하이퍼스레딩 환경에서의 저장장치 입출력 폴링 태스크 배치 기법

김성윤[0] 하성준 안민우 정진규
성균관대학교 정보통신대학
ksungyun0584@gmail.com, seongjun6608@gmail.com, mwahn402@csi.skku.edu, jinkyu@skku.edu

I/O Polling Task Placement on Hyperthreading-enabled Multicore CPU

Sungyun Kim[0] Seongjun Ha Minwoo Ahn Jinkyu Jeong
College of Information and Communication Engineering, Sungkyunkwan University

요 약

최근 저장장치들의 고성능화로 인해 입출력 요청의 병목 지점이 장치 내부 처리 시간이 아닌 커널 처리 시간으로 옮겨졌다. 이에 따라, 커널에서의 입출력 방식으로 인터럽트 방식이 아닌 busy-waiting 기반의 폴링 방식을 활용하고 있다. 폴링 태스크는 입출력 요청의 완료를 대기하며 잠들지 않고, 완료를 확인하기 위한 메모리 참조를 반복적으로 수행한다. 따라서, 폴링 태스크들은 CPU 자원을 지속적으로 사용하지만 내부적으로 연산 유닛을 거의 활용하지 않으며, 하이퍼스레딩 환경에서 각 물리적 코어에 주어진 연산 유닛을 효율적으로 활용하기 위한 태스크 배치가 필요하다. 본 논문에서는, 폴링 태스크와 연산 중심의 태스크들이 동시배치되는 상황을 고려하여 실험적인 최적의 배치를 찾아내고, 두 종류의 태스크들이 시스템에 주어진 CPU 자원 및 연산 유닛을 효율적으로 사용하는 태스크 배치 기법을 제안한다.

1. 서 론

최근 삼성 Z-SSD, 인텔 Optane SSD와 같은 고성능 저장장치들이 등장하면서, 입출력 요청의 병목 지점이 장치 내부 처리 시간이 아닌 커널 처리 시간으로 옮겨졌다. 이에 따라, 커널에서의 입출력 방식으로 인터럽트 방식이 아닌 busy-waiting 기반의 폴링 방식을 활용하는 연구들이 진행되고 있다 [1]. 폴링 태스크들은 입출력 요청의 완료를 확인하기 위하여 잠들지 않고 CQ(Completion Queue) 메모리 영역을 반복적으로 참조한다는 특징을 가지고 있다. 따라서, 폴링 방식은 CPU 자원을 지속적으로 사용하지만, 연산 유닛을 활용하지 않는다.

하이퍼스레딩 [2] 기술은 인텔의 동시 멀티스레딩 기술로써, 물리적 코어 하나에 논리적 코어 두 개를 할당해 CPU의 시간당 처리량을 최대화하는 기술이다. 각 논리적 코어는 동시에 명령어들을 패치할 수 있지만 물리적 코어에 주어진 ALU와 FPU와 같은 연산 유닛들은 공유하기 때문에 서로 다른 연산 중심의 두 프로그램이 같은 물리적 코어에 배치되어 실행된다면 각 프로그램의 하이퍼스레딩 성능은 감소하게 된다.

하지만, 연산 중심 태스크와 폴링 태스크가 같이 물리적 코어에 배치되는 경우라면 성능 증가를 기대할 수 있다. 폴링 태스크의 경우 지속적인 CPU 자원을 활용하지만 반복적인 메모리 참조만 수행함으로써 연산 유닛 사용량은 적다. 따라서, 같은 물리적 코어의 서로 다른 논리적 코어에 폴링 태스크들이 배치되어서 실행된다면 CPU를 지속적으로 점유하고 있지만, 연산 유닛에 대한 자원 낭비가 발생하게 된다. 반대로, 연산 중심 태스크들이 같은 물리적 코어에 배치되어서 실행된다면, 태스크들 모두 연산 유닛을 활용하려 하며, 연산 명령의 순차화에 의한 병목현상이 발생하게 된다. 결론적으로, 연산 중심 태스크의 성능 이득을 기대할 수 있는 배치는 같은 물리적 코어에 연산 중심 태스크와 폴링 태스크가 배치되어 실행되는 경우이다.

기존의 리눅스 로드밸런서의 경우 각 태스크의 특성을 고려하지 않고, 시스템의 전체적인 CPU 로드를 태스크들의 마이그레이션을 통하여 균등화하는 방식으로 동작하고 있다. 따라서, 본 논문에서는 연산 중심 태스크와 폴링 태스크들이 동시 배치되어 실행되는 상황에서 2비트 폴링 예측기를 통하여 폴링 태스크들을 구분하고, 폴링 태스크의 특성을 고려한 새로운 로드밸런서를 제시한다. 본 로드밸런서는 시스템의 논리적 코어의 개수보다 많은 태스크들에 대해서도 고려하고 있다.

이 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(NRF-2017R1C1B2007273, NRF-2016M3C4A7952587)

2. 태스크 배치에 따른 성능 분석

이 장에서는 연산 중심의 태스크(이하 C 태스크)와 폴링 태스크(이하 P 태스크)들을 서로 다르게 배치함에 따라 성능 변화를 측정하고, 주어진 상황에서 최적의 배치를 하기 위한 규칙들을 제시한다. 실험에 사용된 P 태스크는 Intel SSD DC P3700에 입출력을 요청하도록 fio 벤치마크를 사용하였으며, C 태스크는 SPEC_CPU2017 (INT, FLOAT) 벤치마크를 사용하였다. 실험에 사용된 CPU는 4개의 물리적 코어가 각각 2개씩의 논리적 코어를 지원하는 Intel Xeon CPU E3-1270 v5를 사용하였다.

각 C 태스크들과 P 태스크들의 배치에 따라 성능 변화를 측정하기 위해 총 세 종류의 배치를 측정하였으며, SPEC_CPU의 경우 2개의 INT 워크로드(perlbench, x264)와 2개의 FLOAT 워크로드(cactuBSSN, wrf)를 사용하였다 [3].

배치에 대한 설명은 그림 1과 같다. Free 배치는 기존의 리눅스 로드밸런서에 의한 태스크 배치가 이루어지는 경우이며, Bunch와 Scatter 배치는 C 태스크 및 P 태스크를 각 물리적 코어에 같이 배치하는 경우와 따로 배치하는 경우를 의미하며, Colocate와 Spread 배치는 물리적 코어 내부의 각 논리적 코어에 같이 배치하는 경우와 따로 배치하는 경우를 의미한다.

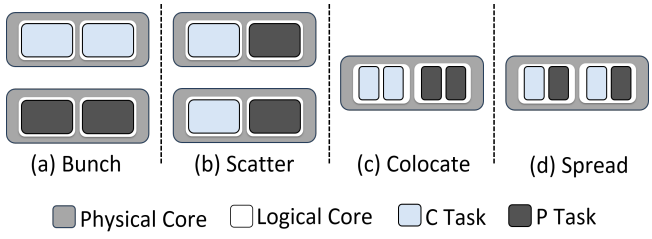

그림 1 태스크 배치 종류

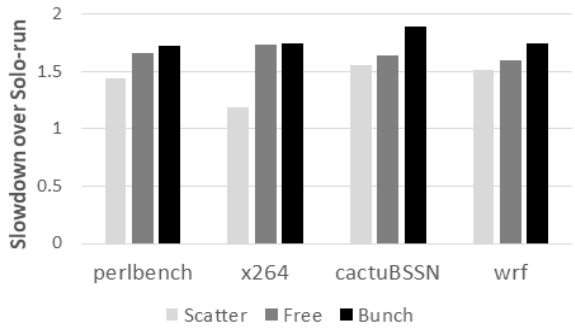

그림 2 물리적 코어 간 배치의 C 태스크 성능 결과

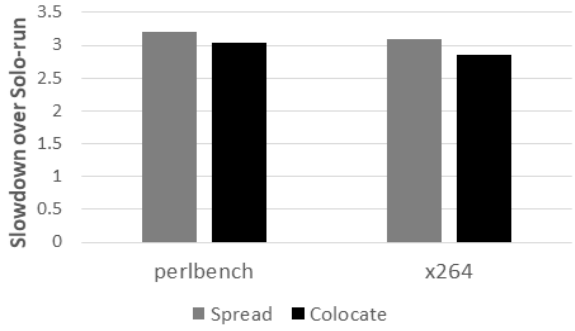

그림 3 물리적 코어 내부 배치시 C 태스크 성능 결과

2.1. 물리적 코어 간의 배치

이 장에서는 물리적 코어 간의 C 태스크와 P 태스크의 배치에 따른 성능 차이를 분석한다. 실험은 2개의 코어를 사용하여 총 4개의 태스크(2개의 C 태스크와 2개의 P 태스크)를 동시에 구동하며 성능을 측정하였다. 태스크의 배치는 Bunch (그림 1-a), Scatter(그림 1-b), Free(로드밸런서에 의한 배치) 이렇게 세가지 경우를 실험하였다. C 태스크는 총 4가지 워크로드(perlbench, x264, cactuBSSN, wrf)를 각 케이스별로 구동하였다.

그림 2는 C 태스크의 실행시간을 단일 C 태스크에서 측정한 실행시간으로 표준화한 결과이다. 그림 2를 보면 Bunch 배치에 비해 Scatter 배치에서 최대 약 21.9%, 평균 17%의 성능 증가가 있다. 또한, Free 배치가 Scatter 배치보다 C 태스크의 성능이 감소하는데, 이는 리눅스 로드밸런서가 각 태스크의 특성을 고려하지 않고 마이그레이션을 하기 때문이다.

P 태스크는 각 배치 별로 입출력 처리량(bandwidth)과 응답시간(latency) 값의 표준 편차는 1% 미만의 결과를 얻었다. 즉 P 태스크의 배치는 P 태스크의 성능과 큰 영향이 없는 것을 확인하였다.

2.2. 논리적 코어간(물리적 코어 내부)의 배치

이 장에서는 물리적 코어 내부의 C 태스크와 P 태스크의 배치에 따른 성능 차이를 분석한다. 실험은 하나의 물리적 코어를 활용하며, C 태스크와 P 태스크를 각각 두 개씩 실행하여 Colocate(그림 1-c)와 Spread(그림 1-d) 배치의 성능 차이를 측정하였다.

그림 3은 C 태스크의 실행 시간을 단일 C 태스크의 실행시간에 대하여 표준화한 결과이다. 실험 결과, Colocate 배치가 Spread 배치보다 평균 9.09%의 성능 증가를 확인할 수 있으며, 이는 Colocate 배치의 경우 C 태스크와 P 태스크가 하나의 물리적 코어 내부 실행 유닛을 공유하지만 사용하는 유닛이 덜 겹쳐 경합이 상대적으로 작기 때문이다.

P 태스크는 이전 실험과 유사하게 배치에 상관없이 입출력 처리량과 응답시간이 동일하게 측정되었다.

2.3. 최적의 태스크 배치

물리적 코어 간의 태스크 배치 실험에서는 Scatter 배치의 경우, 논리적 코어 간(물리적 코어 내부)의 태스크 배치 실험에서는 Colocate 배치의 경우가 성능저하를 최소화 함을 알 수 있었다. 물리적 코어끼리는 C와 P 태스크가 분산 될수록, 같은 물리적 코어의 논리적 코어끼리는 모여 있을수록 성능이 제일 높다. 실험 결과를 통해 C 태스크와 P 태스크가 동시 배치되는 경우, 최적의 배치를 위한 규칙을 아래와 같이 정리할 수 있다.

1) P 태스크는 어떤 태스크와 같은 코어에 배치되어도 입출력 처리량과 응답속도의 변화는 없다.
2) C 태스크의 경우 서로 다른 물리적 코어에 분배되어 실행되어야 한다.
3) C 태스크의 경우 한 물리적 코어 안에서는 같은 논리적 코어에 모여 실행되어야 한다.

3장에서는 발견한 규칙들에 기반한 태스크 배치를 수행하는 CPU 로드밸런서를 제안한다.

3. 로드밸런서 설계

이 장에서는 실험 결과를 통해 세운 규칙들을 바탕으로 태스크들의 특성을 고려한 CPU 로드밸런서 설계를 제시한다. 제시하는 로드밸런서는 2비트 폴링 예측기를 통하여 P 태스크를 구분하고, 각 논리적 코어 별 런큐(runqueue)의 P 태스크 개수를 기반으로 마이그레이션을 발생시킨다.

2비트 폴링 예측기는 각 태스크마다 정의되어 있는 변수이며, 태스크가 새로 생성(fork) 될 시 폴링 예측기 값을 1로 초기화한다. 입출력 요청의 완료를 폴링 방식으로 대기할 때, 커널 내부에서는 blk_mq_poll() 함수가 호출된다 [4]. 따라서, 폴링 예측기는 태스크가 해당 함수를 호출한 경우 1씩 증가하고, 특정 시간동안 blk_mq_poll() 함수를 호출하지 않으면 1씩 감소한다. 각 런큐의 P 태스크의 개수가 변화하는 경우는 P 태스크가 종료 또는 sleep/wakeup을 하는 경우, 로드밸런서에 의한 마이그레이션이 발생하는 경우, 2비트 폴링 예측기의 값이 변화하는 경우이다.

기존의 리눅스 로드밸런서는 타이머(timer) 인터럽트의 softirq에서 동작하며, 호출하는 코어로부터 물리적으로 가까운 코어부터 탐색하여 로드 차이가 정해진 기준 값을 넘는 코어를 찾는다. 탐색을 통해 찾아낸 코어로부터 로드의 차이가 0에 가까워 질 때까지 태스크들을 마이그레이션을 해오는 work-steal 방식으로 두 코어의 로드를 맞춰준다 [5]. 따라서, 본 논문에서는 각 코어별로 타이머 인터럽트가 동작할 때, 마이그레이션을 해올 태스크들을 선정하는 과정에서 태스크의 특성을 고려한 정책을 세웠다.

P 태스크의 특성을 고려한 리눅스 로드밸런서의 설계는 아래와 같다. 태스크 마이그레이션이 발생할 때, P_s는 태스크의 출처 코어(s)의 P 태스크 개수이며, P_d는 태스크의 도착 코어(d)의 P 태스크 개수이고, 두 값은 한 번의 태스크 마이그레이션 마다 다시 계산된다.

그림 4 C 태스크 성능 결과 그래프

1) 태스크 마이그레이션

s와 d가 같은 물리적 코어인 경우, $P_d \geq P_s$일 때, s로부터 P 태스크를 마이그레이션하여 Colocate 배치를 맞춰주려 한다. 또한, s와 d가 서로 다른 물리적 코어인 경우 각각이 속한 물리적 코어 전체의 상황을 판단한다. 시스템 전체 P 태스크 개수(P_T), 전체 물리적 코어 개수(N_P), s와 d가 속한 물리적 코어의 P 태스크 개수(P_S, P_D)를 이용하여 $P_D < \frac{P_T}{N_P}$이고, $P_S > \frac{P_T}{N_P}$인 경우 s로부터 P 태스크를 마이그레이션하여 Scatter 배치를 맞춰준다. 이 외의 경우에는 P 태스크를 제외한 C 태스크를 마이그레이션한다.

2) 출처 코어 s 결정

s와 d가 같은 물리적 코어에 속한 서로 다른 논리 코어인 경우, s의 런큐에 P 태스크만 존재하고 $P_d < P_s$인 경우와, d의 런큐에 P 태스크만 존재하고 $P_s = 0$인 경우와 같이 이미 Colocate 배치가 이루어져 있는 경우에는 s를 다른 물리적 코어들 중에서 다시 선정한다.

4. 실험 결과

그림 4는 3장에서 제시한 로드밸런서의 C 태스크 성능을 측정한 결과이다. 총 4개의 물리코어 (총 8개의 논리코어)를 사용하며, 각 배치별로 C 태스크 8개, P 태스크 8개를 동시 배치하여 2장에서 실험적으로 찾은 Best와 Worst 배치, 그리고 기존의 리눅스 로드밸런서에 의하여 태스크들의 배치가 이루어지도록 하는 Free 배치와 성능을 비교하였다. 각 배치별로 Best의 실행 시간으로 표준화한 결과이며, Proposed는 본 논문에서 제시하는 로드밸런서의 성능이다. perlbench의 경우 Free 배치 대비 7.92%, Worst 대비 18.74%의 성능 향상이 있었으며, x264의 경우 Free 배치 대비 11.68%, Worst 대비 24.76%의 성능 증가를 얻을 수 있었다.

그림 5와 그림 6은 P 태스크의 성능 측정 결과이며, 각 배치별로 응답시간과 대역폭 성능의 편차가 각각 최대 2.3%, 2.34%임을 확인하였다.

제안하는 기법은 전체 P 태스크를 모든 물리적 코어에 균등하게 분배하고 (Scatter), 각 물리적 코어 내부적으로는 한 논리적 코어에서 실행되도록 하였으며 (Colocate), 이에 따라 C 태스크와 P 태스크 모두 Best 배치와 거의 동일한 성능을 얻을 수 있었다.

5. 결론 및 향후 연구

본 논문에서는 저장장치 입출력시 폴링을 수행하는 태스크와 연산중심 태스크가 동시 배치되는 상황을 고려하여 실험을 통한 최적의 배치를 찾아내고, 이를 통해 두 종류의 태스크들이 시스템에 주어진 CPU 자원 및 연산 유닛을 효율적으로 사용할 수 있는 새로운 태스크 배치 기법을 제시했다. 폴링 태스크와 연산 중심 태스크를 각 물리적 코어에 균등하게 분배하며, 물리적 코어 내부적으로는 같은 논리적 코어에 모여 실행될 수 있는 방식의 새로운 로드밸런서를 통하여 Worst 배치 대비 평균 21.75%, Free 배치 대비 평균 9.8%의 성능 향상을 얻을 수 있었다.

연산중심 태스크의 성능에 영향을 주는 요인에는 CPU 자원 및 연산 자원 활용률을 제외하고도 CPU 캐시(cache) 자원의 활용률도 영향을 준다. 폴링 태스크의 연산 자원 활용률과 함께 CPU 캐시의 활용률을 고려한 로드밸런서 연구를 향후 진행할 계획이다.

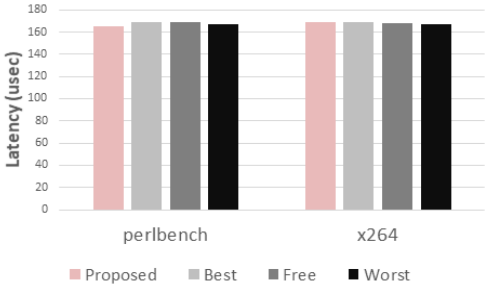

그림 5 P 태스크 응답 시간 측정 결과

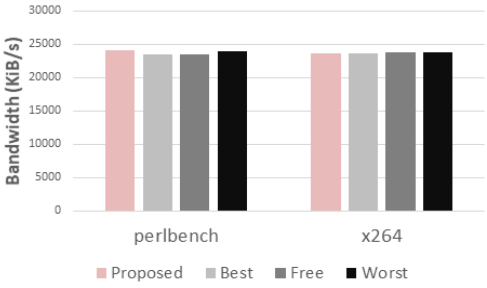

그림 6 P 태스크 대역폭 측정 결과

6. 참고문헌

[1] Yang, Jisoo, Dave B. Minturn, and Frank Hady. "When poll is better than interrupt." FAST. Vol. 12. 2012.

[2] Marr, Deborah T., et al. "Hyper-Threading Technology Architecture and Microarchitecture." Intel Technology Journal 6.1 2002.

[3] Limaye, Ankur, and Tosiron Adegbija. "A workload characterization of the spec cpu2017 benchmark suite." 2018 IEEE International Symposium on Performance Analysis of Systems and Software ISPASS. IEEE, 2018.

[4] 천명준, 한상욱, and 김지홍. "초저지연 저장장치를 위한 적응형 폴링 선택 기법." 한국정보과학회 학술발표논문집 2018

[5] Lozi, Jean-Pierre, et al. "The Linux scheduler: a decade of wasted cores." Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016.