ISSN 2383-6318(Print) / ISSN 2383-6326(Online)
KIISE Transactions on Computing Practices, Vol. 25, No. 12, pp. 616-621, 2019. 12
https://doi.org/10.5626/KTCP.2019.25.12.616

IoT 디바이스에서 쓰레드 동작을 최적화하기 위한 CPU 중재자

(CPU Mediator for Optimizing Thread Operations on IoT Devices)

임 근 식 † 강 동 현 †† 엄 영 익 †††
(Geunsik Lim) (Dong Hyun Kang) (Young Ik Eom)

요 약 운영체제 기술은 전통적으로 고성능 컴퓨팅 환경을 기반으로 발전해 왔으며, 최근의 운영체제들은 운영체제 내부의 쓰레드 간 성능 간섭을 줄이기 위해 주로 1:1 쓰레드 모델을 채택하고 있다. 그러나, 기존 쓰레드 모델은 제한된 하드웨어 자원을 가진 IoT 디바이스 환경에서는 시스템의 성능을 저하시키는 문제점을 가지고 있다. 이에, 본 논문에서는 고성능 기반 1:1 쓰레드 모델을 분석하고 이를 기반으로 IoT 디바이스 환경에 적합한 쓰레드 동작 최적화 기법을 제안한다. 특히, 제안 기법은 개발자에게 POSIX 호환성을 갖는 사용자 레벨의 쓰레드 스케줄링 API를 제공하기 때문에 개발자들은 단일화된 최적화 인터페이스를 통해 다양한 IoT 디바이스들의 쓰레드 성능을 일관되게 개선 및 제어할 수 있다. 본 논문에서 제안하는 기법은 Raspberry Pi3 환경에서 구현되었으며, 실험 결과 CPU 부하가 높은 작업 환경에서 쓰레드의 응답 시간을 기존 쓰레드 모델 대비 7.3배 줄일 수 있음을 확인하였다.

키워드: 쓰레드 모델, 쓰레드 최적화, 쓰레드 프로파일링, 쓰레드 스케줄링

Abstract The technologies of the operating systems have traditionally been developed based on the high-performance computing environments and they have been adopting the 1:1 thread mapping model to mitigate interference among the threads in the operating system. However, the conventional thread model reveals performance problems in the IoT environments because of its limited hardware resources. In this paper, we first study the 1:1 thread mapping model based on the high-performance computing systems and then propose a new scheme that optimizes the behavior of thread management on IoT devices. Particularly, since the proposed technique provides developers with a POSIX-com patible user-level thread scheduling API, the developers can control and improve the thread performance of various IoT devices with the unified optimization interface. We implemented the proposed scheme on Raspberry Pi3 and clearly confirmed through experiments that our scheme can reduce the response time of the threads 7.3 times on average, compared with the conventional thread model under the CPU-intensive workloads.

Keywords: thread model, thread optimization, thread profiling, thread scheduling

· 이 논문은 2019년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(IITP-2015-0-00284, (SW 스타랩) 중대형 디스플레이 기반 동시 다중 사용자 지원 UX 플랫폼 SW 개발)
· 이 논문은 2019 한국컴퓨터종합학술대회에서 'IoT 애플리케이션을 위한 쓰레드 동작 최적화 기법'의 제목으로 발표된 논문을 확장한 것임

† 정 회 원 : 성균관대학교 전자전기컴퓨터공학과
leemgs@skku.edu
†† 정 회 원 : 동국대학교 경주캠퍼스 컴퓨터공학과 교수
dhkang@dongguk.ac.kr
††† 종신회원 : 성균관대학교 소프트웨어대학 교수
(Sungkyunkwan Univ.)
yieom@skku.edu
(Corresponding author임)

논문접수 : 2019년 9월 9일
(Received 9 September 2019)
논문수정 : 2019년 10월 21일
(Revised 21 October 2019)
심사완료 : 2019년 10월 24일
(Accepted 24 October 2019)

정보과학회 컴퓨팅의 실제 논문지 제25권 제12호(2019. 12)

1. 서 론

최근 IoT(Internet-of-Things) 디바이스의 빠른 보급으로 인해 학계와 산업계의 많은 개발자들이 IoT 디바이스 환경에서 연구 및 개발을 진행하고 있다. 그러나, 일반적인 IoT 디바이스들은 서버 또는 데스크톱 환경에 비해 상대적으로 제한된 하드웨어 자원을 포함하고 있기 때문에 기존 운영체의 기술을 IoT 디바이스에서 수정 없이 사용할 경우, 의도하지 않은 디바이스의 성능 하락이 발생한다.

예를 들어, 현대의 운영체제들은 커널 공간과 사용자 레벨사이에서 쓰레드들(threads)을 1:1 구조로 매핑 시키는 1:1 쓰레드 모델을 사용함으로써, 쓰레드 간의 성능 간섭을 줄이고 시스템의 쓰레드 생성 및 스케줄링의 성능 오버헤드를 개선시키고자 노력한다[1-5]. 그러나, 이러한 1:1 쓰레드 모델은 IoT 디바이스 환경에서 새로운 문제들을 발생시킨다[6-11]. 다시 말해, IoT 디바이스 환경에서 쓰레드들의 수가 증가할수록 제한적인 시스템 자원을 가진 IoT 디바이스에서의 쓰레드 성능 저하가 발생한다. 이는 IoT 디바이스에서 실행중인 쓰레드의 개수가 증가할수록 CPU 자원에 대한 경쟁이 심화되고 그 결과 쓰레드들의 실행 시간을 지연시키기 때문이다. 또한, 쓰레드들이 제한된 CPU 자원을 얻기 위해 서로 경쟁하는 상황에서 발생하는 빈번한 쓰레드의 문맥교환(context switching)은 쓰레드의 응답 시간(response time)을 지연시키고 전반적인 성능 저하를 초래한다.

본 논문에서는 기존 IoT 디바이스 환경을 기반으로 하드웨어의 업그레이드 없이 쓰레드의 동작을 최적화하는 CPU 중재자를 제안한다. 제안된 CPU 중재자는 쓰레드의 응답 시간을 최소화하기 위해서 긴급히 처리해야 하는 쓰레드들에게 우선순위를 부여하고, 이를 통해 쓰레드 간의 스케줄링을 효율적으로 수행한다. 우선순위를 부여하기 위해, CPU 중재자는 새로운 사용자 레벨의 스케줄링 API(Application Programming Interface)를 애플리케이션 개발자에게 제공한다.

본 논문의 나머지 부분은 다음과 같이 구성되어 있다. 2절에서는 쓰레드 모델들의 배경 지식들을 기술한다. 3절에서는 제안된 기법의 설계 및 구현 내용을 상세히 설명한다. 4절은 평가 결과를 보여준다. 관련 연구는 5절에서 설명하며, 마지막으로, 6절에서 논문을 결론짓는다.

2. 쓰레드 모델

초창기 애플리케이션들은 사용자 레벨(user-level) 쓰레드를 사용하였기 때문에 애플리케이션 개발자들은 쓰레드의 생성 및 관리를 편리하게 수행할 수 있었다. 그러나, 멀티 코어(multi-core) 프로세서의 등장으로 커널 레벨(kernel-level)의 도움 없이 수행되는 사용자 레벨의 쓰레드는 쓰레드의 기능 및 성능상의 한계를 보여주었다. 이에, 대부분의 운영체제들은 멀티 코어 프로세서를 효율적으로 사용하기 위해 커널 레벨의 쓰레드 지원을 추가하였다[2]. 이에, 본 장에서는 사용자 레벨과 커널 레벨사이의 쓰레드 사용 모델에 대해서 설명한다. 그림 1은 각 쓰레드 모델이 어떻게 동작하는지 쓰레드 모델들 간의 동작구조 차이를 운영체제 별로 보여주고 있다.

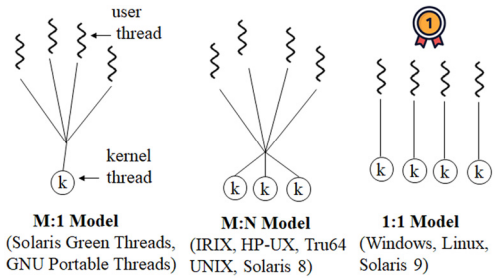

그림 1 운영체제의 쓰레드 모델들
Fig. 1 Thread models in operating systems

M:1 Model. 이 모델의 경우, 다수의 사용자 레벨의 쓰레드와 1개의 커널 레벨 쓰레드가 매핑되는 구조를 가지며, 사용자 레벨 쓰레드들 간의 빠른 문맥교환이 가능하다는 장점을 가진다. 특히, 사용자 레벨에서 다수의 쓰레드들을 관리할 수 있기 때문에 문맥교환을 매우 빠르게 수행할 수 있으며, 구현이 간단하고 이식성이 우수하다. 그러나, 이 모델은 커널 쓰레드가 1개이기 때문에 커널 레벨의 시스템 서비스가 블록킹(blocking) 되어 있는 동안 관련 쓰레드들이 오랜 기간 동안 지연될 수 있는 문제점이 있다. 예를 들어, 하나의 사용자 레벨 쓰레드가 블록킹이 가능한 시스템콜(예: fsync)을 호출할 때 전체 쓰레드들이 함께 블록킹 된다. 즉, 이 모델은 병렬처리에 제한을 받기 때문에 여러 개의 쓰레드들로 구성되는 단일 프로세스 애플리케이션은 멀티 코어 프로세서의 장점을 얻을 수 없다. 또한, 커널이 사용자 레벨의 쓰레드 존재 유무를 모르기 때문에 커널이 사용자 레벨의 쓰레드들을 스케줄링을 할 수 없다[9].

M:N Model. 이 모델은 다수의 사용자 레벨 쓰레드들을 다수의 커널 쓰레드들에게 매핑한다는 점에서 이전의 M:1 모델과 차이가 있다. 즉, M:N 모델에서는 사용자 레벨의 쓰레드를 사용 가능한 커널 레벨의 쓰레드 개체 중 하나와 연결함으로써, M:1 모델에서의 블록킹에 대한 단점을 보완하였다. 특히, 개발자가 이 모델을 효과적으로 사용한다면, 대부분의 문맥교환이 사용자 레벨에서 발생되기 때문에 문맥교환 속도가 빠르다는 장점이 있다. 그러나, 이 모델은 다른 모델에 비해 구현이 복잡하다는 문제점이 있다. 특히, 사용자 레벨에서 선언된 쓰레드의 우선 순위가 커널 레벨로 전달되지 않기

때문에 애플리케이션 개발자는 커널 개체에 대한 직접적 제어권을 가질 수 없다[12,18].

1:1 Model. 최근의 운영체제들은 대부분 1:1 쓰레드 모델을 채택하고 있다. 이는 1:1 쓰레드 모델이 커널 레벨에서 쓰레드들을 생성 및 스케줄링하기 때문에 멀티코어 프로세서 환경에서 하나의 쓰레드가 서로 다른 CPU에서 동시에 실행되도록 지원하기 때문이다. 특히, Linux의 O(1) 스케줄러의 도입은 1:1 모델의 쓰레드 수행 속도를 가속화하였다. Linux O(1) 스케줄러의 실행 큐(run queue)와 비트맵 우선순위 배열(bitmap array)이 효과적으로 커널 레벨에서의 쓰레드간 우선 순위를 지원함으로써 쓰레드의 성능이 대폭 개선되었기 때문이다[2,13]. 그러나, 본 모델은 커널 레벨에서 문맥교환을 수행하기 때문에 시스템 콜이 발생할 때 다른 모델에 비해 상대적으로 느린 문맥교환 속도를 보여준다[14,15]. 특히, 각각의 쓰레드가 커널 레벨의 시스템 자원을 사용하기 때문에 애플리케이션 개발자들이 많은 쓰레드들을 동시에 생성한다면, 쓰레드들의 성능이 전반적으로 감소하게 된다[1,16,17].

3. CPU 중재자

본 논문에서는 IoT 디바이스 환경에서 1:1 쓰레드 모델을 효율적으로 지원하기 위한 최적화 방안으로 CPU 중재자를 제안한다. 그림 2는 CPU 중재자 기반 시스템의 전체 흐름도를 나타내고 있다. CPU 중재자는 기존 쓰레드 모델에 비해 빠른 응답성을 보장하기 위해 쓰레드 레벨 스케줄링을 지원하며, 이를 위해 사용자에게 새로운 스케줄링 API를 제공한다.

3.1 쓰레드 레벨 스케줄링

IoT 디바이스의 모든 쓰레드들이 빠른 응답성을 요구하지는 않기 때문에 쓰레드들의 응답성을 기준으로 다음과 같이 쓰레드들을 그룹 레벨로 묶은 후, 그룹 내부의 쓰레드들에게 공평하게 CPU자원을 할당한다[20].

- Urgent Group: IoT 디바이스의 스크린 상에 표시되어 사용자와 상호 동작하고, 빠른 응답성과 높은 CPU 사용률을 요구하는 쓰레드 그룹.
- Normal Group: IoT 디바이스의 스크린 상에 표시되지만, 빠른 응답성을 요구하지 않는 쓰레드 그룹.
- Service Group: IoT 디바이스에서 서비스 데몬 역할을 수행하는 쓰레드로서, 실시간 응답성에 대한 특성을 요구하지 않는 쓰레드 그룹.
- Background Group: 작업을 수행하고 있으나, 사용자에게는 보이지 않는 쓰레드들의 그룹.

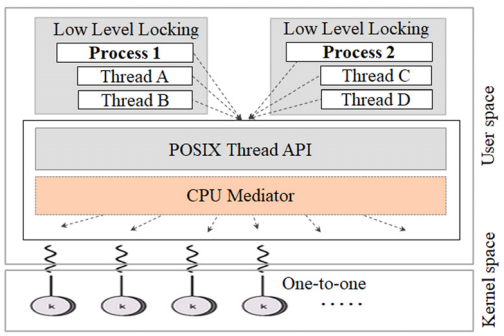

그림 2 CPU 중재자 기반 시스템의 전체 흐름도
Fig. 2 Overall flow of the system with the CPU Mediator

그룹 레벨의 스케줄링은 CPU 자원을 공평하게 분배함으로써 효율적인 쓰레드 관리를 보장하지만, 빠른 응답성을 요구하는 쓰레드가 CPU 자원을 즉시 획득하지 못한다는 문제점을 가진다. 또한, 쓰레드의 개수가 증가할 경우, CPU 자원에 대한 경합(contention)이 빈번하게 발생하기 때문에 쓰레드의 응답성은 상당히 저하된다. 이러한 문제점을 해결하기 위해, CPU 중재자는 기존 그룹 레벨 스케줄링을 기반으로 쓰레드 레벨 스케줄링을 추가적으로 지원한다. 즉, 빠른 응답성을 요구하는 쓰레드가 존재하지 않을 경우, 기존 스케줄링 기법과 동일하게 그룹 단위로 쓰레드들을 스케줄링 한다(즉, 일반 모드). 만약 빠른 응답성을 요구하는 쓰레드가 존재하게 되면 해당 쓰레드의 CPU 점유를 즉시 보장하기 위해 쓰레드 레벨 스케줄링을 수행한다(즉, 응답성 모드).

그림 3은 그룹 레벨 스케줄링(즉, 일반 모드)에서 쓰레드 레벨 스케줄링(즉, 응답성 모드)으로 전환하기 위한 CPU 중재자의 예제를 보여준다. 그림에서 보여주듯이, 빠른 응답성을 요구하는 쓰레드에 대한 정보가 API (3.2에서 자세하게 설명함)를 통해 호스트로부터 전달되면, CPU 중재자는 빠른 응답성을 요구하는 쓰레드들을 기존 그룹에서 Fast Region으로 일시적으로 이동시키고 (TID 1과 TID 3) 응답성이 중요하지 않은 다른 쓰레드들을 Lazy Region으로 이동시킨다. 그리고 Fast Region에 속한 쓰레드들에게 더 많은 CPU 점유율을 할당함으로써 해당 쓰레드들의 빠른 응답성을 보장한다. 반면, Lazy Region에 속하는 쓰레드들은 Fast Region에 비해 상대적으로 낮은 CPU 점유율을 할당받기 때문에 잠시 동안 쓰레드의 실행이 지연된다. 마지막으로, CPU 중재자는 Fast Region의 모든 쓰레드가 빠른 응답성을 사용자에게 제공한 이후에는 쓰레드 레벨 스케줄링을 다시 그룹 레벨 스케줄링으로 전환한다.

3.2 SCHED_TEK API

SCHED_TEK API는 쓰레드 레벨 스케줄링 장치를 사용자에게 제공하기 위한 프로그래밍 인터페이스이다. 운영체제는 사용자로부터 빠른 응답성을 요구하는 쓰레드가 어떤 것인지 커널 레벨에서 확인할 수 없다. 따라서 본 기법에서는 CPU 중재자에게 응답성이 중요한 쓰

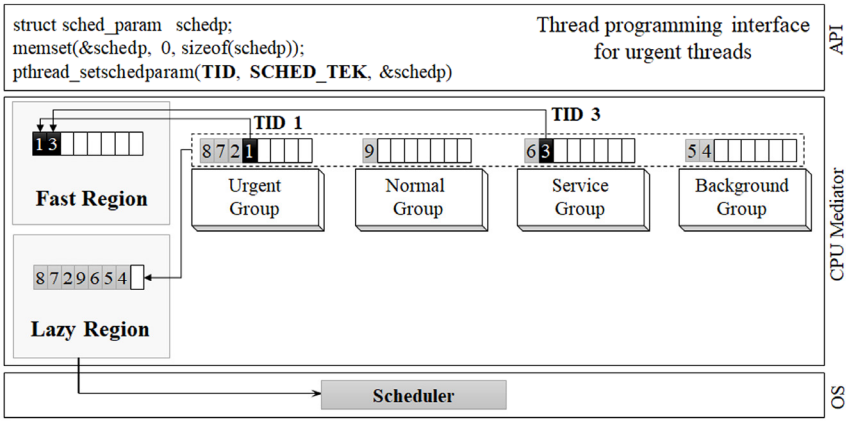

그림 3 SCHED_TEK으로 긴급한 쓰레드들의 응답 시간을 가속화하기 위한 쓰레드 프로그래밍 모델
Fig. 3 Thread programming model for accelerating the response time of urgent thread(s) with SCHED_TEK

레드에 대한 힌트 정보를 사용자로부터 전달하기 위해 POSIX Thread API을 새롭게 정의하여 사용자에게 제공한다. 아래의 두개의 명령은 새롭게 정의한 API이다.

- gettid(): 사용자 레벨에서 쓰레드의 ID 정보를 얻기 위한 API
- pthread_setschedparam(): 빠른 응답성을 요구하는 쓰레드에 대한 힌트 전송을 위한 API

CPU 중재자가 제공하는 API를 이용하여 사용자는 커널 레벨에서 수행되는 스케줄링 우선순위를 사용자가 쉽게 제어할 수 있으며, 지연되는 쓰레드에 대한 불필요한 문맥교환 횟수를 감소시킬 수 있다. 예를 들어, 사용자가 터치 스크린을 이용하여 특정 메뉴를 선택하였을 때 사용자는 선택한 메뉴를 처리하는 쓰레드들에 대해 SCHED_TEK을 지정할 수 있으며, 설정된 쓰레드의 작업을 완료할 때까지 다른 쓰레드들(예: 소프트웨어 업데이트 쓰레드, 백그라운드 서비스 쓰레드)의 CPU 점유율을 순간적으로 지연시킴으로써 빠른 응답을 보장한다.

4. 실 험

본 절에서는 논문에서 제안한 CPU 중재자와 기존 시스템 간의 실험 평가 결과를 보여준다. 제안 기법의 성능 효과를 검증하기 위하여 IoT 디바이스 환경을 구축하였으며, 실험에 사용된 환경은 Raspberry Pi3, SoC Broadcom BCM2837, CPU 4x ARM Cortex-A53@1.2GHz, GPU Broadcom VideoCore IV, RAM 1GB LPDDR2 (900MHz), MicroSD 64GB로 구성되어 있고, Tizen/ARM 32bit[20], Linux Kernel 4.4 운영체제를 사용하였다.

4.1 빠른 응답성을 요구하는 쓰레드의 응답성 실험

IoT 디바이스가 빠른 응답성을 요구하는 쓰레드들을 처리하는 속도를 효과적으로 개선하는지 확인하기 위해 우리는 아래의 테스트 시나리오로 실험을 수행하였다.

단계 1: 실험 환경은 CPU 경쟁 조건을 만들기 위하여 각 그룹별(예: urgent, normal, service, background group) 5개의 CPU 집중적인 쓰레드들을 실행한다. **단계 2:** CPU 사용율이 항상 100%인 상황에서 사용자가 IoT 디바이스의 특정 메뉴를 선택할 때 그 메뉴를 처리하는 쓰레드들의 응답 속도를 측정한다.

그림 4는 저사양의 CPU 클럭 속도(CPU clock: 1.2GHz)를 가지고 있는 냉장고와 같은 IoT 디바이스 환경을 Raspberry Pi3에 시뮬레이션한 결과를 보여주며, 사용자가 터치스크린을 통하여 특정 메뉴를 선택하였을 때 빠른 응답성을 요구하는 쓰레드들의 CPU 사용율을 측정한 실험 결과이다. 그림 4에서 보여주듯이, 기존 시스템은 쓰레드들이 CPU 자원을 확보하기 위하여 경쟁할 때 빠른 응답성을 요구하는 쓰레드들의 응답 시간이 1423 ms ~ 2410 ms (평균 응답 시간: 1719 ms) 사이로 불규칙적인 결과를 보여준다. 그러나, CPU 중재자를 사용하는 제안 시스템의 경우, 동일한 그룹의 많은 쓰레드들이 CPU 자원을 대상으로 활발하게 경쟁함에도 불구하고 147 ms ~ 273 ms (평균 응답 시간: 235 ms)의 응답시간을 보여주며, 이는 기존 시스템 대비 평균 7.3배의 성능 개선을 보여주는 것이다.

그림 5는 기존 시스템과 제안 시스템 간의 문맥교환 발생횟수들을 비교 실험한 결과이다. CPU 중재자의 SCHED_TEK은 쓰레드들의 공평한 스케줄링이 아닌 긴급하지 않은 쓰레드들의 CPU 스케줄링을 지연시키는 방법을 제공하기 때문에 1:1 쓰레드 모델의 성능 저하를 초래하는 쓰레드들의 빈번한 문맥교환 동작을 최소화한다. 그 결과 CPU 중재자는 쓰레드들 간에 발생하는 문맥교환 동작 회수를 기존 시스템 대비 42%까지 줄임을 볼 수 있다.

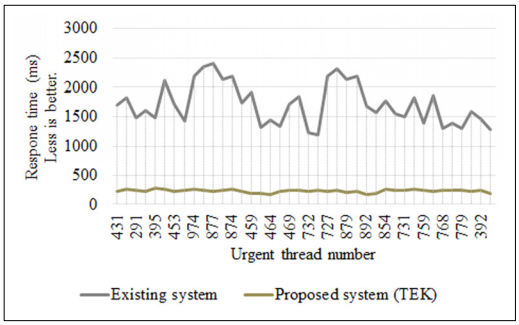

그림 4 CPU 경쟁조건에서 긴급 쓰레드들의 사용자 응답성을 개선하기 위한 SCHED_TEK 정책의 실험 결과

Fig. 4 Experiments of user-space SCHED_TEK policy for improving the user responsiveness of urgent thread(s) during CPU contention

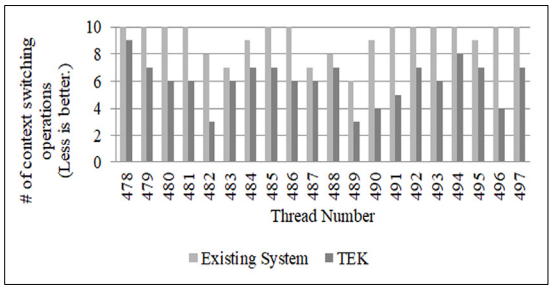

그림 5 쓰레드들의 문맥교환 동작 실행 횟수

Fig. 5 The execution frequency of threads' context switching operations

5. 관련 연구

*Ulrich Drepper*는 Linux 시스템의 표준으로써 사용중인 LinuxThread 라이브러리가 쓰레드에 대한 POSIX 표준을 충족시키지 못한다는 것을 지적하였다[1]. 또한 기존 쓰레드 라이브러리가 최신 프로세서 아키텍처를 고려하지 않았기 때문에 Fast Use-Space Mutex (FUTEX)라는 차세대 쓰레드 라이브러리의 설계를 제안하였다.

*Wong*은 2개의 Linux 스케줄러 O(1)과 CFS[19]에서 사용하는 스케줄링 기술들을 설명하였다[2]. CFS는 2.6.23 Linux 커널에서 O(1)을 대체하는 Linux 커널 스케줄러이다. CFS의 설계 목표는 대화식 성능을 저하시키지 않으면서 실행 작업 공간에서 공정한 CPU 자원 할당을 제공하는 것이다.

이러한 연구들은 Large-Scale 하드웨어의 컴퓨팅 환경에서 쓰레드의 성능을 향상시키는 방법에만 집중하여 왔다. 즉, 이러한 아이디어들은 IoT 디바이스와 같은 제한적인 시스템 자원을 가진 Small-Scale 하드웨어 환경에서 쓰레드의 성능 저하 문제들을 다루지 못하고 있다.

6. 결 론

본 논문에서는 제한된 CPU 및 메모리 자원을 가진 IoT 디바이스 환경에서 쓰레드의 성능을 최적화하기 위한 CPU 중재자를 제안하였다. CPU 중재자는 사용자 레벨에서 쓰레드의 스케줄링이 가능하도록 새로운 API을 제공하고 이를 기반으로 특정 쓰레드들의 빠른 응답성을 보장한다. 실험 평가를 통해 CPU 중재자 기반의 시스템이 기존 시스템 대비 7.3배 빠른 응답성을 보장하는 것을 확인하였다.

References

[1] U. Drepper and I. Molnar. (2005, Feb. 21). The Native POSIX Thread Library for Linux [Online]. Available: https://akkadia.org/drepper/nptl-design.pdf, (downloaded 2019, Aug. 25)

[2] C. Wong, I. Tan, R. Kumari, J. Lam, and W. Fun, "Fairness and Interactive Performance of O(1) and CFS Linux Kernel Schedulers," *Proc. of the International Symposium on Information Technology*, Vol. 4, pp. 1-8, 2008.

[3] F. Mueller, "A Library Implementation of POSIX Threads under UNIX," *Proc. of the USENIX Conference*, pp. 29-41, 1993.

[4] H.-J. Boehm, *Threads cannot be Implemented as a Library*, ACM Sigplan Notices, Vol. 40, No. 6, pp. 261-268, 2005.

[5] J. Nakashima and K. Taura, "MassiveThreads: A Thread Library for High Productivity Languages," *Proc. of the Lecture Notes in Computer Science*, Vol. 8665, pp. 222-238, 2014.

[6] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-Aware Thread Management," *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 145-160, 2018.

[7] B. Barney, *POSIX Threads Programming*, National Laboratory, Vol. 5, p. 46, 2009.

[8] F. W. Miller, "pk: A POSIX Threads Kernel," *Proc. of the USENIX Annual Technical Conference, FREENIX Track*, pp. 179-181, 1999.

[9] R. S. Engelschall, "Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation," *Proc. of the USENIX Annual Technical Conference*, pp. 1-12, 2000.

[10] J. Howell, B. Parno, and J. R. Douceur, "How to Run POSIX Apps in a Minimal Picoprocess," *Proc. of the USENIX Annual Technical Conference*, pp. 321-332, 2013.

[11] M. Rieker, J. Ansel, and G. Cooperman, "Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux," *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*,

Vol. 6, pp. 492-498, 2006.
[12] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast User-level Locking in Linux," *Proc. of the Ottawa Linux Symposium*, Vol. 85, 2002.
[13] S. Thibault, (2005, Jun. 27). A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines [Online]. Available: https://arxiv.org/abs/cs/0506097, (downloaded 2019, Aug. 25)
[14] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative Task Management without Manual Stack Management," *Proc. of the USENIX Annual Technical Conference*, General Track, pp. 289-302, 2002.
[15] A. Gidenstam and M. Papatriantafilou, "LFthreads: A Lock-Free Thread Library," *Proc. of the International Conference on Principles of Distributed Systems*, pp. 217-231, 2007.
[16] U. Drepper, (2011, Nov. 5). Futexes are Tricky [Online]. Available: https://www.akkadia.org/drepper/futex.pdf, (downloaded 2019, Aug. 25)
[17] P. Holman and J. H. Anderson, "Locking under Pfair Scheduling," *ACM Transactions on Computer Systems (TOCS)*, Vol. 24, No. 2, pp. 140-174, May 2006.
[18] N. C. Brown, "C++ CSP2: A Many-to-Many Threading Model for Multicore Architectures," *Proc. of the Communicating Process Architectures 2007: WoTUG-30*, pp. 183-205, 2007.
[19] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, "Towards Achieving Fairness in the Linux Scheduler," *ACM SIGOPS Operating Systems Review*, Vol. 42, No. 5, pp. 34-43, Jul. 2008.
[20] G. Vashisht and R. Vashisht, "A Study on the Tizen Operating System," *International Journal of Computer Trends and Technology (IJCTT)*, Vol. 12, 2014.

임 근 식
2003년 아주대학교 컴퓨터공학과 학사 2014년 성균관대학교 IT융합학과 석사 2003년~현재 삼성전자 수석연구원. 2019년~현재 성균관대학교 전자전기컴퓨터공학과 박사과정. 관심분야는 운영체제, 모바일 플랫폼, 시스템 최적화, On-Device AI

강 동 현
2000년 한국산업기술대학교 컴퓨터공학과 학사. 2008년 성균관대학교 전자전기컴퓨터공학과 석사. 2018년 성균관대학교 전자전기컴퓨터공학과 박사. 2018년~2019년 삼성전자 책임연구원. 2019년~현재 동국대학교 경주캠퍼스 조교수. 관심분야는 스토리지 시스템, 시스템 소프트웨어, 운영체제

엄 영 익
1983년 서울대학교 계산통계학과 학사 1985년 서울대학교 전산과학과 석사 1991년 서울대학교 전산과학과 박사 1993년~현재 성균관대학교 소프트웨어대학 교수 관심분야는 시스템 소프트웨어, 운영체제, 파일/스토리지 시스템, 가상화, UI/UX