅스의 사용자 응답성을 향상시키려 노력했다. 리눅스 커널 2.6.23 이전에, 리눅스는 O(1) 스케줄러를 태스크 스케줄러로 사용하였다 [5]. O(1) 스케줄러는 태스크들이 I/O 기반 또는 CPU기반 태스크인지를 확률과 경험에 의한 방법으로 구별하려고 시도하였다. O(1) 스케줄러는 태스크가 구별되면, I/O기반 태스크들의 우선순위를 높여준다. O(1) 스케줄러는 종종 CFS보다 더 나은 사용자 응답성을 보인다 [2][3]. 불행히도, 이러한 확률과 경험에 의한 방법은 태스크를 잘 못 구별 할 수가 있다. 그 결과 이는 특정 워크로드(workload)에서 기아현상과 공정하지 못한 CPU 할당을 초래한다 [6].

Torrey et al.는 O(1)스케줄러의 사용자 응답성을 향상시키기 위해서 MLFQ 스케줄러를 리눅스 2.6 커널에 구현하였다 [7]. O(1)스케줄러와의 차이점은 확률과 경험에 의한 방법을 제거하고 각 태스크의 CPU 버스트 타임을 이용하여, I/O기반 태스크와 CPU기반 태스크를 구별한 점이다. 실험 결과 사용자 응답성은 향상되었지만, MLFQ의 특성상 우선순위가 높은 태스크가 짧은 타임 슬라이스를 할당 받기 때문에 CPU기반 워크로드에서는 저조한 처리량을 보인다.

Kolivas는 CFS를 대체할 수 있는 스케줄러로 Brain Fuck Scheduler(BFS)을 제안하였다. BFS의 주목적은 확률과 경험에 의한 방법을 사용 하지 않고 더 단순한 런타임 알고리즘을 제안함으로써, 데스크톱 사용자 응답성을 개선시키는 것이다. 태스크가 블로킹(blocking)되었을 때, 태스크의 가상 데드라인과 남은 타임 슬라이스를 유지하여 해당 태스크가 다시 스케줄링 될 때, 더 높은 우선순위를 보장해줌으로써 사용자 응답성을 향상시켰다. BFS는 CFS보다 사용자 응답성 측면에서 보다 나은 성능을 보여주었다 [8]. 또한, 이러한 사용자 응답성 향상은 리눅스 매거진 [9]에 의해 보고되었고, 이는 안드로이드 Éclair 버전에 추가되었다. 그러나, 이러한 개선은 사용자가 인지할 수 있는 성능 차이를 보여주지 못해 안드로이드 Froyo버전에서는 제외되었다. 또한 BFS는 알고리즘의 단순화를 위해 전역화된 실행 큐를 사용하기 때문에 CPU 개수가 증가하면 락 경쟁(lock contention)에 따른 성능 오버헤드를 발생한다.

안드로이드는 또한 다음과 같이 사용자 응답성을 향상시키기 위한 다양한 기법들을 이용하였다. 안드로이드는 응용태스크를 빠르게 실행시킴으로써, 사용자 응답성을 개선시켰다[10]. 안드로이드 응용 태스크들은 자신의 달빅 가상 머신 인스턴스상에서 실행된다. 그러나 가상 머신 인스턴스의 cold-start시간은 상당히 저조한 응답을 초래한다. 안드로이드는 이를 개선시키기 위해, 응용들이 자주 사용하는 라이브러리 클래스들이 미리 초기화된 Zygote를 이용하였다. 응용 태스크는 Zygote로부터 생성됨으로써, 더 빠르게 실행될 수 있다.

안드로이드는 터치스크린 드로잉(drawing)시간을 단축시키기 위해서, 분리된 렌더링 작업을 수행한다

[11]. 안드로이드 스크린은 surface라 불리는 분리된 구역으로 나누어져 있으며 갱신된 surface만 새롭게 렌더링 한다. surfaceflinger 라는 안드로이드 서비스는 여러 개의 surface들을 조합해 프레임 버퍼 장치로 전송한다. 따라서 전체 렌더링 시간은 줄어들고 사용자는 향상된 사용자 응답성을 인지할 수 있다.

최근 출시된 안드로이드 jelly bean은 Wait for vsync 기술과 삼중 버퍼링 기술을 이용하여 일정하고 빠른 프레임률을 보장해줌으로써 사용자 응답성을 향상시켰다 [12]. 이는 드로잉시간뿐만 아니라 스크롤링, 페이징, 애니메이션과 같은 동작들도 사용자가 좀 더 부드럽게 느끼도록 개선시켰다.

본 논문에서 제시한 기법들은 프레임워크 지원을 받아 태스크 스케줄링을 개선시켜 사용자 응답성을 개선시켰으며, 다음과 같이 기존 연구들의 문제점들을 해결하였다.

① O(1) 스케줄러의 기아현상 해결: 제시된 기법들은 사용자가 직접 요청한 입력 처리 응용의 우선순위를 높여주기 때문에 태스크를 잘못 구별하여 발생되는 기아현상을 발생시키지 않는다.

② MLFQ 스케줄러의 저조한 처리량 해결: 제시된 기법들은 MLFQ특성을 따르지 않고 기존 리눅스 스케줄러에서 사용하는 우선순위와 타임 슬라이스를 그대로 이용함으로써 CPU 위주 워크로드에서도 저조한 처리량을 발생시키지 않는다.

③ BFS 스케줄러의 성능 오버헤드 해결: 제시된 기법들은 전역 실행 큐를 사용하지 않고 각 코어당 하나의 실행 큐를 가짐으로써 코어 개수 증가에 따른 락 경쟁을 유발하지 않는다.

## 3. CFS의 분석

CFS는 대칭 멀티코어 프로세서를 위한 스케줄러이다. 이는 각각의 코어에 독립적인 실행 큐를 유지하고 독립적으로 스케줄링 한다. CFS의 주목적은 각 태스크에게 가중치에 비례하는 CPU시간을 할당하는 것이다. 태스크의 가중치는 태스크의 가중치는 nice값에 의해 결정된다.

Nice값의 범위는 -20(우선순위: 100)에서부터 19(우선순위: 140)까지이며 낮은 nice값은 높은 가중치를 나타낸다. CFS는 가상 런타임을 이용하여 그 목적을 성공적으로 달성한다. 한 태스크의 가상 런타임은 해당 태스크의 축적된 런타임이 가중치의 역으로 스케일 되는 값으로 정의된다. $W(\tau_i)$는 태스크 $\tau_i$의 가중치를 나타내며 $\omega_0$는 nice값 0를 의미한다. $A(\tau_i, t)$은 태스크 $\tau_i$가 $[0, t)$시간 동안에 받은 CPU time을 의미한다. time t 에 태스크 $\tau_i$의 가상 런타임은 다음과 같다.

$$VR(\tau_i, t) = \frac{\omega_0}{W(\tau_i)} \times A(\tau_i, t) \quad (1)$$

만일 주어진 시간에 모든 태스크들의 가상 런타임이 동일하다면, CFS는 완벽한 공정성을 보장한다. CFS는 이를 달성하기 위해서 스케줄링 결정시점에서 가장 작은 가상 런타임을 가진 태스크를 스케줄링 한다.

또한 CFS는 멀티코어 시스템에서 실행 큐들 사이의 로드를 균등하게 배분하기 위해서 가중치 기반 로드 밸런싱을 수행한다. CFS는 이를 위해 실행 큐 마다 로드 값을 유지한다. 실행 큐 $Q_k$의 로드는 $Q_k$ 안에 있는 태스크들의 가중치를 모두 합한 값이며 다음과 같이 정의된다.

$$L_k = \sum_{\tau_i \epsilon S_k} W(\tau_i) \quad (2)$$

여기서 $S_k$는 $Q_k$안에 속해 있는 태스크들의 집합이다.

CFS의 로드 밸런싱은 다음과 같은 시점에 수행된다. (1) 새로운 태스크가 생성되거나 수면 또는 대기중인 태스크가 깨어날 때, (2) 태스크들의 수행이 모두 종료되어 실행 큐가 유휴상태일 때, (3) 각 CPU에서 시스템이 정의한 일정 주기마다.

첫 번째 경우, CFS는 가장 작은 로드를 가진 실행 큐를 찾아 해당 태스크를 실행 큐에 삽입한다. 두 번째와 세 번째 경우, CFS는 실행 큐간의 로드 불균형을 방지하기 위해서 로드 가장 큰 실행 큐($Q_{busiest}$)에서 로드 밸런싱을 발생시킨 CPU의 실행 큐($Q_k$)로 태스크를 이주시킨다.

CFS는 다수의 태스크들을 이주 시킬 수 있으며 이주된 결과가 더 큰 불균형을 일으키는 것을 방지하기 위해 다음과 같이 로드의 양을 정의한다.

$$L_{imbal} = \min(L_{busiest} - L_{avg}, L_{busiest} - L_{busiest\_load\_per\_task}, L_{avg} - L_k) \quad (3)$$

여기서 $L_{busiest}$는 $Q_{busiest}$의 로드이며 $L_{avg}$는 전체 시스템의 평균 로드 값이다. $L_{busiest_load_per_task}$는 $L_{busiest}$ 값에 $Q_{busiest}$에 속한 태스크의 수로 나눈 값이다. CFS는 다음과 같은 조건일 때 태스크를 이주시킨다.

$$L_{imbal} \geq W(\tau_i)_{\tau_i \epsilon S_{busiest}}/2 \quad (4)$$

여기서 $S_{busiest}$는 $Q_{busiest}$안에 태스크들의 집합이다. 또한, 태스크 $\tau_i$는 다음과 같은 세가지 조건에 만족해야 다른 CPU로 이주될 수 있다.

① $\tau_i$가 실행 중이 아닐 경우

② $\tau_i$가 이주 될 CPU에서 수행이 허용될 경우

③ $\tau_i$가 현재 CPU에서 캐시 hot이 아닐 경우

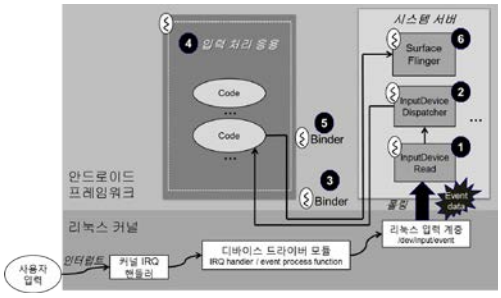


그림 1. 안드로이드 입력 이벤트 전달 경로.

## 4. 안드로이드의 입력 이벤트 처리 과정 분석

이 장에서는 제안된 기법의 이해를 돕기 위해 안드로이드에서 사용자가 발생시키는 입력 이벤트가 처리되는 과정을 분석한다. 그림 1은 안드로이드의 입력 이벤트 처리를 위해 수행되는 리눅스 커널, 시스템 서버, 그리고 응용간의 상호 작용을 보인다. 시스템 서버는 응용에게 각종 서비스를 제공하기 위해 수행되는 프로세스로서 각 서비스마다 하나의 스레드를 생성한다. 이들 중 사용자의 입력 이벤트 처리를 담당하는 스레드는 InputDeivceRead와 InputDeivceDispatcher이며, 스마트폰의 스크린에 출력을 담당하는 스레드는 SurfaceFlinger이다. 다음은 안드로이드에서 사용자 입력 이벤트를 처리하는 네 단계를 보인다.

① 커널은 입력 장치가 발생시키는 인터럽트를 디바이스 드라이버로 전달하고 이는 다시 리눅스 입력 계층을 통해 안드로이드의 시스템 서버로 해당 이벤트를 전달한다.
② 시스템 서버의 InputDeviceRead는 커널이 발생시키는 모든 타입의 입력 이벤트를 큐에 저장한다. InputDeviceDispather는 큐에 저장된 이벤트를 타입 별로 분류한 후, 이벤트를 기다리고 있는 응용 태스크에게 전달한다. 이 때 시스템 서버와 응용 간 통신에 Binder IPC가 사용된다.
③ 응용 태스크는 이벤트를 처리하고 화면에 표시할 이미지를 우선적으로 메모리에 저장한 후, 그 결과를 SurfaceFlinger에게 Binder IPC를 사용하여 전달한다.
④ SurfaceFlinger는 전달받은 데이터를 커널의 프레임 버퍼에 저장하고, 그 결과 사용자는 입력에 대한 응답을 확인할 수 있다.

## 5. 문제 정의

응답 시간이란 스마트폰이 입력 장치에 이벤트가 발생한 시점부터 화면에 결과가 표시되기까지 걸리는 총 소요 시간으로 정의된다. 임의의 입력 처리 응용 태스크 $\tau$의 응답 시간 $RT(\tau)$는 다음과 같이 세 개의 구성 요소로 이루어진다.

$$RT(\tau) = T_S + T_P + T_E \quad (5)$$

이 때 $T_S$, $T_P$, $T_E$는 각각 스케줄링 지연시간, 선점 지연시간 그리고 태스크의 수행 시간을 나타낸다. 스케줄링 지연시간은 태스크가 wake-up되는 시점부터 CPU에 할당되어 실제로 첫 번째 명령어가 수행되는 시점까지의 시간이다. 선점 지연시간은 태스크가 입력 처리 도중 다른 태스크들에 의해 선점되는 총 시간이다. 한편 수행시간 $T_E$는 InputDeviceRead를 비롯한 시스템 서버 태스크들과 $\tau$의 수행시간을 합한 것이다.

이 중 $T_S$는 임의의 태스크에게 할당될 수 있는 최대 타임 슬라이스의 길이로 바운드되며 $T_E$는 입력 처리 응용 태스크의 구현에 의존한다. 반면, $T_P$는 $\tau$가 자신에게 부여된 타임 슬라이스를 소진한 후, 다른 태스크에게 선점 당할 때 마다 증가한다.

본 논문에서는 선점지연시간의 감소를 통해 안드로이드 스마트폰의 사용자 응답성 향상을 달성하고자 한다. 안드로이드에서 높은 선점 지연시간이 발생하는 주된 원인은 태스크 스케줄러인 CFS가 입력 처리 응용 태스크와 다른 태스크들을 구별하지 않기 때문이다. 안드로이드에서 모든 응용 태스크들은 동일한 가중치(nice값 0)를 부여 받는다. 따라서 CFS는 이 태스크들에게 동일한 타임 슬라이스를 할당하고 스케줄링을 수행한다. 불행히도, CFS가 한 태스크에게 부여하는 타임 슬라이스는 입력이벤트를 처리하기에 충분하지 않다. 따라서 입력 처리 도중 여러 번의 선점을 당해 결과적으로 높은 선점 지연시간이 야기된다.

## 6. 프레임워크 지원 우선순위 부스트 기법과 로드 밸런싱 기법

본 장에서는 안드로이드의 선점 지연시간을 줄이기 위한 프레임워크 지원 우선순위 부스트 기법과 로드 밸런싱 기법에 대해 설명한다. 우리의 기법은 크게 두 단계로 이루어진다. 첫째, 프레임워크 레벨에서 입력 처리 응용 태스크가 식별되며, 둘째, 커널 레벨에서 식별된 태스크의 우선순위가 임시로 상향 조정된다.

제안된 기법의 핵심 아이디어는 입력 처리 응용 태스크를 런타임에 효과적으로 식별하고 우선순위를 부스트 하는 것이다. 불행히도 커널 레벨에서 입력 처리 응용 태스크를 구별하는 것은 어려운 일이다. 실제로 리눅스의 O(1) 스케줄러는 확률과 경험에 따른 식별을 시도하였으나 종종 오동작하여 태스크 기아현상 등의 문제가 발생한다고 보고되었다 [5].

우리는 안드로이드 프레임워크에서 사용자 입력에 대한 이벤트가 반드시 InputDeviceDispather를 통해 입력 처리 응용 태스크에게 전달되어야 한다는

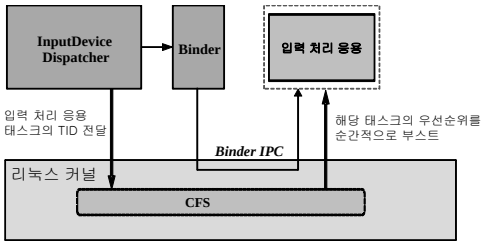


그림 2. 프레임워크 지원 우선순위 부스트 기법

사실에 착안하여 프레임워크 레벨에서 입력 처리 응용 태스크를 식별하고 그 결과를 커널에게 전달한다. 커널은 식별된 태스크의 우선순위는 일정수준 부스트 한다. 결과적으로 입력 처리 응용 태스크에게 긴 타임 슬라이스가 할당된다.

그림 2는 제안된 기법을 보인다. 4장에서 언급했듯이, 사용자 입력에 대한 이벤트는 InputDeviceDispatcher와 binder IPC를 통해 입력 처리 응용 태스크에게 전달된다. 우리는 새로운 시스템 콜을 추가하여 입력 처리 응용 태스크의 ID를 커널에게 전달 하도록 InputDeivceDispatcher를 확장한다.

커널은 프레임워크로부터 입력 처리 응용 태스크의 ID를 전달받아 해당 태스크의 우선순위를 부스트 시킨다. 구체적으로, 입력 처리 응용 태스크가 wake-up 될 때, 커널은 프레임워크로부터 전달된 스레드 ID와 wake-up된 태스크의 스레드 ID를 비교한다. 만일 두 스레드 ID가 일치하면, 커널의 스케줄러인 CFS는 해당 태스크를 입력 처리 응용 태스크로 인식하고, 일시적으로 우선순위를 높여준다. 입력 처리 응용 태스크가 종료되거나 블로킹이 되면, 다시 원래의 우선순위로 낮춰준다.

CFS의 로드 밸런싱은 멀티코어 시스템에서 여러 실행 큐들의 로드를 균등하게 배분한다. 하지만 CFS의 로드 밸런싱은 가중치만을 고려하여 로드를 배분하기 때문에, 입력 처리 응용 태스크가 여전히 긴 실행시간을 가진 태스크에게 계속적으로 방해를 받을 수 있다. 이를 해결하기 위해서, 선별적으로 실행 시간이 긴 태스크를 입력 처리 응용 태스크가 없는 다른 실행 큐로 이주시킴으로써 선점지연시간을 더욱 줄인다.

실행시간이 긴 태스크를 선별하는 방법으로는 가상 런타임이 가장 큰 태스크를 선택하는 것이다. 가상 런타임은 실행시간이 길수록 증가하며 가중치가 낮을수록 증가한다. 따라서 CPU 위주 태스크는 실행시간이 길면서 가중치가 낮기 때문에 가상 런타임이 가장 큰 태스크로 선택되고, 입력 처리 응용 태스크가 없는 다른 실행 큐로 이주된다.

그림 3은 제안된 로드 밸런싱 알고리즘에 대해 상세히 설명한다. 본 알고리즘은 3장에서 설명한 로드 밸런싱 발생 시점 중 세 번째 경우에만 적용된다.

***Algorithm**: migration mechanism for improving interactivity*

```
Migrate(Q_k, Q_busiest)
1  L_imbal ← this value is calculated by (3)
2  if (A input event processing app is runnable or running)
3     then τ_maximum_vr ← NULL
4        do until imbalance > 0
5           for each τ ∈ Q_busiest
6              if VR(τ) > VR(τ_maximum_vr)
7                 then τ_maximum_vr ← τ
8              endif
9           endfor
10          if W(τ_maximum_vr)/2 ≥ imbalance
11             then migrate τ_maximum_vr from Q_busiest to Q_k
12          endif
13          imbalance ← imbalance − W(τ_maximum_vr)
14       enddo
15 else
16    do until imbalance > 0
17       if τ ∈ Q_busiest and W(τ)/2 ≥ imbalance
18          then migrate τ from Q_busiest to Q_k
19       endif
20       imbalance ← imbalance − W(τ)
21    enddo
22 endif
```

그림 3. 제안된 로드 밸런싱 알고리즘의 의사코드

여기서 $Q_k$는 일정 주기마다 태스크 이주를 시도하는 N개의 CPU중에서 임의의 CPU($CPU_k$)의 실행 큐이다. $Q_{busiest}$ 는 N개의 CPU중에서 가장 로드가 큰 실행 큐이다. 만약 $Q_k$ 와 $Q_{busiest}$ 가 동일하다면, 태스크 이주는 일어나지 않는다.

먼저 라인 1에서 본 알고리즘은 (3)에서 언급한 $L_{imbal}$를 계산한다. 라인 2는 입력 처리 응용 태스크의 현재 상태를 확인한다. 만약 입력 처리 응용 태스크가 수행 가능한 상태이거나 수행 중이라면 본 알고리즘은 라인 3에서 가장 큰 가상 런타임을 가진 태스크( $\tau_{maximum\ vr}$ )를 NULL값으로 초기화 시킨다. 이어서 라인 4~14는 루프를 수행하면서 $L_{imbal}$ 만큼 조건에 만족되는 태스크들을 이주시킨다. 라인 5~9은 $\tau_{maximum\ vr}$를 찾기 위해 $Q_{busiest}$에 속한 태스크들의 가상 런타임 크기가 서로 비교한다. 마지막으로 라인 10~11은 3장에서 설명한 태스크 이주조건에 만족되면 $\tau_{maximum\ vr}$를 $Q_{busiest}$에서 $Q_k$로 이주시킨다. 만약 입력 처리 응용 태스크가 수행 가능한 상태 혹은 수행 중이 아니라면 라인 16~21에서 3장에서 설명한 기존 로드 밸런싱 알고리즘을 수행한다.

그림 4는 제안된 로드 밸런싱 알고리즘이 적용된 예이다. 그림에서 알 수 있듯이 $Q_0$ 내에는 CPU 위주 태스크 $\tau_1$ 와 wake-up 태스크들이 존재하고, $Q_{busiest}$ 내에는 CPU 위주 태스크 $\tau_2$와 입력 처리 응용 태스크 $\tau_3$ 가 존재한다. 본 예에서는 $CPU_0$에서 주기적인 로드 밸런싱이 발생했다고 가정한다. 제안된 로드 밸런싱은 태스크 이주 시 가장 긴 가상 런타임을 가진 CPU 위주 태스크 $\tau_2$를 $Q_{busiest}$에서 $Q_0$으로 선별적으로 이주시킨다. 따라서 CPU 위주 태스크 $\tau_1$와 $\tau_2$는 $Q_0$ 내에 존재하게 되어 입력 처리

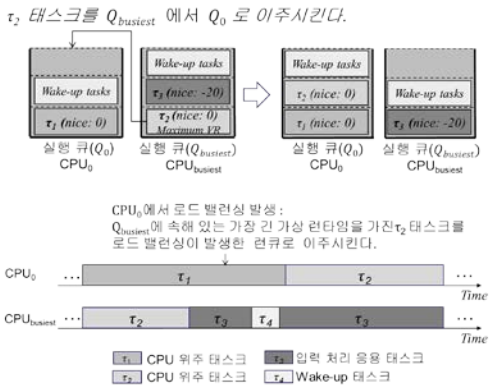


그림 4. 제안된 로드 밸런싱 기법의 예

응용 태스크인 $\tau_3$의 수행을 더 이상 방해하지 않게 되어 $\tau_3$의 선점 지연시간을 최소화한다.

제안된 기법들은 응답시간 단축을 위해 입력 처리 응용 태스크에게 높은 우선순위를 할당하고, 해당 태스크의 선점 지연을 최소화하기 위한 로드 밸런싱을 수행한다. 따라서 본 기법들은 일시적으로 다른 태스크들의 성능에 영향을 끼칠 수 있다. 하지만 실험결과 우리는 일반적인 스마트폰 환경에서 입력 처리 응용 태스크의 수행시간은 100ms 내외임을 확인하였다. 게다가 스마트폰과 같이 사용자와의 상호작용이 중요시되는 환경에서 비대화형 태스크의 일시적인 성능 저하는 사용자가 인지하기 어렵기 때문에 충분히 감수될 수 있다.

## 7. 실험 및 검증

본 장에서는 제안된 기법에 의한 응답시간 단축을 검증하기 위한 실험 결과를 보인다. 우리는 제안된 기법을 안드로이드 2.2 Froyo와 리눅스 2.6.32 커널이 탑재된 Nvidia Tegra2 보드 위에 구현하고 드로잉 응용인 bord 응용[13]의 응답 시간을 측정하였다. 응답시간 측정에는 커널 레벨 프로파일링 도구인 kernel shark[14]이 사용되었다.

우리는 드로잉 응용인 bord를 두 개의 CPU 위주 태스크들과 동시에 실행하였다. 기존 안드로이드 시스템에서 모든 응용 태스크는 우선순위 120을 부여받기 때문에 우리는 CPU 위주 태스크의 우선순위를 모두 120으로 설정하였다. 이때, 낮은 숫자는 보다 높은 우선순위를 나타낸다. 우리는 제안된 기법이 부스트 시키는 태스크의 우선순위를 115, 110, 105, 100으로 변경해가며 실험하였다.

그림 5는 기존 시스템과 제안된 기법들을 적용시킨 시스템에서 측정한 평균 응답시간 결과이다. 실험 결과에서 볼 수 있듯이, 입력 처리 응용 태스크의 우선순위를 높여 줄수록 응답시간이 감소하는 것을 볼 수 있다. 입력 처리 응용 태스크의 우선순위를

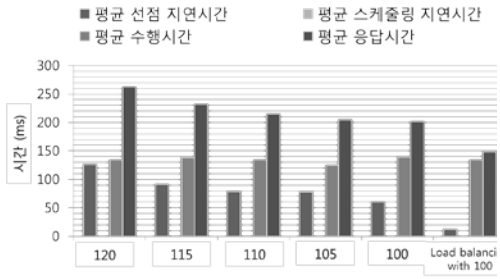


그림 5. 우선순위 부스트 기법과 로드 밸런싱 기법을 통한 응답시간 감소

100으로 부스트 시켰을 때 응답시간이 기존 시스템에 비해 최대 22% 감소하였다.

입력 처리 응용 태스크의 우선순위가 100일 때, 제안된 로드 밸런싱 기법을 적용시켰다. 로드 밸런싱 기법은 가상 긴 가상 런타임을 가진 태스크를 입력처리 응용 태스크가 존재하지 않는 다른 실행 큐로 이주시킨다. 실험 결과에서 볼 수 있듯이, 우선순위 부스트 기법만 적용 시켰을 때 보다 응답시간은 더욱 감소되었고 기존 시스템에 비해서는 최대 43.31% 감소되었다.

## 8. 결론

본 논문에서는 안드로이드의 저조한 사용자 응답성 문제를 분석하고 이를 해결하기 위한 프레임워크 지원 우선순위 부스트 기법과 로드 밸런싱 기법을 제안하였다. 이를 위해 먼저 안드로이드의 입력 이벤트 처리 과정을 분석하고, 사용자 응답성을 정량적으로 나타낼 수 있는 응답시간에 대해 정의하였다. 이어서 응답시간에 가장 큰 영향을 끼치는 지연 요소가 선점 지연시간임을 밝혔다. 제안된 우선순위 부스트 기법은 런타임에 입력 처리 태스크를 프레임워크 레벨에서 식별하고 커널 레벨에서 우선순위를 높임으로써 선점 지연시간을 감소 시켰다.

또한, 제안된 로드 밸런싱 기법은 가장 긴 가상 런타임을 가진 태스크를 우선순위 부스트 된 태스크가 존재하지 않는 다른 실행 큐에 선별적으로 이주시킴으로써 더욱 효과적으로 선점 지연시간을 감소 시켰다. 실험 결과 제안된 우선순위 부스트 기법은 기존 시스템 대비 최대 22% 단축된 응답시간을 보였고, 로드 밸런싱 기법은 기존 시스템 대비 최대 43.31% 단축된 응답시간을 보였다.

## 9. 참고 문헌

[1] http://techland.time.com/2011/12/07/is-android-doomed-to-lag-more-than-ios/

[2] C. S. Pabla, “Completely fair scheduler,“ Linux Journal, vol. 2009, August 2009.

[3] 2009. 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(4), pp. 347~364, 2007.

[4] S. Wang, Y. Chen, W. Jiang, P. Li, T. Dai, and Y. Cui, "Fairness and Interactivity of Three CPU Schedulers in Linux" Proceeding of RTCSA, pp. 172~177, August, 2009.

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

[6] K. Salah, A. Manea, S. Zeadally, and J.M. Alcaraz Calero, "On Linux starvation of CPU-bound processes in the presence of network I/O", Proceeding of Computers & Electrical Engineering, 2011, 37, (6), pp. 1090-1105.

[7] Torrey, A., Cleman, J., and Miller, P. "Comparing interactive scheduling in Linux", Software-Practices & Experience, 2007, 34, (4), pp. 347-364.

[8] T. Groves, J. Knockel, E. Schulte. BFS vs. CFS - Scheduler Comparison. December 2009

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

[10] Bornstein, D. 'Google i/o 2008-dalvik virtual machine internals', 2008

[11] https://plus.google.com/105051985738280261832/posts/XAZ4CeVP6DC

[12] https://developers.google.com/events/io/sessions/gooio2012/109/

[13] http://www.pixle.pl/bord/

[14] http://rostedt.homelinux.com/kernelshark/

손 용 석
2010년 아주대학교 정보컴퓨터공학과 (학사). 2010년 ~ 현재 서울대학교 융합과학기술대학원 지능형융합시스템학과 석사과정. 관심분야는 안드로이드 플랫폼, 내장형 시스템 소프트웨어, 운영체제, 멀티코어 스케줄링.

허 승 주
2007년 건국대학교 컴퓨터공학 (학사). 2009년 고려대학교 컴퓨터·전파통신공학부 (공학석사). 2009년 ~ 현재 서울대학교 융합과학기술대학원 지능형융합시스템학과 박사과정. 관심분야는 내장형 시스템 소프트웨어, 운영체제, 멀티코어 스케줄링.

유 종 훈
2001년 한국과학기술원 전기 및 전자공학과 (학사). 2005년 ~ 현재 서울대학교 전기컴퓨터공학부 석·박사 통합과정. 관심분야는 내장형 시스템 소프트웨어, 실시간 운영체제, 내장형 미들웨어, 가상화 기술.

홍 성 수
1986년 서울대학교 컴퓨터공학과 (학사). 1988년 서울대학교 컴퓨터공학과 (공학석사). 1994년 University of Maryland, Department of Computer Science (공학박사). 1995년 ~ 현재 서울대학교 전기정보공학부 교수. 2004년 ~ 2006년 서울대학교 내장형시스템연구센터 센터장. 2006년 ~ 2012년 서울대학교 융합과학기술대학원 지능형융합시스템학과 학과장. 2008년 ~ 현재 가헌신도리코재단 석좌교수. 2009년 ~ 현재 차세대융합기술원 스마트시스템연구소 연구소장. 2012년 ~ 현재 한국자동차공학회 전기전자시스템/ITS 부문위원회 부회장. 관심분야는 내장형 실시간 시스템 설계, 실시간 운영체제, 내장형 미들웨어, 실시간 시스템 설계 방법론, 소프트웨어 공학, 컴포넌트 기반 소프트웨어 설계.