공학석사학위논문

처리 시간에 민감한 워크로드의 성능 보장과 서버 자원 사용률 향상을 위한 효과적인 메모리 대역폭 할당 기법

An Efficient Memory Bandwidth Management for Ensuring Performance of Latency-sensitive Workload and High Server Utilization

2018 년 2 월

서울대학교 대학원

컴퓨터 공학부

민 지 수

처리 시간에 민감한 워크로드의 성능 보장과 서버 자원 사용률 향상을 위한 효과적인 메모리 대역폭 할당 기법

An Efficient Memory Bandwidth Management for Ensuring Performance of Latency-sensitive Workload and High Server Utilization

지도교수 엄 현 상
이 논문을 공학석사 학위논문으로 제출함
2017 년 11 월

서울대학교 대학원
컴퓨터 공학부
민 지 수

민지수의 공학석사 학위논문을 인준함
2018 년 1 월

위 원 장 염 헌 영 (인)

부위원장 엄 현 상 (인)

위 원 전 병 곤 (인)

초 록

웹 서비스나 소셜 네트워크 등과 같은 처리 시간에 민감한 워크로드들은 원하는 수준 내에 서비스를 제공하기 위해 성능에 민감하다. 이러한 특징으로 많은 경우 처리 시간에 민감한 워크로드들은 서버를 단독으로 사용하고 있고 그 결과 낮은 하드웨어 사용률의 경향을 보인다. 이를 해결하기 위해 처리 시간에 민감한 워크로드와 일괄 처리 워크로드들을 한 서버에 함께 배치하여 수행하게 함으로 하드웨어의 사용률을 높일 수 있다. 하지만 이 경우 여러 워크로드들이 동시에 공유하는 서버 자원의 경합 상황으로 인해 처리 시간에 민감한 워크로드는 원하는 성능을 보장받지 못하게 될 수 있다. 따라서 처리 시간에 민감한 워크로드의 성능을 정확히 보장하면서 서버 자원의 사용률을 높이는 방법이 필요하다. 이를 위해 본 논문에서는 여러 워크로드들이 동시 수행되는 상황에서 처리 시간에 민감한 워크로드에게 자원을 격리 제공하여 성능을 보장받을 수 있게 한다. 그리고 격리하여 제공할 자원의 크기를 예측하는 성능 예측 모델을 제안한다. 이 모델은 머신 러닝 알고리즘으로 생성되어 기존 연구에 비해 좀 더 정확하고 효율적인 자원 크기를 예측하고 할당한다는 장점이 있다. 본 논문에서 제안한 모델을 통해 처리 시간에 민감한 워크로드를 위한 메모리 대역폭의 크기를 예측하여 할당하고 처리 시간을 측정했을 때 완벽하게 원하는 성능을 보장하였으며 서버의 하드웨어 사용률도 최대 7.2배 증가하였다.

주요어 : 성능 예측, 메모리 대역폭, 공유 자원 관리, 머신 러닝 모델, 처리 시간에 민감한 워크로드
학 번 : 2016-21199

목 차

표 목차

그림 목차

제 1 장 서 론

웹 검색엔진, 소셜 네트워크, 금융 시장 어플리케이션 등과 같은 워크로드들은 서비스 제공자와 사용자 간 약속된 서비스 품질을 제공하기 위해 성능에 민감하다. 처리 시간에 민감한 워크로드들은 높은 서비스 품질을 제공하기 위해 서버 자원 대부분을 단독으로 사용하고 있다. 그 결과 데이터센터 내의 하드웨어 활용률은 10%~45% 정도로 현저히 낮은 경향을 보이고 있다[2]. 최근 사용률이 낮은 서버의 자원들을 활용하여 일괄 처리 워크로드들을 동시에 배치하고 하드웨어 사용률을 향상시키는 연구들이 많이 있다[1, 3, 4, 5, 11, 12]. 하지만 동시 배치함에 따라 처리 시간에 민감한 워크로드들은 일괄 처리 워크로드와 하드웨어 자원을 공유하게 되어 예상치 못한 느린 처리 시간을 경험하게 될 수 있다. 따라서 서버 하드웨어 자원의 사용률을 높이고 처리 시간에 민감한 워크로드의 성능을 보장하는 방법이 필요하다. 이를 위해 이전 많은 연구들에서는 처리 시간에 민감한 워크로드와 일괄 처리 워크로드가 하드웨어 자원을 공유함으로 생길 수 있는 경합 상황을 제거하거나 최소화하는 연구들을 진행했다. 대부분의 연구들은 공유하는 하드웨어 자원 중 코어나 캐시 메모리를 물리적으로 격리하는 방법으로 경합 상황을 해결할 수 있었다[3, 4, 11].

하지만 이전 연구[1]에서는 코어와 캐시 메모리의 격리에도 불구하고 처리 시간에 민감한 워크로드가 성능이 향상되지 않음을 밝히고 그 원인이 또 다른 공유 자원인 메모리 대역폭에 있음을 실험을 통해 확인하였다. 이 연구에서는 메모리 대역폭 관리를 통해 워크로드 간 경합 상황을 최소화시키는 방법을 제안한다. 제안한 기법은 사전 프로파일링이 진행되며 처리 시간에 민감한 워크로드들이 어떠한 상황에서도 완전히 성능을 보장받을 수 있도록 최악의 상황을 가정하여 진행된다. 그리고 사전 프로파일링으로 수집한 데이터를 바탕으로 예측선을 그리고 이를 기반으로 자원의 크기를 예측하고 할당해주었을 때 성능을 완전히 보장할 수 있었고 하드웨어 사용률 또한 향상되었다. 하지만 이 연구에서 제안한 기법인 최악의 상황에서의 프로파일링은 할당할

자원의 크기를 실제 수행 시에 필요한 크기보다 크게 예측하는 한계점이 있다. 이러한 경향을 보이는 원인으로는 메모리 대역폭의 경우 현재까지는 물리적으로 격리하는 방법이 없어 소프트웨어적으로 구현된 메모리 대역폭 격리 방식을 사용한 것에 있다. 함께 수행되는 일괄 처리 워크로드들이 메모리 대역폭에 주는 스트레스 정도가 최악의 상황의 보다 낮은 경우 예측한 처리 시간보다 낮게 나오는 경우가 생긴다는 것이다.

따라서 본 논문에서는 낭비되는 자원의 크기를 줄이고자 정확하고 효과적으로 성능을 보장하는데 필요한 자원의 크기를 예측하는 새로운 예측 모델을 제안한다. 제안하는 모델은 머신 러닝을 통해 생성된다. 머신 러닝은 성능에 영향을 미치는 여러 요인들의 복잡한 관계를 파악하고 반대로 중요하지 않은 요인들을 걸러내는 장점이 있다.

먼저 예측 모델을 생성하기 위해 처리 시간에 영향을 주는 요인들을 정의한다. 그리고 임의로 요인들을 조합하여 측정한 데이터 값들을 수집한다. 수집한 데이터를 기반으로 머신 러닝 알고리즘에 적용하여 모델을 생성한다. 이 과정들은 실제 워크로드들이 수행되기 전에 이루어진다. 예측 모델이 생성된 후 실제 워크로드들이 수행되는 여러 상황을 반영하여 처리 시간에 민감한 워크로드에 할당할 자원의 크기를 예측하여 할당한다.

본 논문의 공헌은 다음과 같다.

처리 시간에 민감한 워크로드와 일괄 처리 워크로드가 동시에 같은 서버에 배치되어 수행될 때 처리 시간에 민감한 워크로드의 성능을 보장하는데 필요한 자원의 크기를 예측하는 예측 모델을 제안한다.

예측된 자원의 크기를 처리 시간에 민감한 워크로드에게 제공하여 성능을 보장했을 뿐 아니라 일괄 처리 워크로드와 함께 수행하게 하여 서버의 하드웨어 사용률이 7.2배 높아지게 되었다.

본 논문의 구성은 다음과 같다. 제 2 장에서는 기존 연구의 한계점을 소개하며 본 논문의 배경을 소개한다. 제 3 장에서는 본 논문에서 제안하는 성능 예측 모델을 소개한다. 제 4 장에서는 예측 모델의 사용법에 대한 소개와 제 5장에서는 본 논문에서 제안한 모델을 이용하여 얻은 성능 결과와 서버 사용률에 결과 값을

보인다. 제 6장에서는 관련 연구를 소개하고 제 7 장에서 본 논문의 결론을 맺는다.

제 2 장 배 경

2.1 기존 연구와 한계점

높은 서비스 품질을 제공하기 위한 성능 보장과 하드웨어의 사용률 향상을 위해 이전 연구[1]에서는 메모리 대역폭이 최악인 상황일 때를 가정하여 사전 프로파일링을 진행한다. 그리고 처리 시간에 민감한 워크로드가 일괄 처리 워크로드와 동시 배치되어 수행 될 때의 성능을 보장받기 위해 필요한 메모리 대역폭 크기를 프로파일링을 기반으로 하여 예측하고 제공한다.

해당 이전 연구 결과인 OMBM(Optimized Memory Bandwidth Management)은 메모리 대역폭 크기를 예측하는 부분과 예측한 크기를 할당하는 부분으로 구성되어있다. 먼저 OMBM의 예측기에서는 처리 시간에 민감한 워크로드가 어떠한 일괄 처리 워크로드 조합과 수행되어도 성능을 보장 받을 수 있도록 메모리 대역폭 상황이 최악일 때를 가정하여 예측하는 예측선을 생성한다. 예측기에서 가정하는 최악의 상황을 만들기 위해 처리 시간에 민감한 워크로드가 돌아가는 코어를 제외하고 나머지 코어에 STREAM 벤치마크[15]를 수행하게 한다. STREAM 벤치마크는 메모리 대역폭에 일정하게 스트레스를 만들어 주는 장점이 있다. 최악의 상황을 구성하고 처리 시간에 민감한 워크로드에게 일괄 처리 워크로드들과 격리된 자원을 각각 할당한다. 격리된 자원의 크기를 최소 크기에서부터 최대 크기까지 늘려가며 자원의 크기에 따른 처리 시간에 민감한 워크로드의 처리 시간을 측정한다. 측정한 값을 가지고 예측선을 그려낸다.

예측선을 그려내는 순서는 다음과 같다. 먼저 그림 2.1과 같이 메모리 대역폭 크기 별 QPS(Queries per Second)에 따른 처리 시간을 측정하여 예측선을 그려낸다. 이 예측선을 기준으로 수행될 처리 시간에 민감한 워크로드의 QPS가 정해지면 두 번째 예측선을 그려낸다. 그림 2.2는 그림 2.1에서 어떤 워크로드의 QPS가 7,000일 때를 그려낸 두 번째 예측선이다. 이 예측선으로 어떤 워크로드가 성능을 보장받기 위해 할당 받아야 하는 자원의 크기를

구할 수 있다. 사용자가 보장 받아야 하는 성능의 기준 값을 SLO(Service Level Object)라고 표현할 때 처리 시간에 민감한 워크로드는 정해진 SLO 이내로 성능이 보장되어야 한다. 그래서 SLO를 보장받기 위해 할당 받아야 하는 메모리 대역폭의 크기를 두 번째 예측선을 통해 구할 수 있다.

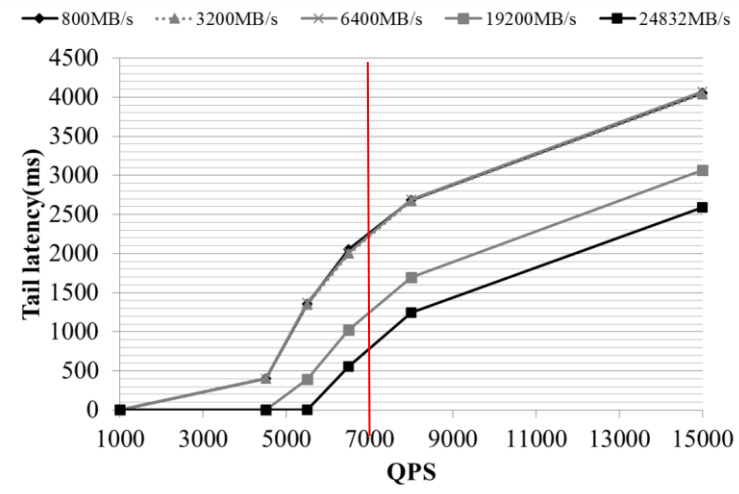


그림 2.1 OMBM으로 생성한 첫 번째 예측선

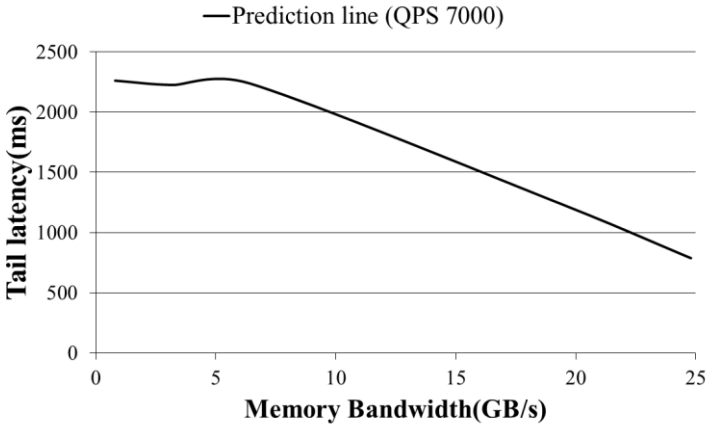


그림 2.2 OMBM으로 생성한 두 번째 예측선

본 논문에서는 최악의 상황을 가정하고 예측한 자원의 크기가 실제 수행 시에는 낭비되는 경향을 보인다는 한계점을 발견하였다. 그림 2.3은 메모리 대역폭 크기 별로 처리 시간에 민감한

워크로드가 실제로 임의의 일괄 처리 워크로드들의 조합과 수행되었을 때 처리 시간을 나타낸 그래프이다. 실제로는 그림 2.3에 나타난 예측선과 거의 일치하는 값으로 처리 시간이 측정되어야 한다. 하지만 그림 2.3을 보면 0.8GB/s에서 20GB/s의 메모리 대역폭 크기의 구간까지 예측선의 값과 차이가 나는 것을 볼 수 있다.

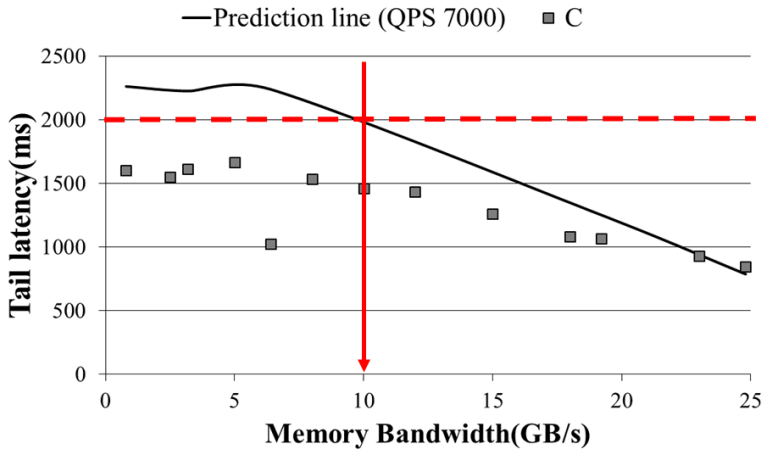


그림 2.3 OMBM으로 생성한 예측선과 실제 처리 시간 값

예를 들어 SLO가 2,000ms라고 한다면, SLO를 보장해주기 위해 워크로드에게 할당할 메모리 대역폭의 크기는 그림 2.3을 기준으로 약 10GB/s로 계산된다. 이렇게 계산된 메모리 대역폭의 크기를 처리 시간에 민감한 워크로드에게, 나머지를 일괄 처리 워크로드에게 할당하고 처리 시간에 민감한 워크로드의 처리 시간을 측정했더니 실제 처리 시간은 2,000ms 보다 훨씬 낮게 측정되었다. 이것을 다시 말하면 최소한의 메모리 대역폭만을 할당해주어도 SLO를 충분히 맞추어 줄 수 있다는 것이다.

위의 결과와 같이 예측한 값이 실제 측정 값과 차이가 나는 이유는 메모리 대역폭이 최악의 상황일 때를 예측하여 제공했기 때문이다. 메모리 대역폭 격리의 경우 현재까지는 물리적으로 격리하는 방법이 없기 때문에 OMBM에서는 Memguard[6]라는 소프트웨어 기반 메모리 대역폭 격리 시스템을 이용하였다. 그 결과 함께 수행되는 일괄 처리 워크로드 조합에 따라 서버에 영향을 주는 정도에 따라 예측 처리 시간이 차이가 날 수 있다는 것이다.

만약 서버에 주는 스트레스 정도가 최악의 상황 이하라면 그 이하의 처리 시간이 나올 수 있다.

따라서 본 논문에서는 이전 연구의 한계를 해결하기 위하여 메모리 대역폭 경합의 영향을 최악의 상황으로만 국한시키지 않고 다양한 메모리 대역폭 경합의 정도에도 정확하게 메모리 대역폭 크기를 예측하는 기법을 제안한다.

제 3 장 성능 예측 모델

본 논문에서는 이전 연구에서의 한계점을 해결하고 효율적으로 자원의 크기를 예측하는 예측 모델을 제안한다. 머신 러닝을 통해 생성하는 모델은 먼저 데이터 수집 단계를 거치고 데이터를 머신 러닝 알고리즘에 적용한다. 자원의 크기를 예측하는 모델은 서버의 종합적인 상황을 반영하고 그에 따라 정확한 최소한의 자원 크기를 예측해내야 한다. 이를 위해 머신 러닝 알고리즘을 통한 모델 생성은 의미가 있다. OMBM에서 제안한 모델은 모든 상황을 반영하기 위해서 각 상황에 맞는 예측선을 그려내야 한다. 이렇게 된다면 예측선을 그려내기 위해 수집해야 하는 데이터의 양이나 데이터를 기반으로 그려내는 그래프의 크기와 개수가 급격히 증가하게 될 것이다. 그래서 오프라인으로 수집하는 적절한 데이터의 양으로도 정확도가 높은 모델을 생성해낼 수 있는 방법이 필요하며, 본 논문에서는 이를 위해 머신 러닝 알고리즘으로 생성하는 모델을 제안한다. 이 모델은 서버의 다양한 상황을 잘 반영하여 정확한 자원의 크기를 예측할 수 있도록 한다. 예측 모델은 메모리 대역폭에 가해지고 있는 스트레스 정도, 처리 시간에 민감한 워크로드의 QPS, 보장 받아야 할 처리 시간의 기준 값을 입력 값으로 받는다. 모델은 입력 값을 받아 성능을 보장 할 수 있는 메모리 대역폭 크기를 결과 값으로 보여준다.

3.1 데이터 수집 단계

모델을 생성할 때 필요한 데이터는 오프라인으로 수집된다. 데이터는 아래 표 3.1에 나타난 특징에 의해 수집된다. 아래 표의 특징은 앞으로의 내용과 알고리즘에서도 같은 의미로 계속해서 쓰이게 된다.

특징 이름 (영어 이름)	설명
서버의 메모리 대역폭 스트레스 정도 (*Stress Degree*)	처리 시간에 민감한 워크로드 이외의 워크로드들이 메모리 대역폭에 가하고 있는 크기

처리 시간에 민감한 워크로드의 1초당 쿼리 개수 (*QPS*)	처리 시간에 민감한 워크로드의 1초당 쿼리 개수
처리 시간에 민감한 워크로드에게 제공된 메모리 대역폭의 크기 (*AllocatedBW*)	할당된 메모리 대역폭의 크기
처리 시간에 민감한 워크로드의 처리 시간 (*Latency*)	위의 세 가지 요인들을 적용해서 측정된 처리 시간

표 3.1 데이터를 수집할 때 쓰이는 특징의 종류와 그에 대한 설명

표의 첫 번째 줄부터 세 번째 줄의 특징은 처리 시간에 민감한 워크로드의 처리 시간에 영향을 주는 특징들이다. 이 세 가지 특징들을 다양하게 조합하여 실제 상황처럼 서버를 구성하여 처리 시간에 민감한 워크로드의 처리 시간을 측정한다. 이것이 마지막 네 번째 특징을 의미한다. 이 데이터 특징을 기준으로 수집된 데이터를 모델을 생성하는데 사용한다.

첫 번째 특징은 현재 서버의 메모리 대역폭에 가해지고 있는 스트레스 정도를 의미한다. 앞으로 논문에서 *Stress Degree*로 표현되는 특징이다. 이 특징의 데이터 값은 처리 시간에 민감한 워크로드 외에 함께 수행되는 워크로드가 메모리 대역폭에 가하는 정도를 Intel Performance Counter Monitor(PCM)[7]로 계산한 것이다. 모델 생성을 위해 데이터 수집 시에는 처리 시간에 민감한 워크로드가 수행되는 코어를 제외한 나머지 코어에서 STREAM 벤치마크를 수행시켜 메모리 대역폭에 가하는 스트레스 정도를 PCM으로 측정한다. 메모리 대역폭이 받는 스트레스 정도를 다양하게 구성하기 위해 STREAM 벤치마크가 사용하는 array의 크기를 다양하게 구성한다. 모든 스트레스 정도로 데이터를 구성하기에는 데이터의 양이 많아지기 때문에 서버의 메모리 대역폭에 가하는 스트레스 정도를 몇 개의 값으로 정한다. 본 논문에서는 25.6GB/s가 최대인 메모리 대역폭에서 6.8GB/s, 10.7GB/s, 14.3GB/s, 17.9GB/s 및 21.9GB/s로 나눈다. 두 번째 특징은 처리 시간에 민감한 워크로드의 1초당 쿼리 개수를 의미하며 QPS로 나타낸다. 처리 시간에 민감한 워크로드의 QPS에 따라 처리 시간이 차이가 나므로 QPS의 크기도 다양하게

구성하도록 한다. 그리고 세 번째 특징은 처리 시간에 민감한 워크로드에게 제공된 메모리 대역폭의 크기로, *AllocatedBW*라고 표현한다. *AllocatedBW*는 Memguard를 통해 처리 시간에 민감한 워크로드에게 할당되는 메모리 대역폭 크기를 의미한다. 두 번째, 세 번째 특징의 값은 최소, 최대 범위 값 내의 값으로 임의로 정한다. 마지막 네 번재 특징은 첫 번째부터 세 번째 특징의 데이터 값들을 조합하여 처리 시간에 민감한 워크로드가 얻은 실제 처리 시간으로, *Latency*라고 정의한다. 이 특징은 예측 모델을 사용하는 사용자가 보장받기 원하는 처리 시간 값인 *SLO*로 표현될 수 있다.

3.2 모델 훈련 단계

위의 단계에서 수집한 데이터를 머신 러닝 알고리즘에 적용하고 모델을 훈련시키는 과정을 거친다. 이렇게 생성된 모델은 주어진 입력 값을 분류하거나 계산해내어 결과 값을 보여준다. 모델 훈련 단계는 실제 수행이 이루어지기 전에 이루어진다. 처리 시간에 민감한 워크로드 별로 데이터를 수집하고 모델을 생성한다. 이 과정에서 본 논문에서는 머신 러닝 알고리즘에 데이터를 적용할 때 Weka[8]라는 오픈 소스 머신 러닝 툴을 이용하였다

모델을 생성할 때 사용할 수 있는 머신 러닝 알고리즘에는 다양한 알고리즘들 중 성능 예측 모델에 적용하기에 가장 적합한 것을 찾아야 한다. 이 때 고려해야 할 중요한 요소로는 첫째로 모델이 얼마나 정확히 예측 값을 분류 또는 계산해내는지, 둘째로 훈련시키는 과정에서 모델이 얼마나 빠르게 생성되는지, 마지막으로 예측 값을 얼마나 빠르게 계산해내는지가 있다. 이러한 요소들을 고려하였을 때 본 논문에서는 의사 결정 나무 알고리즘 중 REPTree 알고리즘을 사용하기로 한다.

REPTree 알고리즘은 정확한 예측 값을 제공하는 성능 예측 모델로 이미 연구에 적용되었던 적이 있고[9] 의사 결정 나무 알고리즘 중에서도 모델의 빠른 생성 시간으로 잘 알려져 있다. 이 알고리즘을 통해 훈련된 모델은 트리 형태로서 트리의 각 노드는 데이터의 각 특징으로 구성되어있다. 그리고 각 노드에는 다음 노드를 선택하는 기준이 주어져 있다. 생성된 모델에 입력 데이터가

주어지면 트리의 루트 노드에서부터 결과 값이 있는 리프 노드를 찾아간다. 리프 노드까지 찾아가며 거치는 각 노드에서는 기준 값과 입력 값을 비교하며 다음으로 가게 되는 노드를 선택하게 된다.

입력 데이터는 예측하고자 하는 특징을 제외한 나머지 특징들의 값을 입력 값으로 한다. 예를 들어 표 3.1에서 *AllocatedBW*를 예측해내려고 한다면 리프 노드는 *AllocatedBW* 특징을 가지게 되며 나머지 노드들은 나머지 특징들로 모델이 생성된다. 그 다음 입력 값을 모델에 대입하면 각 노드에서는 특징들의 값이 기준 값의 이상인지 이하인지를 판단하여 다음 노드로 이동한다. 이렇게 각 노드를 따라서 리프 노드까지 이르게 되면 주어진 입력 값으로 인해 예측되는 값을 출력해준다. 실제 예를 들어 보면 어떤 노드의 특징이 QPS이고 기준 값이 3,000이라고 할 때, 사용자로부터 받은 입력 값 중 QPS가 3,000이하라면 왼쪽 노드로 내려가고 3,000보다 크다면 오른쪽 노드로 내려가게 된다.

제 4 장 구 현

위에서 제안한 방법으로 처리 시간에 민감한 워크로드의 성능 보장을 위해 성능 예측 모델을 생성하고 모델을 적용하여 처리 시간에 민감한 워크로드에게 할당할 메모리 대역폭 크기를 예측하고 할당하는 단계를 거친다.

4.1 예측 단계

생성된 트리 형태의 모델은 Algorithm 1을 거쳐 처리 시간에 민감한 워크로드에게 할당할 메모리 대역폭 크기를 예측해낸다. 일반적으로 트리 모델을 검색하는 방식으로 찾아낸 결과 값은 정확도가 낮은 경향이 있다. 따라서 본 논문에서는 Weka에 구현된 일반적인 검색 알고리즘을 수정한 알고리즘을 제안한다. 결과 값을 찾아가는 검색 알고리즘이 바뀌었을 뿐 예측 모델에는 아무런 영향이 없다.

이 알고리즘에서 사용자가 정한 성능 보장 기준 값을 *SLO*라고 표현한다. 다시 말하면 *SLO* 이하로 처리 시간이 나와야 사용자가 원하는 성능을 보장한 것이라고 말할 수 있다. 결과 값을 얻기 위한 입력 값은 *Instance*로 표현된다. *Instance*는 사용자가 입력한 *SLO*를 포함하여 처리 시간에 민감한 워크로드의 *QPS*, 그리고 *Stress Degree* 특징으로 구성되어 있다. Output 중 *PredictedBW*는 처리 시간에 민감한 워크로드에 할당할 메모리 대역폭의 크기이다. 처리 시간에 민감한 워크로드가 성능을 보장받기 위해 적어도 할당 받아야 하는 메모리 대역폭의 크기를 의미한다. *PredictedBW* 값은 트리 구조에서 리프 노드의 값에 해당한다.

트리 형태의 모델에서 트리의 노드를 검색하며 각 노드가 가지고 있는 기준 값에 따라 알맞은 다음 가지를 선택한다(5 및 12줄). 예를 들어 현재 노드의 특징에 대응하는 입력 값이 기준 값보다 작다면 노드의 왼쪽(8줄)으로 반대의 경우 오른쪽을 선택하게 된다. 재귀적인 방법으로 리프 노드까지 찾아 들어가 결과

Algorithm 1 Decision tree Modified Search Algorithm

Input: *Instance* : A set of QPS value, SLO value, and Stress Degree
Output: *PredictedBW* : Predicted size of memory bandwidth to allocate
A : Minimum latency
B : Maximum latency under the SLO
Data: *Tree* : Generated decision tree
SplitPoint : Value of point where the tree is splited
ATTR : One of QPS, SLO and CurrentBW

```
1:  A ← 0
2:  B ← ∞
3:  def modifiedSearch(Instance):
4:    if Tree.Node ≠ LEAF
5:      if Instance.value(ATTR) < SplitPoint
6:        if ATTR = "SLO"
7:          B ← SplitPoint
8:        PredictedBW ← Tree.Node[Left].modifiedSearch(Instance)
9:        if ATTR = "SLO"
10:         if Change = true
11:           Change ← false
12:     else
13:       if ATTR = "SLO"
14:         oldA ← A
15:         A ← SplitPoint
16:       PredictedBW ← Tree.Node[Right].modifiedSearch(Instance)
17:       if ATTR = "SLO"
18:         if Change = true
19:           B ← A
20:           A ← oldA
21:           PredictedBW ← Tree.Node[Left].modifiedSearch(Instance)
22:           Change ← false
23:   if Node = LEAF
24:     if B > Instance.value("SLO")
25:       Change ← true
26:     else
27:       Change ← false
28:     return Tree.Node[output]
```

값을 구한다 (8줄 및 16줄).

수정된 알고리즘에서는 *A*와 *B*라는 변수를 추가한다. 트리를 검색하면서 노드의 특징이 SLO인 경우에만 *A*와 *B*라는 변수의 값을 업데이트한다. 예를 들어 어떤 노드가 가진 특징이 *SLO*이고 기준 값 (*SplitPoint*)보다 입력 받은 SLO 값이 작다면 B가 *SplitPoint*가 된다 (6 및 7줄). 반대로 기준 값보다 주어진 SLO 값이 크다면 *A*가 *SplitPoint*가 된다 (13−15줄). 이런 방식으로 트리 구조를 검색하며 *A*와 *B*의 값을 업데이트한다. 그리고 리프 노드에 도달했을 때 *A*와 *B*의 값이 최종적으로 결정되어 있게 된다.

A와 B 변수를 수정한 알고리즘에 추가한 이유는 다음과 같다. 아무런 제한 없이 *A*와 *B*의 값을 구하면 어떤 경우 A의 값은 *SLO*보다 작으나 *B*의 값이 *SLO*보다 큰 경우가 생기기 때문이다. 예를 들어 입력 값의 *SLO*가 200ms일 경우, *SLO* 특징을 가진 어떤 노드에 도달했을 때 그 때의 기준 값이 250이었다면 *B*가 250ms로 정해지게 되고 왼쪽 가지로 내려가게 된다 (6-8줄). 이렇게 리프 노드까지 도달하여 끝이 나면 결론적으로 *SLO*는 *SLO* ≤ 250의 범위를 가지게 된다. 이 범위는 200ms 이하의 값을 항상 보장해 줄 수 없을 가능성이 생기게 된다.

그래서 *A*와 *B*의 결과 값은 항상 *SLO*보다 작아야 한다는 조건이 추가된다. 이 조건을 만족시키기 위해 리프 노드에 도달했을 때 *B*의 값이 주어진 입력 값의 *SLO*보다 작은지 확인한다 (24줄). 만약 *B*의 값이 *SLO*보다 작지 않다면 Change라는 변수를 참으로 설정한다 (25줄). 그리고 리프 노드에서 자신의 부모 노드로 빠져나가면서 B의 값을 다시 구하려고 하고 원래 들어왔던 가지의 반대쪽 가지로 들어간다 (21줄).

4.2 할당 단계

모델과 위의 알고리즘을 통해 구한 메모리 대역폭의 크기를 처리 시간에 민감한 워크로드에게 적용하고 나머지는 일괄 처리 워크로드에게 할당해준다. 메모리 대역폭을 점유할 수 있게 하는 운영체제 상에서 구현된 시스템인 Memguard를 이용한다. Memguard는 각 코어 별로 메모리 대역폭 사용량을 측정하기 위해 L3 캐시 미스 성능 카운터(LLC Miss Performance Counter)를 측정한다. 만약 코어가 설정된 메모리 대역폭 크기보다 더 많이 사용한다면, 시스템은 운영체제 스케줄러에게 코어가 메모리 대역폭을 더 이상 사용하지 못하도록 요청한다. 그럼 스케줄러는 코어에 있던 워크로드를 잠시 제외하여 다음 시간 때까지 메모리 대역폭을 사용하지 못하도록 한다. 비록 현재는 메모리 대역폭을 물리적으로 격리시키지 못하고 있지만 그와 비슷한 효과를 보인다.

제 5 장 실험 결과 및 분석

5.1 실험 환경과 벤치마크 워크로드

본 논문의 실험들은 쿼드 코어 Intel i7-4770k 3.5GHz 프로세서가 내장된 Intel Haswell 서버에서 이루어졌다. 서버에는 8MB 16-way set associative 캐시 메모리와 25.6GB/s 메모리 대역폭이 설치되어있고 리눅스 3.14.15 버전의 커널과 Hadoop 2.6을 사용하였다. 이 논문에서는 처리 시간에 민감한 워크로드로는 MIT에서 개발한 Tailbench 벤치마크[10]를 사용했다.

5.2 예측 정확도

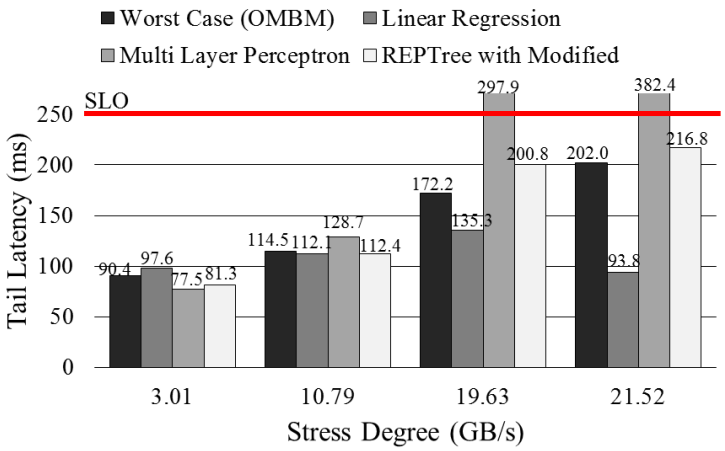
처리 시간에 민감한 워크로드의 성능을 보장해주기 위해 필요한 메모리 대역폭 크기를 예측하는 성능 예측 모델을 생성한다. 데이터를 수집한 후 여러 머신 러닝 알고리즘에 적용하여 모델을 생성하고 각 모델이 예측한 메모리 대역폭 크기를 할당한 후 처리 시간에 민감한 워크로드가 사용자가 원하는 성능을 보장해주는지를 확인한다.

그림 5.1-5.3은 본 논문에서 제안한 성능 예측 모델의 정확도를 보이기 위한 실험이다. REPTree 알고리즘으로 생성한 모델과 비교할 대상으로 최악의 상황을 가정하여 예측하는 OMBM과 머신 러닝 알고리즘 중 Linear Regression 알고리즘과 Multi Layer Perceptron 알고리즘으로 각각 생성한 모델을 사용한다. 그래프에서 각각 Worst Case(OMBM), Linear Regression 및 Multi Layer Perceptron으로 나타나 있다. 그리고 본 논문에서 제안한 모델은 REPTree with Modified로 나타낸다.

이 실험은 각 예측 모델들에 같은 입력 값을 적용하여 각 모델로부터 성능 보장을 위해 예측한 메모리 대역폭의 크기를 구한다. 그리고 구한 메모리 대역폭 크기를 처리 시간에 민감한 워크로드에게 할당하여 실제 걸리는 처리 시간을 측정하였다. 이 실험에서 처리 시간은 Tail Latency로 표현하며, 이것은 95번째

백분위의 처리 시간을 의미한다. 그림 5.1-5.3의 (a) 그래프에서 x축은 처리 시간에 민감한 워크로드를 제외한 워크로드들이 메모리 대역폭에 주는 영향을 나타낸다. 이 실험은 STREAM 벤치마크를 이용하여 메모리 대역폭에 스트레스를 주었다. 그리고 (b)는 각 모델이 Stress Degree에 따른 예측한 메모리 대역폭 크기를 나타낸다.

그림 5.1은 Silo라는 워크로드가 QPS를 8,000을 가질 때의 정확도 실험이다. (a)의 그래프를 보면 Multi Layer Perceptron으로 생성한 모델을 제외하고는 모두 SLO를 만족하고 있다. Multi Layer Perceptron은 SLO를 보장할 만큼의 메모리 대역폭 크기를 예측해주지 못했다. 나머지 모델들은 SLO를 모두 만족시켰지만 Linear Regression 알고리즘을 생성된 모델은 Worst Case나 REPTree with Modified 모델보다 더 많은 크기를 예측해 주었음으로 다른 두 모델보다 정확도가 떨어진다는 것을 알 수 있다. 나머지 두 모델은 SLO를 만족하면서도 최소한의 메모리 크기를 예측하여 제공해주었다.



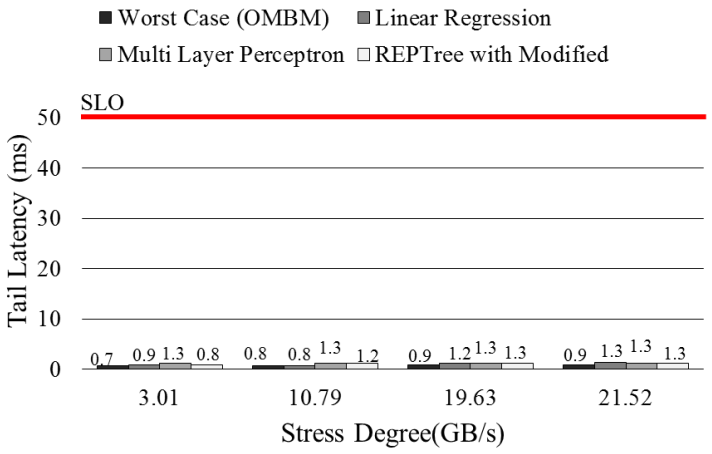
(a)

	Current Memory Bandwidth Situation (GB/s)			
	3.01	**10.79**	**19.63**	**21.52**
Worst Case(OMBM)	9.881	9.881	9.881	9.881
Linear Regression	12.204	12.575	12.997	13.087
Multi Layer Perceptron	7.112	7.392	7.528	7.545
REPTree with Modified	9.236	9.236	9.236	9.236

(b) 예측된 메모리 대역폭 크기 (GB/s)

그림 5.1 Prediction Accuracy - Silo (QPS 8,000)

그림 5.2는 Masstree 워크로드의 실험결과이다. Masstree는 QPS가 3,000인 상황이다. Masstree는 모든 예측 모델이 SLO보다 한참 낮은 처리 시간을 보이고 있다. 이는 QPS가 3,000일 때 로드가 상대적으로 적어 항상 처리 시간이 낮은 채로 측정된다. 그렇기 때문에 예측 모델 전부 성능을 충분히 보장하고 있음을 볼 수 있지만 이와 같은 경우 가장 적은 크기의 자원을 예측한 것이 정확도가 높다고 결론 지을 수 있다. Worst Case의 경우는 최악의 경우를 예측하기 때문에 가장 큰 메모리 대역폭 크기를 예측하였다. 가장 작은 메모리 대역폭 크기를 예측한 모델은 REPTree with Modified이다.



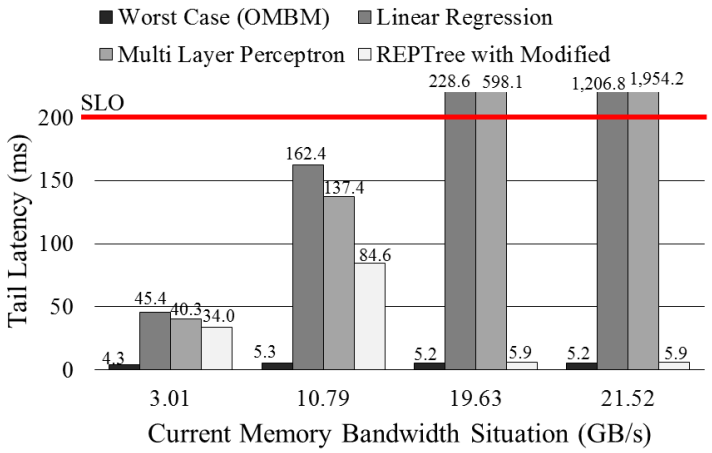
(a)

	Current Memory Bandwidth Situation (GB/s)			
	3.01	**10.79**	**19.63**	**21.52**
Worst Case(OMBM)	16.472	16.472	16.472	16.472
Linear Regression	12.356	12.356	12.356	12.356
Multi Layer Perceptron	5.909	6.385	6.862	6.958
REPTree with Modified	4.593	4.593	4.593	4.593

(b) 예측된 메모리 대역폭 크기 (GB/s)

그림 5.2 Prediction Accuracy – Masstree (QPS 3,000)

그림 5.3는 Img-dnn 워크로드의 실험결과이다. 현재 QPS는 550이다. (a)의 실험결과를 살펴보면 Linear Regression 모델과 Multi Layer Perceptron 모델은 19.63GB/s 상황과 21.52GB/s 상황에서 SLO를 만족시키지 못하고 있는 모습을 볼 수 있다. 이것으로 볼 때 두 모델은 처리 시간에 민감한 워크로드 이외의 워크로드들이 수행되는 상황을 충분히 반영하지 못하고 있음을 알 수 있다. 반면 Worst Case 모델의 경우에는 항상 최악의 경우를 가정하고 예측하기 때문에 가장 넉넉한 메모리 대역폭 크기를 할당하며 여유 있게 SLO를 만족시키고 있다. 그리고 REPTree with Modified 모델의 경우에도 SLO를 충분히 여유 있게 만족시키고 있다. 그러나 (b)를 보면 19.63GB/s 이상의 상황일 때는 그 상황에 맞게 메모리 대역폭 크기를 충분히 늘려 SLO를 보장할 수 있게 해주고 있다. REPTree with Modified 모델의 정확도가 높아진 것은 수정한 알고리즘에서 A와 B 변수를 적용하여 어떤 상황에든 SLO를 충분히 만족 시킬 수 있게 하기 때문이다.



(a)

	Current Memory Bandwidth Situation (GB/s)			
	3.01	**10.79**	**19.63**	**21.52**
Worst Case(OMBM)	23.284	23.284	23.284	23.284
Linear Regression	12.590	14.717	17.132	17.649
Multi Layer Perceptron	10.198	13.469	18.565	19.113
REPTree with Modified	17.875	17.875	22.391	22.391

(b) 예측된 메모리 대역폭 크기 (GB/s)

그림 5.3 Prediction Accuracy – Img-dnn (QPS 550)

5.3 성능 보장과 사용률 향상

이번 절에서는 실제 일괄 처리 워크로드와의 실험 결과를 보인다. 아래 표 5.1은 처리 시간에 민감한 워크로드와 수행할 일괄 처리 워크로드 조합을 정리한 것이다.

조합	워크로드	평균 Stress Degree
A	Kmeans, Bayesian, Pagerank	3.6
B	Kmeans, libquantum, bwaves	13.8
C	Kmeans, Pagerank, lbm	7.5

표 5.1 실험에 쓰인 일괄 처리 워크로드 조합

이 실험에서는 각 조합의 메모리 대역폭 사용량을 확인해본 결과 크게 변동이 심하지 않아 각 조합의 메모리 대역폭 사용량의

평균으로 예측하게 하였다.

그림 5.4의 (a)-(c)은 처리 시간에 민감한 워크로드와 일괄 처리 워크로드와의 조합과 실험을 했을 때 예측한 메모리 대역폭 크기와 실제 처리 시간을 나타낸 결과이다.

	예측한 크기	실제 처리 시간
A	20.029 GB/s	34.57 ms
B	20.029 GB/s	72.875 ms
C	20.029 GB/s	49.182 ms

(a) silo, QPS 8,000, SLO 200ms

	예측한 크기	실제 처리 시간
A	6.507 GB/s	0.627 ms
B	6.507 GB/s	0.739 ms
C	6.507 GB/s	0.833 ms

(b) masstree, QPS 3,000, SLO 200ms

	예측한 크기	실제 처리 시간
A	10.931 GB/s	38.582 ms
B	14.464 GB/s	26.947 ms
C	10.931 GB/s	4734.795 ms

(c) img-dnn, QPS 550, SLO 5,000

그림 5.4 일괄 처리 워크로드와 함께 수행 시 처리 시간에 민감한 워크로드의 실제 처리 시간

일괄 처리 워크로드와 함께 동시 수행하게 될 때 CPU 사용률에 대한 결과는 다음 그림 5.5의 (a)-(c)와 같다.

	처리 시간에 민감한 워크로드 혼자 (%)	일괄 처리 워크로드와 함께 수행 (%)
A	17.77	97.25
B	17.90	95.25
C	15.07	96.5

(a) silo

	처리 시간에 민감한 워크로드 혼자 (%)	일괄 처리 워크로드와 함께 수행 (%)
A	24.82	94.65
B	24.15	93.275
C	15.52	92.7

(b) masstree

	처리 시간에 민감한 워크로드 혼자 (%)	일괄 처리 워크로드와 함께 수행 (%)
A	17.77	64.08
B	13.45	97.6
C	18.92	97.1

(c) img-dnn

그림 5.5 CPU 사용량

그림 5.5는 서버의 하드웨어 사용률의 향상을 보여주는 결과이다. 이 값들은 서버에서 사용할 수 있는 CPU에 대한 사용량을 백분율로 구한 값이다. 위의 결과를 분석해보면 평균적으로 하드웨어 사용률을 5.2배 상승했고 최대 7.2배가 상승했음을 알 수 있다.

제 6 장 관련 연구

6.1 서비스 품질 보장과 서버의 효율적인 사용

빅데이터와 클라우드 서비스가 발전하면서 데이터 센터들의 비중이 점점 커져가고 있지만 서비스들의 서비스 품질을 보장하기 위해 효율적으로 서버가 사용되지 못하고 있다. 이러한 한계점을 해결하기 위해 공유자원을 관리하여 경합 상황을 제거하거나 어플리케이션 간의 관리로 경합 상황을 피하는 연구들이 진행되어 왔다.

Kasture et al.은 주기적으로 최적의 LLC 파티션 사이즈를 계산하여 처리 시간에 민감한 워크로드와 일괄 처리 워크로드에게 각각 나누어 준다[4]. 가장 최적의 성능을 보이기 위해 처리 시간에 민감한 워크로드의 캐시 메모리 사용을 모니터링하며 제공할 캐시 메모리의 크기를 예측한다. Zhu et al. 또한 캐시 격리 기법을 이용하고 또한 DVFS를 이용한 주기 조절을 통해 관리하는 기법을 제안하였다[11]. 하지만 이 연구들은 캐시 메모리에서의 경합 상황만을 고려하였기 때문에 캐시 메모리 이외의 메모리 대역폭에서의 경합 상황을 충분히 반영하지 못한다는 단점이 있다.

Lo et al.은 모든 코어, L3 캐시, 파워, 네트워크와 같은 대부분의 공유 자원을 관리하는 연구를 진행했다[5]. 또한 이 연구에서는 메모리 대역폭에서의 경합 상황도 반영하여 메모리 대역폭에서의 경합 상황을 피할 수 있게 했다. 처리 시간에 민감한 워크로드들의 로드와 실제 처리 시간을 모니터링 하면서 서비스 품질을 완전히 제공하지 못하는 일정 수준을 넘어가면 자원들을 처리 시간에 민감한 워크로드가 충분한 자원을 사용할 수 있게 한다. Mars et al.과 Yang et al.는 처리 시간에 민감한 워크로드를 일괄 처리 워크로드와 동시에 수행할 때의 성능의 방해 정도를 측정함으로 문제를 해결한다[12, 13]. 이 연구들에서 제안한 시스템에서는 메모리 서브 시스템에 주어지는 스트레스의 크기에 따라 처리 시간에 민감한 워크로드의 성능을 미리 확인한다. 그리고 일괄 처리 워크로드가 메모리 서브 시스템에 주는 스트레스의

크기를 측정한다. 이렇게 측정된 정보를 가지고 처리 시간에 민감한 워크로드와 동시에 수행할 일괄 처리 워크로드를 선택하게 된다. 위 두 연구들에서는 시스템을 주기적으로 모니터링 하고 수행 도중 일괄 처리 워크로드의 수행을 제한하는 등의 제한 점이 있다.

6.2 머신 러닝을 적용한 성능 예측 모델

Dwyer et al.은 처음으로 머신 러닝을 적용하여 복잡한 문제를 해결한 연구이다[9]. 이 연구에서는 HPC 워크로드를 대상으로 성능 저하를 예측하는 모델을 제안하였다. 이 모델은 멀티 코어 환경에서 동시 스케줄링 된 워크로드들의 Performance Counter 값을 입력 값으로 받는다. 동시 스케줄링 된 워크로드들이 자원을 공유함으로 생길 수 있는 얼마나 성능 저하가 되는지 결과 값으로 보여준다. 이를 이용하여 워크로드 간 동시 수행으로 큰 성능 저하가 예측 될 시 스케줄링을 피하는 방법의 스케줄링 제안하기도 했다. Dauwe et al.연구도 비슷하게 어플리케이션의 동시 수행으로 인해 생기는 성능 저하를 예측하는 모델을 제안하였다[14]. 이 연구는 Dwyer et al.과는 다르게 신경망 모델을 이용하여 모델을 생성하였다.

제 7 장 결 론

서버의 하드웨어 사용률을 높이기 위해 처리 시간에 민감한 워크로드와 일괄 처리 워크로드를 함께 수행하려고 할 때 공유 자원에서의 경합으로 처리 시간에 민감한 워크로드는 성능을 보장받지 못하는 경우가 있다. 따라서 본 논문에서는 이러한 문제를 해결하기 위한 방법으로 머신 러닝으로 생성한 성능 예측 모델을 제안한다. 이 성능 예측 모델을 생성하기 위한 데이터들을 수집하기 위해 모델에 쓰일 특징의 종류를 정의하고 수집한 데이터들을 머신 러닝 알고리즘을 통해 성능 예측 모델을 생성한다. 이 모델을 이용하여 서비스 프로바이더는 최소한의 자원의 크기를 예측하고 할당하여 처리 시간에 민감한 워크로드의 성능을 완전히 보장하고 서버의 사용률을 높일 수 있다.

참고문헌

[1] H. Sung, J. Min, S. Ha, and H. Eom, "OMBM: optimized memory bandwidth management for ensuring qos and high server utilization," in 2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017, 2017, pp. 269-276.

[2] L. A. Barroso and U. Hoelzle, The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines, 1st ed. Morgan and Claypool Publishers, 2009.

[3] X. Yang, S. M. Blackburn, and K. S. McKinley, "Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading," in 2016 USENIX Annual Technical Conference (USENIX ATC 16). Denver, CO: USENIX Association, 2016, pp. 309-322.

[4] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict qos for latency-critical workloads," in Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 729-742.

[5] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in Proceedings of the 42Nd Annual International Symposium on Computer Architecture, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 450-462.

[6] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for

efficient performance isolation in multi-core platforms," in 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2013, pp. 55-64.

[7] "Intel Performance Counter Monitor," https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[8] "Weka," https://www.cs.waikato.ac.nz/ml/weka/.

[9] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 83:1-83:11.

[10] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in 2016 IEEE International Symposium on Workload Characterization (IISWC), Sept 2016, pp. 1-10.

[11] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 33-47.

[12] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in Proceedings of the 44th Annual IEEE/ACM International Symposium on

Microarchitecture, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 248-259.

[13] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in Proceedings of the 40th Annual International Symposium on Computer Architecture, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 607-618.

[14] D. Dauwe, E. Jonardi, R. Friese, S. Pasricha, A. A. Maciejewski,
D. A. Bader, and H. J. Siegel, "A methodology for co-location aware application performance modeling in multicore computing," in 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, May 2015, pp. 434-443.

[15] "STREAM Benchmark," http://www.cs.virginia.edu/stream/ref.html

Abstract

An Efficient Memory Bandwidth Management for Ensuring Performance of Latency-sensitive workload and high server utilization

Latency-critical workload such as web search service or Social networks are sensitive to tail latencies for meeting Quality of Serivce(QoS). Most of the server executes only the latency-critical workload for ensuring the highest QoS. In this case, this leads to low server hardware utilization. For improving hardware resource utilization, the service provider has to co-locate the latency-critical workloads and batch processing ones. However, the QoS of each latency-critical workload cannot be ensured because of interference on shared server resource. To solve this problem, we propose a performance prediction model that isolates the resources and provides them to the workloads. This model predicts the more exact size of the resource than the prior work. Before generating the model, we define the attributes of the data that will be used for training the data. And then we have chosen one of machine learning algorithms that is the best as the performance prediction model. After collecting the data, we train the model with the algorithm that we chose. This model predicts and allocates the exact size of memory bandwidth to each latency-critical workload. We have experimentally found that a server, to which our scheme has been applied, can achieve the QoS perfectly and improved the server resource utilization by up to 7.2x.

Keywords : Performance Prediction, Memory Bandwidth, Management for Shared Resource, Machine Learning Model, Latency-critical Workload

Student Number : 2016-21199