(2011추계 우수발표논문) 멀티코어 시스템에서 공정성 보장을 위한 Virtual Runtime 기반 로드 밸런싱 알고리즘

(Virtual Runtime based Load Balancing for Guaranteeing Fairness on Multicore Systems)

허 승 주† 유 종 훈††
(Sungju Huh) (Jonghun Yoo)

홍 성 수†††
(Seongsoo Hong)

요 약 리눅스의 기본 스케줄러인 CFS 는 클라우드 서버와 같은 대규모 시스템에서 널리 사용되고 있지만, 이는 시스템의 규모가 극도로 커짐에 따라 시스템이 요구하는 수준의 공정성을 보장하지 못한다. 이 논문은 CFS 에 대한 심도 깊은 분석을 통해 공정성 보장의 실패의 원인을 규명하고, 이를 해결하기 위한 virtual runtime 기반의 로드 밸런싱 알고리즘을 제안한다. 이는 태스크들의 virtual runtime 차이를 바운드 시키기 위해 주기적으로 태스크 이주를 수행한다. 이를 위해 알고리즘은 인접한 두 CPU 의 load 차이가 최대 가중치 차이 이하가 되도록 바운드 하고, virtual runtime 이 큰 태스크들을 load 가 큰 CPU 에서 수행되게 보장한다. 우리는 제안된 알고리즘을 리눅스 커널 2.6.38.8 상에 구현하고 일련의 실험을 수행하였다. 그 결과 기존 CFS 의 경우 virtual runtime 차이는 선형적으로 증가하는데 반해 제안된 기법은 50.53 단위 시간으로 virtual runtime 차이를 바운드시킬 수 있으며 고작 0.14%의 런타임 오버헤드를 야기시킴을 보였다.

키워드 : 멀티코어 스케줄링, 로드 밸런싱, 공정성

Abstract While the primary task scheduler of Linux, CFS, is widely adopted for the large-scale cloud system, it cannot provide a desired level of fairness when a system scales up to an extreme degree. This paper formally analyzes the behavior of CFS to precisely characterize the reason why it fails to achieve the fairness in multicore systems. Based on the analysis, we present a virtual runtime-based load balancing algorithm which directly bounds the maximum virtual runtime difference among tasks by periodically migrating tasks. In doing so, it bounds the load difference between two adjacent cores by the largest weight in the task set and makes the core with larger virtual runtimes receive a larger load and thus runs more slowly. We have implemented the algorithm into Linux kernel 2.6.38.8. Experimental results show that the maximal virtual runtime difference is 50.53 time units while incurring only 0.14% of run-time overhead comparing to CFS.

Key words : Multi-core Scheduling, Load Balancing, Fair Share Scheduling

1. 서 론

CFS(complete fair scheduler)[1]는 2.6.23 커널 이래로 리눅스의 주된 태스크 스케줄러로서 사용되어 왔다. CFS 의 목적은 태스크들의 가중치(weight)에 비례하여 CPU 시간을 할당해 주는 것이다. 일반적인 작업 환경에서 이는 기존 스케줄러에 비해 높은 반응성과 공정성을 보이기 때문에 CFS 는 모바일 장치에서부터 대규모의 클라우드 서버까지 다양한 분야의 컴퓨팅 시스템에서 사용되고 있다.

불행히도 CFS 는 시스템의 규모가 극도로 커짐에 따른 scalability 지원이 미흡한 것으로 알려져 있다. 특히 수천에서 수만 개의 태스크가 공존하는 시스템에서 CFS 에 의해 스케줄링 되는 태스크들은 종종 심각한 기아현상을 겪는다고 보고된 바 있다 [2]. 그럼에도 불구하고 CFS 에 대한 분석적이고 비평적인 평가는 그리 많지 않으며 대규모 시스템에서 scalability 지원 미흡의 원인을 명확하게 설명하지 못한다 [3][4].

본 논문에서 우리는 CFS 를 명확하게 분석하여 멀티코어에서 다양한 가중치를 갖는 태스크들을 처리할 때 CFS 가 공정성을 보장하지 못함을 보인다. 우

· 본 연구는 삼성전자 DMC 연구소의 지원을 받아 수행하였음. (No. 0418-20110011, HW SW 융복합 원천기술 개발)
· 이 논문은 제38회 추계학술발표회에서 ‘ 멀티코어 시스템에서 공정성 향상을 위한 Virtual Runtime 기반 로드 밸런싱 메커니즘’의 제목으로 발표된 논문을 확장한 것임
† 학생회원: 서울대학교 융합과학기술대학원 지능형융합시스템학과/전기컴퓨터공학부
sjhuh@redwood.snu.ac.kr
jhyoo@redwood.snu.ac.kr
†† 종신회원: 서울대학교 전기컴퓨터공학부 교수
sshong@redwood.snu.ac.kr
논문접수:
심사완료:

정보과학회논문지: 컴퓨팅의 실제 및 레터 제XX권 제X호(2011.XX)

리의 분석 결과는 CFS 로 스케줄링된 태스크들이 멀티코어 환경에서 예측할 수 없는 CPU 시간 분포를 갖는 이유를 설명한다. 분석에 기반하여 우리는 virtual runtime 에 기반한 새로운 로드 밸런싱 메커니즘을 제안한다. 이 메커니즘은 주기적으로 태스크들을 이주하여 임의의 두 태스크의 virtual runtime 차이를 상수로 바운드한다.

이 논문의 나머지 부분은 다음과 같이 구성된다. 2 장에서는 본 연구의 관련 연구를 정리한다. 3 장에서는 CFS 의 동작원리를 명확히 분석한다. 이어서 4 장은 멀티코어에서 CFS 가 공정성을 성공적으로 달성하지 못하는 문제점을 보인다. 5 장에서는 이러한 문제를 해결하기 위한 virtual runtime 기반 로드 밸런싱 알고리즘을 설명한다. 6 장은 제안된 메커니즘의 구현과 실험 결과를 보인다. 마지막으로 7 장에서는 본 논문의 결론을 내린다.

2. 관련 연구

최근 수년간 스케줄링 알고리즘의 측면에서 공정성을 달성하기 위한 여러 연구들이 있어왔다. 이 연구들은 알고리즘이 사용하는 실행큐 구조에 따라 두 가지로 분류된다: (1) 전역 실행큐와 (2) 분산 실행큐.

전자[5][6]는 일반적으로 가용한 자원 정보를 전역으로 활용할 수 있기 때문에 CPU 의 수가 적은 시스템에서는 후자에 비해 높은 성능을 보이고 쉽게 공정성을 달성할 수 있다. 하지만 이들은 시스템의 규모가 커짐에 따라 동기화나 lock contention 으로부터 야기되는 성능 저하를 야기한다.

분산 실행큐를 사용하는 알고리즘들[1][7][8] 은 CPU 끼리 오직 작은 양의 데이터만을 공유하기 때문에 위에서 언급된 scalability 문제를 쉽게 해결할 수 있다. 이 분류에 속하는 대부분의 알고리즘들은 실행큐 내에서 독자적인 방법으로 공정성을 달성하고, 멀티코어 환경에서 공정성 달성을 위해 가중치 기반의 로드 밸런싱을 사용한다. 하지만 불행히도, 이들이 도입한 가중치 기반의 로드 밸런싱은 모든 workload 에 대해서 공정성 달성을 보장하지 못한다는 문제점을 보인다. 우리는 이 문제를 4 장에서 대해서 상세히 다룬다.

이러한 가중치 기반 로드 밸런싱의 문제를 해결하기 위한 연구로는 우리가 아는 한 Li 가 제안한 Distributed Weighted Round Robin(DWRR)[9]이 유일하다. 이 알고리즘은 weighted round robin 방식으로 실행큐 내의 공정성을 달성하고, round balancing 알고리즘을 사용하여 서로 다른 실행큐 간의 공정성을 달성한다. 하지만 DWRR 은 CPU 당 2 개의 실행큐(Expired/Active)를 유지하기 때문에 마치 O(1) 스케줄러가 그랬던 것처럼 반응성이 떨어지는 문제를 내포한다 [10].

3. CFS의 분석

CFS 는 태스크의 가중치에 기반하여 공정성을 달성하는 스케줄링 알고리즘이다. 태스크의 가중치는 시스템 관리자에 의해 각 태스크에 부여되는 nice 값에 의해 정해진다. Nice 값은 -20 과 19 사이의 정수로서 작은 nice 값은 큰 가중치에 상응한다.

CFS 는 SMP(symmetric multi-processor)를 지원하기 위해 각 CPU 마다 태스크의 실행큐를 유지한다. 실행큐 내의 태스크들은 각 태스크에 부여된 virtual runtime 이 증가하는 순서로 정렬된다. 이때 virtual runtime 은 태스크의 가중치에 의해 역으로 스케일링된 누적 실행 시간이다. 구체적으로, CFS 에서 시간 t 에 태스크 τ_i 의 virtual runtime $VR(\tau_i, t)$은 아래와 같이 정의된다.

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

여기서 ω_0 은 nice 값 0 의 가중치를, $W(\tau_i)$는 태스크 τ_i 의 가중치를 의미한다. 또한 $PR(\tau_i,t)$는 시간 t 일 때 태스크 τ_i 의 실제 누적 실행시간을 의미한다.

CFS 는 실행큐 내의 모든 태스크들에게 가중치에 비례하는 time slice 를 할당하여 태스크가 이 기간 동안 선점 당하지 않고 실행될 수 있게끔 한다. CFS 에서 태스크 τ_i 의 time slice TS_i 는 아래와 같이 계산된다.

$$TS_i = \frac{W(\tau_i)}{\sum_{j\in\varphi} W(\tau_j)} \times P \quad (2)$$

여기서 φ 는 실행큐 내에 있는 실행 가능한 태스크들의 집합을 의미하고, P 는 주어진 작업량에 대해서 상수이다. P 는 다음과 같이 정의된다.

$$P = \begin{cases} sysctl\_sched\_latency & \text{if } n < nr\_latency \\ min\_granularity \times n & otherwise \end{cases} \quad (3)$$

여기서 n은 실행큐 내의 실행 가능한 태스크의 개수를 의미한다. *sysctl_sched_latency*, *nr_latency*, *min_granularity* 는 시스템 전체에 대한 상수이며 현재 리눅스 상에는 6, 8, 0.75 로 정의되어 있다.

CFS 의 런타임 알고리즘은 다음과 같다. 스케줄링 시점마다 CFS 는 현재 수행 중인 태스크의 virtual runtime 을 갱신하고 *NEED_RESCHED* flag 를 확인한다. *NEED_RESCHED* flag 는 수행중인 태스크가 자신에게 할당된 time slice 를 소진하면 설정된다. 만약 이 flag 가 설정되어 있으면 CFS 는 현재 실행 중인 태스크를 선점하고 실행큐 내의 가장 작은 virtual

runtime 을 갖는 태스크를 스케줄링 한다. 이러한 스케줄링 정책은 실행큐 내의 모든 태스크들의 virtual runtime 차이를 일정 수준 이하로 밸런싱 함으로써 공정성 만족을 보장한다.

CFS 는 멀티코어 환경에서 보다 공정성을 보장하기 위해 가중치 기반 로드 밸런싱을 수행한다. CFS 는 실행큐 마다 load 값을 유지하여 시스템 내의 실행큐에 대해 균형을 맞춘다. 실행큐 Q_i 의 load 는 아래와 같이 정의된다.

$$L_{Q_i} = \sum_{j \in \varphi_i} W(\tau_j) \quad (4)$$

여기서 φ_i 는 실행큐 Q_i 내의 실행 가능한 태스크들의 집합을 의미한다.

CFS 가 로드 밸런싱을 수행하는 시점은 다음과 같다: (1) 태스크가 새롭게 생성되거나(fork, exec) 대기 상태에서 깨어날 때(wakeup), (2) 실행큐 내의 태스크들이 수행을 종료하여 실행큐가 유휴 상태가 되었을 때, (3) 서로 다른 실행큐 사이에 load 의 불균형이 발생했을 때. 첫 번째 경우, CFS 는 새로 생성되거나 깨어난 태스크 τ_i 를 load 가 가장 작은 실행큐 Q_{idlest} 로 이주시킨다. 이때 태스크 τ_i 의 virtual runtime 은 이주된 Q_{idlest} 에서 가장 작은 virtual runtime 보다 조금 작은 값으로 설정되어 τ_i 가 빠른 응답시간을 가질 수 있도록 보장한다. 두 번째 경우, CFS 는 load 가 가장 큰 실행큐 $Q_{busiest}$ 를 찾아 태스크들을 Q_i 로 이주시킨다. 이 때 CFS 는 이주시킨 결과가 더 큰 불균형을 야기하지 않을 만큼의 태스크들을 한번에 이주시킨다. 세 번째 경우, CFS 는 실행큐 Q_i 에 대해서 일정 주기마다 load 가 가장 큰 실행큐 $Q_{busiest}$ 로부터 태스크들을 가져올 수 있게 한다. 첫 번째 경우와 마찬가지로 이주시킨 결과가 더 큰 불균형을 야기하지 않을 만큼의 태스크를 한번에 이주시킨다. 한 번에 이주시키는 태스크의 양은 다음과 같다.

$$L_{imbal} = min(min(L_{busiest}, L_{avg}), L_{avg} - L_k) \quad (5)$$

여기서 $L_{busiest}$ 는 $Q_{busiest}$ 의 load 이고 L_{avg} 는 시스템의 전체 평균 load 이다. CFS 는 다음의 조건이 만족될 때에는 태스크를 이주시키지 않음으로써 너무 잦은 로드 밸런싱으로 인한 오버헤드를 방지한다.

$$L_{imbal} < \min_{\tau_i \in \varphi_{busiest}} (W(\tau_i))/2 \quad (6)$$

여기서 $\varphi_{busiest}$ 는 $Q_{busiest}$ 에 있는 실행 가능한 태스크 집합이다. 로드 밸런싱을 수행하는 주기는 Q_i 와 $Q_{busiest}$ 의 캐시 locality 에 따라 다르다. 현재 리눅스 상에는 최소 1 분에 한번씩은 주기적인 로드 밸런싱을 수행하도록 정의되어 있다.

표 1. CFS가 멀티코어 환경에서 공정성을 보장하지 못하는 태스크 집합의 예

	Nice value	*Weight*	*Initial distribution*
τ_1	0	1024	P_1
τ_2	5	335	P_2
τ_3	5	335	P_2
τ_4	5	335	P_2
τ_5	5	335	P_2

4. 멀티코어 환경에서 CFS의 문제점

본 장에서는 CFS 가 멀티코어에서 공정성을 보장하지 못함을 명확히 보인다.

임의의 시간 구간 $[t_1, t_2]$에서 태스크 τ_i 의 이상적인 실행시간은 다음과 같다.

$$IR_{\tau_i}(t_1,t_2) = \frac{W(\tau_i)}{\sum_{j \in \Phi} W(\tau_j)} \times (t_2 - t_1) \times M \quad (7)$$

여기서 Φ 는 시스템 전체의 실행 가능한 태스크들의 집합을 의미하고 M 은 CPU 의 개수를 나타낸다.

반면 CFS 는 임의의 구간 $[t_1, t_2]$에서 태스크 τ_i 에게 다음의 실행시간을 할당한다.

$$R_{\tau_i}(t_1,t_2) = \frac{W(\tau_i)}{\sum_{j \in \varphi_i} W(\tau_j)} \times (t_2 - t_1) \quad (8)$$

여기서 φ_i 는 τ_i 가 속한 실행큐 내의 태스크들의 집합을 의미한다. 계산의 단순화를 위해 (t_2-t_1)은 P 의 배수라고 가정한다.

CFS 가 멀티코어에서 공정한 스케줄링을 하기 위해서 모든 실행큐의 load 가 모두 동일해야 한다. 하지만 태스크의 가중치는 양의 자연수로 정의되기 때문에 모든 실행큐의 load 를 항상 동일하게 만드는 것은 현실적으로 불가능에 가깝다. 표 1 은 멀티코어 환경에서 CFS 가 공정성을 보장하지 못하는 태스크 집합의 예를 보인다. 이 예에서 CPU 의 개수는 두 개(P_1 과 P_2)이고 태스크들은 CFS 의 로드 밸런싱 정책에 따라 최대한 균등하게 분포되어 있다. (τ_1 는 P_1 에, 나머지 태스크들은 P_2 에 할당되어있다.) 태스크들은 무한 루프를 수행하는 CPU 바운드 태스크들이다. 이 경우, 조건 (6)이 만족이 되기 때문에 CFS 는 더 이상 로드 밸런싱을 수행하지 않는다. 결과적으로 태스크 τ_1 는 나머지 태스크들보다 3 배 더 높은 가중치를 가졌음에도 불구하고 실제로는 4 배 더 많은 CPU 시간을 할당 받게 된다. 결과적으로 τ_1 의 virtual runtime 은 다른 태스크의 virtual runtime 보다 항상 빠르게 증가하게 되어 공정성 달성에 실패하게 된다.

5. Virtual Runtime 기반 로드 밸런싱

이 장에서 우리는 CFS 가 멀티코어에서도 공정성을 보장할 수 있도록 만들기 위한 virtual runtime 기반 로드 밸런싱을 제안한다.

본 알고리즘의 목적은 임의의 태스크 쌍에 대해서 virtual runtime 의 차이가 상수로 바운드될 수 있도록 태스크를 CPU 에 할당하는 것이다. 수식 (1)에 의해서 태스크들이 자신의 가중치에 비례한 만큼 CPU 시간을 할당 받았다면 태스크들의 virtual runtime 은 동일하다. 따라서 virtual runtime 차이를 상수로 바운드 하는 것은 공정한 스케줄링의 달성을 의미한다.

제안된 로드 밸런싱 알고리즘의 개괄은 그림 1 와 같다. 본 알고리즘은 T 밀리초마다 주기적으로 두 CPU 의 실행큐 내의 태스크 집합을 취합하여 virtual runtime 이 큰 순서대로 정렬한다. 그 후, 제안하는 로드 밸런싱 알고리즘은 PARTITION 과 MIGRATE 을 순차적으로 수행한다.

PARTITION 알고리즘은 정렬된 태스크 집합 X 를 진행이 빠른 (즉, virtual runtime 이 큰) 태스크 그룹 g_{large} 와 진행이 느린 (즉 virtual runtime 이 작은) 태스크 그룹 g_{small} 로 분할된다. 이 때 PARTITION 알고리즘은 다음의 3 가지 특성을 만족시킨다. (1) g_{large} 에 속한 모든 태스크들의 virtual runtime 은 g_{small} 에 속한 태스크들의 그것보다 항상 같거나 크다. (2) g_{large} 의 가중치 합은 g_{small} 의 가중치 합보다 항상 크다. (3) 두 그룹 사이의 가중치 차이는 시스템 내의 가장 큰 가중치보다 크지 않도록 보장한다. 이러한 분할은 로드 밸런싱을 수행한 후에 g_{large} 에 속한 태스크들의 진행을 느리게 하고 g_{small} 에 속한 태스크들의 진행을 빠르게 한다. 또한 어떠한 임의의 두 태스크에 대해서 특정 상수 값 이상 virtual runtime 의 차이가 벌어지지 않도록 보장한다.

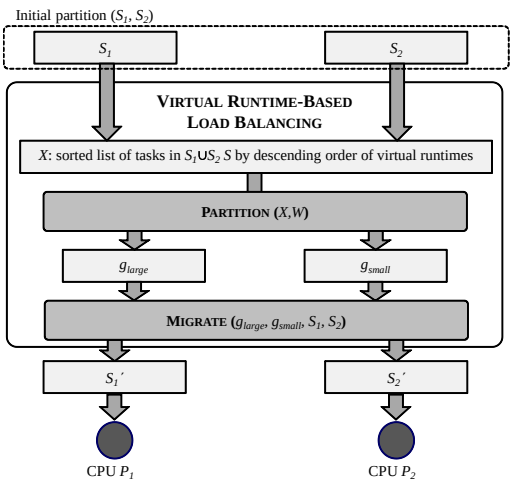

그림 1. Virtual runtime 기반 로드밸런싱의 개괄

Algorithm: VIRTUAL RUNTIME-BASED LOAD BALANCING
inputs: S_1, S_2, W
1 List $X \leftarrow$ sort tasks in $S_1 \cup S_2$ in the descending order of their virtual runtimes
2 $(g_{large}, g_{small}) \leftarrow$ PARTITION(X, W)
3 MIGRATE $(g_{large}, g_{small}, S_1, S_2)$

PARTITION
inputs: X, W
1 $g_{large} \leftarrow \emptyset, g_{small} \leftarrow \emptyset$
2 Integer $u \leftarrow 0$
3 Integer $L \leftarrow$ sum of weights of tasks in X
4 **while** $(u < 1/2 \times L)$
5 $t \leftarrow$ remove the head of X
6 $u \leftarrow u + W(t)$
7 $g_{large} \leftarrow g_{large} \cup \{t\}$
8 $g_{small} \leftarrow$ the set of remaining tasks in X
9 **return** (g_{large}, g_{small})

MIGRATE
inputs: $g_{large}, g_{small}, S_1, S_2$
1 $\delta_{large-1} \leftarrow g_{large} - S_1, \delta_{large-2} \leftarrow g_{large} - S_2$
2 $\delta_{small-1} \leftarrow g_{small} - S_1, \delta_{small-2} \leftarrow g_{small} - S_2$
3 **if** $(|\delta_{large-1} + \delta_{small-2}| > |\delta_{large-2} + \delta_{small-1}|)$ **then**
4 migrate $\delta_{large-2}$ from core P_1 to P_2
5 migrate $\delta_{small-1}$ from core P_2 to P_1
6 else
7 migrate $\delta_{small-2}$ from core P_1 to P_2
8 migrate $\delta_{large-1}$ from core P_2 to P_1
9 **endif**
end

그림 2. 제안하는 알고리즘의 명세

PARTITION 알고리즘을 통해 얻은 g_{large} 와 g_{small} 은 MIGRATE 알고리즘을 통해 각 태스크 집합이 어떤 CPU 에서 수행될지 결정된다. MIGRATE 알고리즘은 태스크 이주 횟수를 최소화할 수 있도록 태스크 이주를 수행시킨다. 그림 2 은 제안된 알고리즘을 상세히 설명한다.

상기 알고리즘은 2 개의 CPU 들을 위한 로드 밸런싱을 지원한다. 우리는 제안된 알고리즘이 N 개의 CPU 들을 지원할 수 있게 하기 위해서 CPU 들을 로드 밸런싱 시점마다 교차적으로 짝을 짓는다. 구체적으로, $k\lambda$ 일 때와 $(k+1)\lambda$ 일 때 다르게 짝을 지어 N 개의 CPU 에서도 어떠한 두 태스크 쌍의 virtual runtime 의 차이가 상수로 바운드 될 수 있게 한다.

6. 실험 및 검증 결과

본 장에서는 제안된 로드 밸런싱 알고리즘의 공정성을 평가하기 위해서 일련의 실험을 보인다. 또한 제안하는 알고리즘의 로드 밸런싱 주기 T 조정에 의

표 2. 타깃 시스템의 하드웨어/소프트웨어 명세

Hardware	***CPU***	Intel® Core™ 2 Duo E8500 Clock: 3.16GHz
	Main memory	4-GB DDR2
Software	***Operating system***	Ubuntu 11.04 Kernel version: 2.6.38.8
	GNU libc	glibc 2.13

한 공정성과 런타임 오버헤드의 trade-off 를 보인다.

우리는 제안하는 로드 밸런싱 알고리즘을 Linux 커널 2.6.38.8 상에 구현했다. 표 2 는 본 알고리즘을 실험하고 검증한 하드웨어 및 소프트웨어 타깃 시스템을 보인다.

제안하는 방법의 공정성을 평가하기 위해, 6 개의 CPU 바운드 태스크들로 구성된 태스크 집합을 실행시켰다. 이들의 nice 값은 -5, 5, 0, 0, 0 그리고 0 으로 설정했다. 우리는 이 태스크 집합의 virtual runtime 의 최대 차이 $D_{max}(t)$를 측정했다.

또한 런타임 오버헤드의 측정을 위해, Kernbench 벤치마크 프로그램[11]을 사용했다. 이 프로그램은 복수의 태스크를 생성하여 Linux 커널 소스를 병렬적으로 컴파일 한다. 이를 위해 태스크의 수는 6 개, nice 값은 모두 0 으로 설정했다. 이러한 환경에서 컴파일 하는 데 소요되는 시간을 측정하여 기존 CFS 와 비교함으로써 런타임 오버헤드를 측정했다.

그림 3 은 기존 CFS 와 제안하는 알고리즘을 적용한 CFS 의 $D_{max}(t)$를 비교한 결과이다. 가로 축은 시간을, 세로 축은 $D_{max}(t)$를 의미한다. 실험 결과에서 명백히 드러나듯 기존 CFS 의 경우 $D_{max}(t)$는 시간에 지남에 따라 발산하는 반면, 제안하는 알고리즘의 경우 $D_{max}(t)$는 작은 상수에 바운드 된다. 본 실험에서 우리는 로드 밸런싱 주기 T 를 100, 500, 1000 밀리초로 바꾸어 실험하여 T 가 작을 때 더 높은 공정성을 보장함을 확인했다. T=100 일 때와 T=1000 일 때 $D_{max}(t)$는 각각 50.54 와 387.08 단위시간으로 바운드 된다. 여기서 1 단위시간은 nice 값 0 의 태스크의

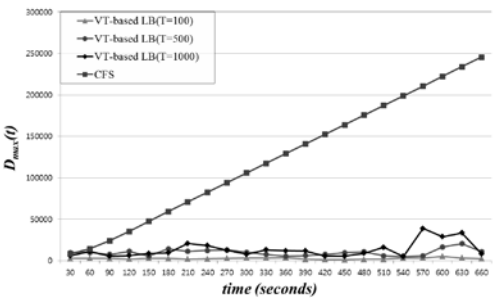

그림 3. 제안하는 Virtual runtime 기반 로드밸런싱 알고리즘과 기존 CFS와의 공정성 비교

표 3. Virtual runtime 기반 로드밸런싱 알고리즘의 런타임 오버헤드 비교

	Legacy CFS	***New CFS (T=100)***	***New CFS (T=500)***	***New CFS (T=1000)***
Compilation elapsed time (s)	48.678	48.748	48.442	48.348

실행시간으로 환산하면 1 밀리초에 해당한다.

제안하는 알고리즘의 런타임 오버헤드는 표 3 에 나타나있다. 실험 결과에서 볼 수 있듯, 제안하는 알고리즘의 성능은 기존 CFS 와 거의 유사했으며 야기되는 런타임 오버헤드는 고작 0.14%에 불과했다.

7. 결론

본 논문은 멀티코어 환경에서 공정성 향상을 위한 virtual runtime 기반 로드 밸런싱 알고리즘을 제안했다. 이를 위해 우리는 리눅스 CFS 의 동작원리를 분석하여 CFS 가 멀티코어에서 공정성의 측면에서 문제점이 있다는 것을 밝혔다. 제안하는 알고리즘은 멀티코어 환경에서 모든 태스크 쌍에 대해서 virtual runtime 의 차이를 상수로 바운드한다. 우리는 실험 및 검증을 통해 제안한 알고리즘이 멀티코어에서 공정성을 향상시킴을 보였다.

참 고 문 헌

[1] Molnar. The Completely Fair Scheduler,
[2] http://people.redhat.com/mingo/cfs-scheduler/.
[3] http://www.mattheaton.com/?p=222.
[4] 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.
[5] A. Srinivasan, and J.H. Anderson, "Fair scheduling of dynamic task systems on multiprocessors", Journal of Systems and Software, 2005, 77, (1), pp. 67-80.
[6] B. Caprita, W.C. Chan, J. Nieh, C. Stein, and H. Zheng, "Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems". Proc. the annual conference on USENIX ATC 2005, pp. 337-352.
[7] B. Caprita, J. Nieh, and C. Stein, "Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling". Proc. the twenty-fifth annual ACM symposium on Principles of distributed computing 2006, pp. 72-81.
[8] http://people.redhat.com/mingo/O(1)-scheduler/README
[9] T. Li, D. Baumberger, and S. Hahn, "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin", ACM SIGPLAN Notices, 2009, 44, (4), pp. 65-74.
[10] http://www.hpl.hp.com/research/linux/kernel/o1-starve.php
[11] http://ck.kolivas.org/kernbench/