리눅스 기반 멀티코어 시스템에서 공정성 향상을 위한 Virtual Runtime 밸런싱 알고리즘

현진화 ○, 노순현, 박대동, 홍성수
서울대학교 전기컴퓨터공학부
{jhhyeon○, shnoh, sshong}@redwood.snu.ac.kr

Virtual Runtime Balancing Algorithm for Improving Fairness on Linux-based Multicore Architecture

Jinhwa Hyeon○, Soonhyun Noh, Daedong Park, Seongsoo Hong
Department of Electrical and Computer Engineering, Seoul National University

요 약

클라우드 서버 시장이 증가함에 따라 클라우드 서버 시스템이 요구하는 공정성 보장을 위해 리눅스의 공정 스케줄러인 CFS가 널리 이용되고 있다. 하지만 CFS의 로드 밸런싱 알고리즘은 멀티코어 환경에서 태스크간 공정성을 보장해주지 못하는 문제가 있다. 그 동안 이러한 CFS의 한계점을 개선하기 위한 로드 밸런싱 알고리즘이 많이 제안되어 왔다. 본 논문에서는 기존 연구들을 분석하고 기존 연구들의 한계점을 규명한다. 또한 기존 연구들의 한계점을 개선하기 위해 임의의 두 태스크간 virtual runtime의 차이를 상수로 바운드시키는 로드 밸런싱 알고리즘을 제안한다. 제안하는 로드 밸런싱 알고리즘은 주기적으로 동작하며 로드가 작은 코어일수록 코어에 존재하는 태스크들의 virtual runtime 중 최솟값이 작아지도록 태스크를 이주시킨다. 우리는 제안된 알고리즘을 리눅스 커널 상에 구현하고 기존 CFS의 로드 밸런싱 기법, 기존 연구의 로드 밸런싱 알고리즘과 비교하는 실험을 진행하였다. 그 결과 기존 CFS로 스케줄링되는 태스크들의 virtual runtime 차이는 무한대로 발산하는데 비해 제안된 알고리즘으로 스케줄링되는 태스크들의 virtual runtime 차이는 0.2초 이내로 바운드되었고, 기존 연구의 로드 밸런싱 알고리즘에 비해 로드 밸런싱마다 이주되는 태스크 횟수의 평균이 87.2% 줄어들었다.

1. 서 론

공정 스케줄러는 각 태스크에게 태스크의 가중치(weight)에 비례하게 CPU 자원을 할당하는 스케줄러이다. 이러한 스케줄러는 각 태스크에게 일정량 이상의 자원 요구사항을 보장할 수 있기 때문에 성능 고립화(performance isolation)가 중요한 클라우드 서버에서 주로 이용된다.

그 동안 멀티코어 시스템에서 공정성을 보장하기 위한 공정 스케줄링 알고리즘들이 많이 제안되었다. 이 알고리즘들은 사용하는 실행큐 구조에 따라 크게 두 종류로 분류된다. 하나는 전역 실행큐 알고리즘이고 다른 하나는 분산 실행큐 알고리즘이다. 전역 실행큐 알고리즘은 시스템 전체에 단 하나의 실행큐를 가지고 태스크들을 스케줄링하는 알고리즘이다. 이러한 알고리즘들[1][2][3]은 코어의 개수가 적은 규모가 작은 시스템에서는 잘 동작하지만 코어의 개수가 많아질수록 동기화, lock contention 등의 문제로 인해 성능 저하가 극심해지는 단점이 있다.

분산 실행큐 알고리즘은 코어마다 하나의 실행큐를 가지고 태스크들을 스케줄링하는 알고리즘이다. 이 분류에 속하는 대부분의 알고리즘들은 개별 코어 스케줄링 알고리즘을 이용하여 각 실행큐에서 공정성을 달성하고, 멀티코어 환경에서 공정성을 달성하기 위해 가중치 기반의 로드 밸런싱을 사용한다. 리눅스의 CFS는 대표적인 가중치 기반의 로드 밸런싱을 이용하는 알고리즘이다. CFS는 모든 태스크의 virtual runtime이 같아지도록 스케줄링 함으로써 공정성을 보장한다. Virtual runtime이란 어떤 태스크가 할당 받은 누적 CPU 시간을 그 태스크의 가중치로 나눈 값이다. CFS는 가중치 기반 로드 밸런싱 알고리즘의 한계점으로 인해 임의의 태스크 쌍에 대해 virtual runtime 차이가 무한대로 발산하는 경우가 발생한다[4].

[4]는 CFS의 로드 밸런싱 알고리즘을 개선한 분산 실행큐 알고리즘이며 임의의 태스크 쌍에 대해 virtual runtime 차이를 상수로 바운드시킨다. 하지만 [4]에서 제안된 알고리즘은 태스크의 개수가 증가함에 따라 로드 밸런싱마다 발생하는 태스크 이주횟수가 증가하기 때문에 cache miss 등의 오버헤드가 증가한다. 본 연구에서는 임의의 태스크 쌍에 대해 virtual runtime 차이를 상수로 바운드 시킴과 동시에 로드 밸런싱 알고리즘이 동작할 때 발생하는 태스크 이주 횟수를 코어의 개수의 제곱에 바운드시키는 알고리즘을 제안한다.

이 논문의 나머지 부분은 다음과 같이 구성된다. 2장에서는 본 연구에서 풀고자 하는 문제를 정의한다. 3장에서는 정의된 문제에 대한 솔루션을 제안한다. 4장에서는 제안된 솔루션을 실험적으로 검증한다. 마지막으로 5장에서는 본 연구의 결론을 내린다.

2. 문제 정의

본 연구에서는 임의의 태스크 쌍에 대해 virtual runtime 차이를 상수로 바운드시키는 로드 밸런싱 알고리즘을 제안한다. 즉, 아래의 식을 만족시키는 로드 밸런싱 알고리즘을 제안한다.

$$\max_{\tau_i,\tau_j \in T}\{|VR(\tau_i) - VR(\tau_j)|\} \le C \quad (1)$$

T는 시스템에 존재하는 모든 태스크들의 집합을 의미하고, τ_i, τ_j는 태스크를 나타낸다. $VR(\tau_i)$는 τ_i의 virtual runtime을

의미한다. C는 임의의 상수를 의미한다.

제안된 로드 밸런싱 알고리즘은 동작 시 발생하는 태스크 이주의 횟수가 코어의 제곱에 바운드되어야 하며, 임의의 두 코어 간 로드의 차이가 코어의 개수에 바운드되어야 한다. 임의의 두 코어 간 로드의 차이가 코어의 개수에 바운드되어야 하는 이유는 어떤 태스크가 다른 태스크들에 비해 매우 빠르거나 매우 느리게 동작하지 않게 하기 위함이다.

3. Virtual Runtime 밸런싱

이 장에서는 3장에서 다룬 문제에 대한 솔루션으로 virtual runtime 밸런싱 알고리즘을 제안한다. 제안하는 알고리즘은 로드가 작은 코어일수록 코어에 존재하는 태스크들의 virtual runtime 중 최솟값이 작아지도록 태스크들을 이주시킨다. 이를 통해 식 (1)을 만족하는데, 개별 코어에서 동작하는 스케줄러는 virtual runtime이 가장 작은 태스크를 수행하고, 로드가 작은 코어일수록 코어에 존재하는 태스크들의 virtual runtime 증가율이 높기 때문이다. 또한 제안된 알고리즘은 virtual runtime이 가장 작은 태스크들을 코어의 개수만큼 선택하여 코어에 분배 함으로써 태스크 이주 횟수를 코어의 제곱에 바운드시킨다.

ALGORITHM: VIRTUAL RUNTIME-BASED LOAD BALANCING

Input:
A set of tasks in a system: $T = \{\tau_1, \tau_2, \ldots, \tau_m\}$
A set of cores in a system: $C = \{c_1, c_2, \ldots, c_n\}$
The number of cores: n

```
1:  H ← SORTBYVR(T)
2:  U ← sub-sequence of H with the first n tasks
3:  H ← T − H
4:  Load [1:n] ← {load of c_1, load of c_2, ..., load of c_n}
5:  i ← 1
6:  while i ≤ n do
7:      Load [core of i-th task in U] ← Load [core of i-th task in U] – weigh
        t of  i-th task in
8:      i ← i + 1
9:  end while
10: D ← SORTBYLOAD(C, Load )
11: H ← SORTBYCORE(H, D)
12: G ← PARTITION(H, n)
13: G ← REVISE(G, U)
14: while i ≤ n do
15:     Assign i-th task group in G to i-th core in D
16:     Assign i-th task in U to i-th core in D
17: end while
18: return
```

그림 1. Virtual Runtime 밸런싱 알고리즘

제안된 알고리즘은 1초마다 주기적으로 수행되며 동작은 그림 1과 같다. 알고리즘에 대한 입력은 태스크의 집합 T, 코어의 집합 C, 코어의 개수 n이다. 먼저 virtual runtime이 가장 작은 n개의 태스크와 n개를 제외한 나머지 태스크들을 각각 U, H에 저장한다(line 1~3). 다음으로 U에 속한 태스크들을 제외하고 각 코어의 로드를 계산한 후, 코어들을 계산된 로드의 오름차순으로 정렬하여 D에 저장한다(line 4~10). 다음으로 D의 순서대로 코어에 존재하는 태스크들 중 H에 속한 태스크들을 나열한다(line 11). 다음으로 각 코어에 분배할 n개의 task group을 만들어 G에 저장한다(line 12, 13). 이 때, G는 아래의 두 가지 성질을 만족하도록 생성된다.

(1) 인접한 task group간 로드의 차이는 $2W_{max}$로 바운드된다.
(2) i번째 task group의 로드와 U의 i번째 태스크의 가중치의 합은 $i+1$번째 task group의 로드와 U의 $i+1$번째 태스크의 가중치의 합보다 작거나 같다.

Task group의 로드는 task group에 속한 태스크들의 가중치의 합을 의미하고, W_{max}는 태스크에게 부여할 수 있는 가중치의 최댓값을 의미한다.

마지막으로 G의 i번째 task group에 속한 태스크들과 U의 i번째 태스크를 D의 i번째 코어에 분배한다(line 14~17). 이 때 발생하는 태스크 이주 횟수는 $\frac{3W_{max}}{W_{min}}n^2$으로 바운드된다. W_{min}은 태스크에게 부여할 수 있는 가중치의 최솟값을 의미한다.

G를 생성하기 위한 알고리즘인 `partition()`, `revise()` 알고리즘의 동작은 각각 그림 2, 3과 같다.

ALGORITHM: PARTITION

Input:
A sequence of tasks: $H = \{\tau_1, \tau_2, \ldots, \tau_m\}$
The number of cores: n

```
1:  g_1 ← ∅, g_2 ← ∅, ..., g_n ← ∅
2:  E ← sum of weights of tasks in H
3:  i ← 1
4:  j ← 1
5:  while i ≤ n do
6:      expected_load ← E/(n − i + 1)
7:      while load of g_i + weight of τ_j ≤ expected_load do
8:          g_i ← g_i ∪ τ_j
9:          j ← j + 1
10:     end while
11:     E ← E − load of g_i
12:     i ← i + 1
13: end while
14: return {g_1, g_2, ..., g_n}
```

그림 2. `Partition()` 알고리즘

`partition()`알고리즘은 입력으로 태스크의 집합 H와 코어의 개수 n을 받아 코어의 개수만큼 task group을 생성한다. 이 때, 인접한 task group간 로드의 차이는 $2W_{max}$로 바운드된다. 먼저 $i=1$부터 $i=n$까지 루프를 수행하면서 task group g_i에 태스크를 할당한다(line 5~13). 태스크를 할당하기 전에 먼저 g_i, g_{i+1}, …, g_n에 할당될 로드의 평균을 계산하고(line 6), g_i의 로드가 계산된 값을 초과하지 않을 때까지 g_i에 태스크를 할당한다(line 7~10).

ALGORITHM: REVISE

Input:
A sequence of task groups: $G = \{g_1, g_2, \ldots, g_n\}$
A sequence of tasks: $U = \{\tau_1, \tau_2, \ldots, \tau_n\}$
The number of cores: n

```
1:  i ← 1
2:  while i < n do
3:      _REVISE_(G, U, i)
4:  end while
5:  return G

_REVISE_(G, U, i)
1:  if load of g_i + weight of τ_i > load of g_{i+1} + weight of τ_{i+1} then
2:      while load of g_i + weight of τ_i > load of g_{i+1} + weight of τ_{i+1} do
3:          move a task in g_i to g_{i+1}
4:      end while
5:      if i ≠ 1 then
6:          _REVISE_(G, U, i − 1)
7:      end if
8:      if i ≠ n − 1 then
9:          _REVISE_(G, U, i + 1)
10:     end if
11: end if
12: return
```

그림 3. `Revise()` 알고리즘

`revise()`알고리즘은 입력으로 원소의 개수가 코어의 개수

와 같은 태스크 그룹의 집합$\{g_1, g_2, \cdots, g_n\}$과 태스크의 집합 $\{\tau_1, \tau_2, \cdots, \tau_n\}$, 코어의 개수 n을 받아 g_i의 로드와 τ_i의 가중치의 합이 g_{i+1}의 로드와 τ_{i+1}의 가중치의 합보다 작도록 $\{g_1, g_2, \cdots, g_n\}$에 존재하는 태스크를 재분배한다. 이 알고리즘은 $i = 1$부터 $i = n - 1$까지 루프를 수행하면서 `_revise_()`를 호출한다(line 2~4). `_revise_()`함수는 g_i의 로드와 τ_i의 가중치의 합이 g_{i+1}의 로드와 τ_{i+1}의 가중치의 합보다 작거나 같아질 때까지 g_i에서 g_{i+1}로 태스크를 하나씩 옮긴다(line 2~4). `revise()`알고리즘으로 조정된 G에서 인접한 task group간 로드의 차이는 $2W_{max}$로 바운드된다.

4. 실험 및 검증 결과

본 장에서는 제안된 로드 밸런싱 알고리즘이 3장의 문제에 대한 솔루션임을 실험적으로 검증한다. 또한 제안된 로드 밸런싱 알고리즘의 오버헤드를 측정한다. 실험을 위해 우리는 제안된 로드 밸런싱 알고리즘을 리눅스 커널 4.5 상에 구현하였다. 표 1은 시스템의 하드웨어 및 소프트웨어 명세를 보인다. 또한 워크로드는 SPEC CPU 2006 benchmark를 이용하였다. 이 benchmark는 single-threaded benchmark로서 리눅스에서 태스크 한 개로 수행되며 가중치는 1024로 설정하였다.

표 1. 타깃 시스템의 하드웨어/소프트웨어 명세

하드웨어	CPU	Intel® Core™ i7-4770 Clock: 3.4GHz
	메인 메모리	8-GB DDR3
소프트웨어	운영체제	Ubuntu 14.04 Kernel version: 4.5

그림 4는 임의의 두 태스크 간 virtual runtime 차이가 상수로 바운드되는지 검증하는 실험에 대한 결과이다. 100개의 태스크를 수행하면서 10초 간격으로 시스템에 존재하는 태스크들의 virtual runtime의 최댓값과 최솟값의 차이를 측정하였다. 기존 CFS 로드 밸런싱으로 스케줄링하는 경우에는 시간에 따라 값이 증가하는 반면, virtual runtime 밸런싱으로 스케줄링하는 경우에는 0.2초 이내로 바운드 되었다.

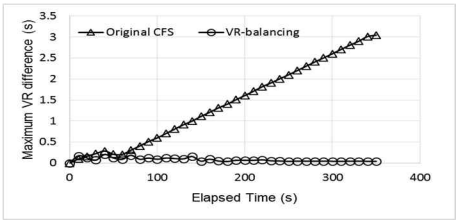

그림 4. 기존 CFS 로드 밸런싱과 제안된 알고리즘 간 maximum virtual runtime difference 비교

표 2는 시스템에 존재하는 태스크의 개수와 로드 밸런싱 때마다 일어나는 태스크 이주 횟수 간 관계를 보이는 실험에 대한 결과이다. 태스크 개수가 50개일 때, 100개일 때, 200개일 때 실험하였으며 각각을 수행하는 동안 일어나는 태스크 이주 횟수와 로드 밸런싱 횟수를 측정하였다. 결과적으로 태스크 개수가 100개일 때 로드 밸런싱마다 일어나는 태스크 이주 횟수가 가장 적고, 태스크 개수가 50개일 때 가장 크게 측정되었다. 따라서 로드 밸런싱마다 일어나는 태스크 이주 횟수는 시스템에 존재하는 태스크 개수에 비례하지 않는다.

표 2. 시스템에 존재하는 태스크 개수에 따른 virtual runtime 밸런싱 알고리즘이 동작할 때 일어나는 태스크 이주 횟수

태스크 개수	50	100	200
태스크 이주 횟수 (A)	3176	3609	10392
로드 밸런싱 횟수 (B)	174	386	795
A / B	18.25	9.35	13.07

표 3은 [4]에서 제안된 알고리즘과 virtual runtime 밸런싱 알고리즘이 동작할 때 일어나는 태스크 이주 횟수를 비교한 실험에 대한 결과이다. 사용된 워크로드는 100개의 태스크로 이루어져 있으며 [4]에서 제안된 알고리즘은 dual-core 시스템에서만 동작하므로 dual-core 시스템에서 실험이 진행되었다. [4]에서 제안된 알고리즘을 적용하였을 때에 비해 virtual runtime 밸런싱 알고리즘을 적용할 때 로드 밸런싱마다 일어나는 로드 밸런싱 횟수의 평균이 87.2% 줄어들었다.

표 3. [4]에서 제안된 알고리즘과 virtual runtime 밸런싱 알고리즘이 동작할 때 일어나는 태스크 이주 횟수 비교

로드 밸런싱 알고리즘	Existing algorithm	VR balancing
태스크 이주 횟수 (A)	5353	681
로드 밸런싱 횟수 (B)	644	641
A / B	8.31	1.06

표 4는 virtual runtime 밸런싱 알고리즘의 오버헤드 측정에 관한 실험에 대한 결과이다. 기존 CFS 로드 밸런싱 알고리즘과 virtual runtime 밸런싱 알고리즘으로 100개의 태스크로 이루어진 워크로드를 수행하는데 걸리는 시간을 측정하였다. Virtual runtime 밸런싱 알고리즘을 적용하였을 때 기존 CFS 로드 밸런싱 알고리즘을 적용하였을 때보다 1.61% 긴 시간이 소요되었다.

표 4. 기존 CFS 로드 밸런싱 알고리즘과 virtual runtime 밸런싱 알고리즘과의 태스크 개수가 100개인 워크로드를 수행하는 데 걸리는 시간 비교

로드 밸런싱 알고리즘	Original CFS	VR balancing
동작 시간(s)	380.86	386.979
오버헤드(%)	-	1.61

5. 결론

본 연구에서 우리는 리눅스 기반 멀티코어 시스템에서 공정성을 보장하기 위한 로드 밸런싱 알고리즘을 제안하였다. 제안된 알고리즘은 로드가 작은 코어일수록 코어에 존재하는 태스크들의 virtual runtime들 중 최솟값이 작아지도록 로드 밸런싱 함으로써 시스템에 존재하는 임의의 태스크 쌍에 대해 virtual runtime의 차이를 상수로 바운드 시키고 로드 밸런싱마다 일어나는 태스크 이주 횟수를 코어의 제곱으로 바운드 시킨다. 실험을 통해 시스템에 존재하는 태스크들의 virtual runtime의 최댓값과 최솟값의 차이가 0.2초 이내로 바운드됨을 검증하였으며 태스크 개수가 증가하더라도 로드밸런싱마다 일어나는 태스크 이주 횟수가 증가하지 않음을 보였다. 또한 제안된 알고리즘으로 인해 발생하는 오버헤드는 1.61%임을 보였다.

사사

본 연구는 미래창조과학부 및 정보통신기술진흥센터의 대학 ICT연구센터육성 지원사업의 연구결과로 수행되었음(IITP-2016-(H8501-16-1015))

참고문헌

[1] 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.

[2] 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.

[3] C. Kolivas. BFS scheduler. http://ck.kolivas.org/patches/bfs/bfs-faq.txt. (2010)

[4] Huh, S., Yoo, J., Kim, M., and Hong, S. "Providing Fair Share Scheduling on Multicore Cloud Servers via Virtual Runtime-based Task Migration Algorithm". Accepted to 32nd International Conference on Distributed Computing Systems, 2012