CMP 환경에서 공유 자원 경합을 고려한 스케줄링 기법

송욱 ○ 김지홍

서울대학교 컴퓨터 공학부

{answer03, jihong}@davinci.snu.ac.kr

Contention-Aware Scheduling Techniques for CMP with Shared Resources

Wook Song○ Jihong Kim

School of Computers Science and Engineering, Seoul National University

요 약

여러 개의 마이크로 프로세서가 하나의 칩에 집약되는 단일 칩 멀티프로세서 환경은 필연적으로 칩 내부의 각 프로세서 간의 공유 자원 경합 문제가 발생한다. 점점 더 많은 수의 프로세서를 집약하는 최근의 경향은 이러한 공유 자원 경합으로 인해 발생하는 성능 하락을 더욱 가중하고 있는 실정이다. 한편 근래의 마이크로 프로세서에서는 하드웨어 성능 카운터를 제공하여, 시스템에서 수행되는 응용의 여러 가지 성능 정보를 측정할 수 있도록 하고 있다. 본 논문에서는 프로세서에서 제공하는 하드웨어 성능 카운터를 이용하여 낮은 부하와 높은 정확도를 보이는 커널 수준의 동적 태스크 프로파일러를 고안하였다. 또한 동적 태스크 프로파일러에서 수집 가능한 하드웨어 성능 정보 이벤트들을 조합하여 함께 수행되는 태스크들간 상호 영향 정도 모델을 제안하였고, 본 모델을 태스크의 프로파일로서 스케줄러에 제공하여 공유 자원 경합을 최소화하는 스케줄링 알고리즘을 제안하였다. 공유 자원 경합을 고려한 스케줄링 기법은 태스크들의 공유 자원 경합 정도에 따라 최대 10.8%에서 최소 2.5%의 수행 시간 증가율 감소를 보였다.

1. 서 론

칩 멀티프로세서(Chip Multi-Processor, 이하 CMP) 시스템은 하나의 고성능 프로세서 설계가 물리적, 기술적 한계에 부딪힘에 따라 여러 개의 마이크로 프로세서를 하나의 칩에 집약하여 병렬화에 의한 전체 시스템 성능 향상을 최대한 얻을 수 있도록 설계한 시스템이다. 이러한 CMP 환경에서의 각 코어는 최하위 수준의 캐시, 시스템의 공유 버스와 같은 칩 내부의 자원을 공유한다. 이렇게 공유되는 자원들은 공유 캐시 메모리처럼 사용할 수 있는 전체 공간이 한정되어 있거나, 또는 공유 버스와 같이 상호 배제적인 성질을 갖고 있기 때문에, 각 코어에서 수행 중인 응용의 공유 자원 사용 패턴에 따라 자원의 경합 문제가 발생할 수 있다. 공유 자원의 경합은 한 코어가 점유하고 있는 공유 자원을 해지하거나, 다른 코어의 공유 자원 요청을 지연시키는 등의 현상을 발생시키므로 성능에 큰 악영향을 끼친다. 이러한 추세는 최근의 응용들이 점차 많은 메모리 사용하는 경향과 맞물려 더욱 심각한 문제로 부각되고 있다.

CMP의 고질적인 공유 자원 경합 문제는 기존의 대칭 멀티 프로세서(Symmetric Multiprocessor, 이하 SMP) 환경보다 스케줄러의 책임을 더욱 무겁게 하였다. 수행될 태스크를 선정할 시점에서 스케줄러가 능동적으로 개입하여 시스템 전체의 적절한 공유 자원 경합 정도를 유지하는 것은 성능 향상에 직접적으로 기여할 수 있으므로, CMP 환경에서의 스케줄러 설계에 필히 고려해야 할 요소이다. 하지만 대부분의 운영체제들이 현재 SMP 커널을 멀티 프로세서를 지원하는 커널로 제공하고 있다. 이러한 SMP 커널은 칩 내부의 각 코어를 완전히 독립된 캐시와 메모리를 확보한 하나의 프로세서의 관점에서 태스크를 할당하고 자원을 관리하므로 CMP에 특화된 공유 자원의 경합을 고려한 스케줄러 연구가 시급하다.

한편 최근의 프로세서는 하드웨어 성능 카운터(Hardware Performance Counter, 이하 HPC)를 포함하여 프로세서에서 수행된 응용의 수행된 사이클, 수행한 명령어의 수, 캐시 미스의 수와 같은 여러 성능 정보를 얻을 수 있는 메커니즘을 제공한다. 이러한 정보는 응용의 성능을 다각도로 분석하여 최적화 하는 과정에 사용될 수 있으며, 실제로 상용화 또는 공개된 많은 최적화 도구들이 프로세서에서 제공하는 HPC를 활용하고 있다. HPC의 제어는 프로세서의 제조사나 모델에 따라 변동이 있는데, PAPI[1]는 사용자가 그러한 변동을 신경쓰지 않고 HPC에서 제공하는 성능 정보를 쉽게 얻을 수 있도록 하는 도구의 대표적인 예이다. 최적화 도구와 같은 오프라인에서의 HPC 사용에 더해, 최근에는 HPC로 수집한 정보를 태스크의

프로파일로서 운영체제에 제공하여 성능, 에너지, 열 등을 고려한 스케줄링 기법에 활용하는 연구도 많이 진행되고 있다.

본 논문에서는 HPC에서 수집된 정보를 바탕으로 태스크 프로파일을 구성하여, 공유 자원의 경합을 줄이는 것에 활용하는 스케줄링 알고리즘을 제안하였다. 이를 위하여 HPC에서 얻을 수 있는 정보를 조합한 수식으로 태스크의 공유 자원의 사용 정도를 모델링 하였고, 이 모델을 적용하여 태스크 프로파일을 구성하였다. 또한 태스크 프로파일에 사용되는 HPC의 값을 적은 부하로 정확하게 수집하기 위해 커널 수준의 동적 태스크 프로파일러를 리눅스 커널에 구현하였다. 제안된 스케줄링 알고리즘은 태스크 프로파일을 바탕으로 현재 시스템의 공유 자원 사용 정도가 일정 수준 이하로 항상 유지될 수 있도록 수행될 태스크를 결정한다. 그 결과, 시스템 전체의 공유 자원 경합을 줄일 수 있고, 이것은 성능 향상으로 이어진다.

이후의 논문 구성은 다음과 같다. 2장에서는 관련 연구를 소개하며, 3장에서는 공유 자원 경합을 고려한 스케줄링 기법을 설명한다. 4장에서는 제안된 스케줄링 기법을 평가하여 본 후, 5장에서 본 논문의 결론과 향후 연구에 관하여 언급할 것이다

2. 관련 연구

멀티 프로세서에서 공유 자원을 효과적으로 관리하기 위한 스케줄링 기법은 오랜 시간 꾸준히 연구가 진행된 분야다. 특히 최근에는 하드웨어 수준의 멀티 쓰레딩 아키텍쳐가 제안됨에 따라 이러한 기술을 적용한 프로세서에서의 스케줄링 연구가 활발히 이루어졌다. 또한 CMP 환경이 고성능 프로세서 설계의 주류로 자리잡기 시작하면서, 공유 2차 캐시 경합 관리, 공유 버스의 대역폭 관리 스케줄링 기법들에 대한 연구도 많이 진행되었다.

McGregor는 공유 메모리 기반의 하드웨어 수준 멀티쓰레딩 프로세서를 탑재한 시스템에서는 공유 자원을 캐시로 한정하는 것이 충분하지 않다고 주장하였다.[2] 또한 그는 공유 캐시에 더해 CMP 환경에서는 공유 버스, 하드웨어 수준의 멀티쓰레딩 프로세서에서는 실행 유닛, 분기 예측기 등을 공유 자원으로 고려해야 함을 역설하였다. [2]에서는 HPC를 통해 수집한 버스 트랜젝션 비율의 합이 적절한 수준이 되도록 미리 태스크를 그룹화한 후, 다음 프로세서 시간 자원 할당 시점에 이 그룹 안의 쓰레드들을 하나씩 물리 코어에 할당하는 스케줄링 알고리즘을 제안하였다. 하지만 모든 쓰레드들의 스케줄링 타임 슬라이스가 같다는 가정을 갖고 있으므로, 일반적인 운영체제에 적용하기는 어렵다는 단점이 있다.

Fedorova는 CMP의 공유 자원 경합으로 인해 태스크의 성능이 주위 환경에 독립적이지 않고 함께 수행되는 태스크(co-runner)에 많은 영향을 받아 성능의 변동이 심할 수 있다는 문제를 발견하였다. 이것을 해결하기 위해 동적으로 태스크의 타임 슬라이스를 조정하는 기법[3]을 제안하였다. 동시에 수행되는 N개의 쓰레드가 비슷한 사이클 당 캐시 미스 수를 갖으면 공평한 공유 캐시 할당이 이루어졌다고 판단하는 FairIPC 모델을 바탕으로 타임 슬라이스를 조정하여 함께 수행되는 태스크에 상관없이 태스크 성능의 독립성을 향상 시켰다.

Knauerhase의 OBS-L[4] 기법은 하드웨어 성능 카운터를 운영체제 수준에서 관찰한 후, 그 정보를 스케줄링에 활용한다는 점에서 본 논문의 연구와 유사점이 많다. 하지만 각 코어의 최하위 캐시 메모리의 미스를 태스크의 프로파일로 사용하였는데, 이 과정에서 공유 버스 간섭은 전혀 고려하지 않았다는 점에 있어 본 논문에서 제안한 기법과 큰 차이가 있다.

3. 공유 자원 경합을 고려한 스케줄링 기법

3.1 CMP에서의 공유 자원 경합이 성능에 미치는 영향

본 절에선 첫 번째, SPEC2000[5] 벤치마크의 조합하여 태스크의 수행 시간이 함께 수행되는 태스크에 따라 어떠한 변화가 있는지 확인할 것이다. 두 번째, 다른 수준의 메모리 접근 정도 -낮은 정도의 메모리 접근 태스크 군, 중간 정도의 메모리 접근을 갖는 태스크 군, 높은 정도의 메모리 접근을 갖는 태스크 군- 를 갖는 태스크 집단들이 기존 리눅스 O(1) 스케줄러에 의해 스케줄링 될 때, 각 태스크 집단의 특성에 따라 성능 측면에서 어떠한 차이가 있는지 확인할 것이다. 이것은 본 논문의 핵심이 되는 연구 동기이다.

태스크의 수행 시간이 함께 수행되는 태스크에 따라 어떠한 변화가 있는지 확인하기 위해 다음과 같이 실험을 설계하였다. SPEC2000의 twolf 벤치마크를 수행 도중 다른 코어로 이동할 수 없도록 특정 코어에 고정시킨 후, 단독으로 수행될 때의 수행 시간을 관찰하였다. 그런 후에, SPEC2000의 다른 벤치마크인 bzip2, mcf, parser, gzip, crafty와 동시에 수행될 때의 twolf 수행 시간을 다시 관찰하였다. 이 경우에도 처음 할당 받은 코어에서만 끝날 때까지 수행되도록 벤치마크들을 코어에 고정하였다.

그림 1은 twolf 단독으로 수행했을 때의 수행 시간과 twolf와 다른 벤치마크를 조합하여 두 개의 응용이 동시에 수행된 경우에서의 twolf 수행 시간을 비교한 그래프이다. 비교의 편의를 위하여 단독으로 수행했을 때의 twolf 수행 시간을 1로 정의한 후, 그에 따라 나머지 실험의 수행 시간을 표준화하였다. 단독으로 수행한 경우보다 두 개의 벤치마크를 동시에 수행시켰을 때의 twolf 수행 시간이 더 길고, 그 비율

또한 함께 수행된 벤치마크에 따라 최소 1.14배부터 최대 2.49배까지 다양한 것을 그림을 통해 쉽게 확인할 수 있다. 즉, 두 개의 코어가 집약된 CMP 시스템에서의 twolf와 함께 수행될 짝은 주어진 벤치마크 집합에서는 crafty로 선정하는 것이 최선이다. 본 실험을 통해 CMP 내부의 각 코어에 할당된 태스크들은 공유 자원의 경합으로 인해 서로의 성능에 영향을 끼치고 있는 사실을 쉽게 유추할 수 있다.

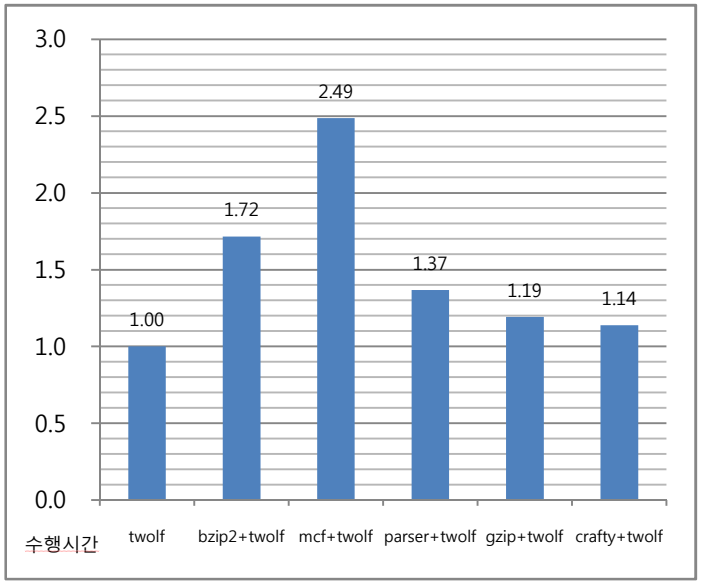

그림 1. twolf 벤치마크의 co-runner에 따른 수행 시간 변화

위의 실험에서 유추한 결론이 맞다면 공유 자원을 많이 사용하는 태스크들은 그만큼 다른 태스크의 영향을 받기 쉬우므로 수행 시간의 변동 폭이 더 클 것이며, 그렇지 않은 태스크는 수행 시간의 변동 폭이 작거나 거의 없을 것이다. 이러한 사실을 확인하기 위하여 메모리 접근 정도를 조절함으로써 공유 자원 사용 정도를 임의로 조정한 세 가지 태스크 군을 구성하였다. 낮은 메모리 접근 정도를 갖는 태스크 군1은 태스크군2와 태스크 군3에 비해 공유 2차 캐시와 공유 버스의 사용 정도가 적을 것으로 기대하고 있으며, 태스크군2 역시 태스크군3에 비해서는 상대적으로 적은 공유 자원 접근을 기대하고 있다. 각 태스크군에 속한 태스크를 4개씩 총 12개의 태스크들을 4 코어의 리눅스 시스템에서 반복하며 수행해보았다. 12개 태스크들은 각 코어에 할당되어 임의로 수행될 것이며 각 코어는 각자의 실행 큐에 세 개씩의 태스크를 관리하게 된다. 실행 큐 안의 모든 태스크들이 균일하게 스케줄링 기회를 갖고, 자원의 경합이 전혀 존재하지 않는다고 가정하면 각 태스크를 독립적으로 한 코어에서 수행했을 때의 수행 시간 대비 3배의 시간이 소요 될 것이다. 이때의 수행 시간을 이상적인 수행 시간이라고 정의하였으며 본 논문에서는 이상적인 수행 시간 대비 적용 스케줄러에서의 수행 시간 증가율을 스케줄러 성능 평가 기준으로 사용하였다. 이 평가 기준을 적용하여 O(1) 스케줄러를 평가한 결과는 그림 2와 같다.

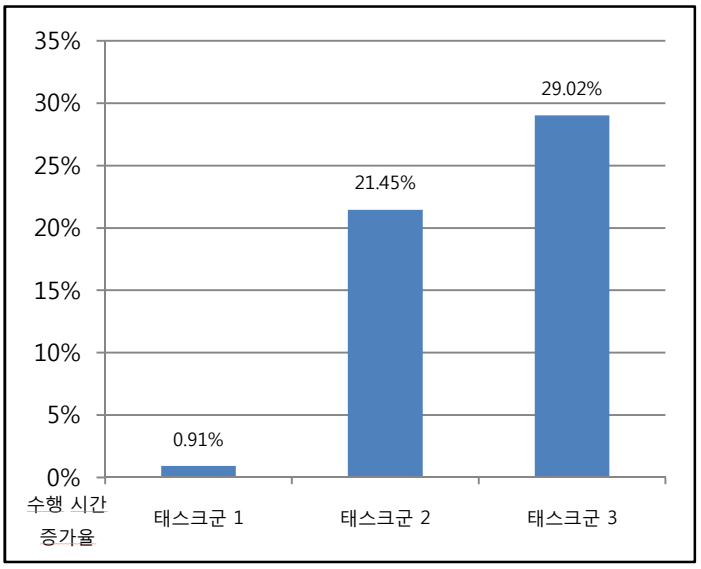

그림 2. 태스크 특성에 따른 성능 하락의 정도

기존 리눅스 O(1) 스케줄러에서는 공유 자원 접근이 많은 태스크일수록 이상적인 수행 시간 대비 수행 시간 증가율이 크다는 것을 그림 2에서 확인할 수 있다. 이것은 기존의 스케줄러가 CMP 환경의 특성을 전혀 고려하고 있지 않기 때문에 발생하는 현상이다. 태스크군3과 같이 공유 자원 접근이 많은 태스크들이 함께 스케줄링 되는 것을 피하고, 더 나아가 태스크군1처럼 함께 수행되는 태스크와의 자원 경합이 거의 존재하지 않는 태스크들을 상대적으로 많은 경합을 일으키는 태스크들과 함께 수행시키는 것으로 공유 자원 경합을 최소화 할 수 있다.

3.2 동시에 수행되는 태스크들의 상호 영향 정도 모델

하드웨어 성능 카운터 이벤트를 조합하여 태스크의 프로파일을 구성하는 방법은 이전의 많은 연구를 통해 이미 많이 알려진 방법이다. 본 연구에서도 ARM11 MPCore의 하드웨어 성능 카운터를 통해 얻을 수 있는 이벤트들 중, 데이터와 명령어 지연 사이클 값들의 분석을 통해 현재 코어에 스케줄링 될 후보 태스크들과 주변 코어에서 수행중인 태스크들 간의 상호 영향이 어떠한지에 대한 모델을 고안하였다. 데이터 의존에 의한 지연 사이클 이벤트 Cycle(DataStall)는 아래의 수식 1과 같이 정의할 수 있다.

(수식 1)

$$\begin{aligned} Cycle(DataStall) = {} & Num(DL2Hit) \times Cycle(L2Lat) \\ & + Num(DL2Miss) \times Cycle(MemLat) \\ & + Cycle(Bus) + Cycle(DTLB) \end{aligned}$$

데이터 의존에 의한 지연 사이클은 데이터 지역 1차 캐시 미스가 발생함으로 인해 소모되는 사이클과 데이터 변환 색인 버퍼 캐시에서 발생한 미스로 인해 소모되는 사이클로 나눌 수 있다. 하지만 데이터 변환 색인 버퍼 캐시 미스 관련 사이클은 다른 항에 비해 상대적으로 매우 작은 값이므로 무시할 수 있다.

데이터 2차 캐시에서 히트인 경우와 미스인 경우는 2차 캐시에 대한 지연 시간과 오프칩 메모리에 대한 지연 시간이 다르므로 다른 지연 사이클을 소모한다. 또한 모든 데이터 접근은 CMP 칩 내부의 공유 버스를 통해 이루어지므로, 버스에서 소모되는 사이클 역시 데이터 의존에 의한 지연 사이클에 포함할 수 있다. 대상 응용의 데이터 의존에 의한 지연 사이클은 태스크에 따라 고정된 값이 아니다. 다른 코어에서 수행 중인 태스크와의 공유 캐시의 경합으로 인해 더 많은 데이터 2차 캐시 미스(DL2Miss)를 발생시킬 수도 있으며, 다른 코어가 공유 버스를 사용의 완료할 때까지 기다려야 하는 경우도 발생한다. 따라서 위의 식에서 다른 코어에서 수행되는 태스크의 영향을 받는 부분과 그렇지 않은 부분을 분리할 필요가 있다.

(수식 2)

$$\begin{aligned}Cycle(DataStall) \approx\ & Num(DL2Hit) \times Cycle(L2Lat) \\ & + Num(DL2Miss) \times (Cycle(L2Lat) \\ & + Cycle(MemLat) - Cycle(L2Lat)) \\ & + Cycle(Bus) \\ \approx\ & Num(DL2Hit) \times Cycle(L2Lat) \\ & + Num(DL2Miss) \times Cycle(L2Lat) \\ & + Num(DL2Miss) \times (Cycle(MemLat) \\ & - Cycle(L2Lat)) + Cycle(Bus)\end{aligned}$$

상수 값은 더하거나 빼어도 전체 값에는 영향을 주지 않으므로, 오프칩 메모리에 접근하는데 소모되는 사이클 값을 공유 2차 캐시에 접근하는데 소모되는 사이클과 그 나머지 값으로 분리하였다. 또한 Num(DL2Miss)와 Cycle(Bus)는 동시에 다른 코어에서 수행되는 태스크에 의해 변동이 있는 값이므로 이 값이 클수록 태스크는 다른 코어에서 수행되는 태스크와 상호 영향 정도가 크다고 볼 수 있다. 따라서 정리하면 수식 3과 같다.

(수식 3)

$$\begin{aligned}Cycle(DataStall) - & Num(DL2Access) \times Cycle(L2Lat) \\ \approx\ & Num(DL2Miss) \times (Cycle(MemLat) - Cycle(L2Lat)) \\ & + Cycle(Bus)\end{aligned}$$

수식 3의 Num(DL2Access)는 MPCore에서 제공하는 데이터 지역 1차 캐시 미스 수와 같은 값이므로 쉽게 얻을 수 있다. 명령어 지역 1차 캐시 미스와 명령어 변환 색인 버퍼 미스에 연관된 지연 사이클 Cycle(InstructionStall)도 앞의 방식과 유사하게 전개하여 수식을 만들 수 있다. CMP 환경에서 동시에 수행되는 태스크들의 상호 영향 정도는 태스크가 CPU 시간 자원을 할당 받을 때마다의 총 수행 사이클에서 동시에 수행 중인 다른 코어의 태스크 특성과 상호 연관된 사이클 값이 차지하는 비율이며, 커널 수준의 동적 태스크 프로파일러는 수행 사이클 값과 4개의 일반 성능 정보 이벤트를 매 스케줄링 타임 슬라이스 단위로 수집하여 프로파일링 평가 기준인 수식 4를 계산한다.

(수식 4)

$$\frac{(Cycle(DataStallRelated) + Cycle(InstructionStallRelated))}{Cycle(EachTimeSlice)}$$

Num(DL2Hit)	전체 데이터 지역 1차 캐시 미스 중 공유 2차 캐시에서 히트가 발생하는 횟수
Num(DL2Miss)	전체 데이터 지역 1차 캐시 미스 중 공유 2차 캐시에서 미스 나는 횟수
Cycle(L2Lat)	공유 2차 캐시에 접근하는데 소모되는 사이클 (상수)
Cycle(MemLat)	오프칩 메모리에 접근하는데 소모되는 사이클 (상수)
Cycle(Bus)	공유 버스에서 소모되는 사이클
Cycle(DTLB)	데이터 변환 색인 버퍼 미스로 인해 소모되는 사이클

표 1. 수식의 각 항에 대한 설명

그림 3은 SPEC2000 벤치마크의 프로파일링 평가 기준 값을 나타낸 것이다. twolf를 단독으로 수행하였을 때에 대비 수행 시간 증가 비율에 대한 경향인 그림 1과 그림 3의 경향은 매우 유사함을 알 수 있다. 따라서 본 모델이 CMP 내부에서 수행되는 태스크 간의 상호 영향 정도를 잘 나타낸 것으로 볼 수 있다..

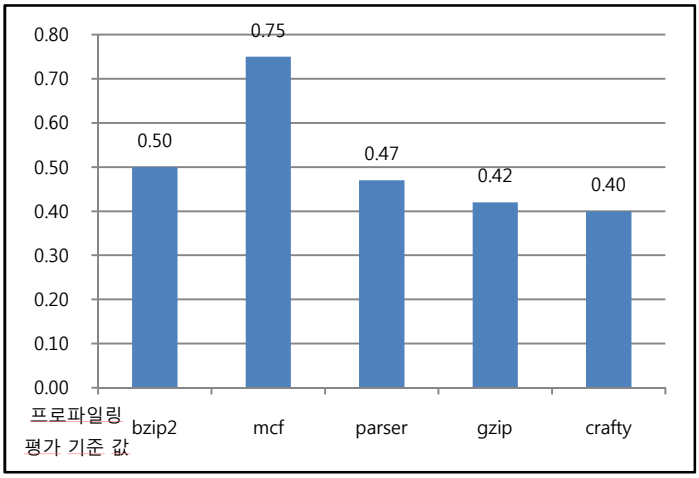

그림 3. SPEC2000 벤치마크의 프로파일링 평가 기준 값

동적 태스크 프로파일러는 앞서 언급한 모델에 필요한 HPC 값을 수집하는 역할을 담당한다. 정확한 태스크 프로파일을 위해서는 낮은 오버헤드로 HPC 값을 측정하는 것이 매우 중요하다. 본 연구에서는 이전 연구[6]에서 구현하여 평가하였던, 커널 수준의 동적 태스크 프로파일러를 사용하였다. ARM11 MPCore에서 제공하는 카운터의 수가 필요한 HPC 이벤트의 수보다 적으므로, 멀티플렉싱 기능을 활성화하여 정보를 수집하였으며, 오차율과 수행 시간 증가율이 모두 1% 이내인 5ms를 샘플링 주기로 사용하였다.

Scheduling Algoritm

```
# sum the profile metrics of all cores excepting tareget core
for i=0 to NUM_CORE
    met_sum += mrtics[i]
    if i == NUM_TARGET continue
end for
if system_min ≤ metsum < system_thr1
    for i=0 to NUM_OF_TASKS_IN_THE_Q
        if (q[i].metric > task_thr2) return q[i]
    end for
    for i=0 to NUM_OF_TASKS_IN_THE_Q
        if (q[i].metric > task_thr1) return q[i]
    end for
    return q[0]
else if system_thr1 ≤ metsum < system_thr2
    for i=0 to NUM_OF_TASKS_IN_THE_Q
        if (task_thr1 ≤ q[i].metric < task_thr2) return q[i]
    end for
    for i=0 to NUM_OF_TASKS_IN_THE_Q
        if (task_thr2 ≤ q[i].metric < task_thr3) return q[i]
    end for
    return q[0]
else if system_thr2 ≤ metsum ≤ max
    for i=0 to NUM_OF_TASKS_IN_THE_Q
        if (task_min ≤ q[i].metric < task_thr1) return q[i]
    end for
    for i=0 to NUM_OF_TASKS_IN_THE_Q
        if (task_thr1 ≤ q[i].metric < task_thr2) return q[i]
    end for
    for i=0 to NUM_OF_TASKS_IN_THE_Q
        if (task_thr2 ≤ q[i].metric < task_thr3) return q[i]
    end for
    return q[0]
```

그림 4. 공유 자원 경합을 고려한 스케줄링 알고리즘

3.3 공유 자원 경합을 고려한 스케줄러

CMP 환경의 공유 자원 경합을 고려하기 위해서 기존의 리눅스 O(1) 스케줄러에 사용되는 우선 순위 기반 다음 태스크 결정 과정에 스케줄링 대상이 되는 코어를 제외한 시스템 전체의 자원 경합 정도를 반영하여 태스크를 선정하는 메커니즘을 추가하였다. 시스템의 자원 경합 정도는 스케줄링 대상이 아닌 CMP 내부의 다른 코어에서 수행 중인 태스크들의 프로파일링 평가 기준 값의 합으로 알 수 있으며, 그 정도에 따라 최소값(system_min)과 최대값(system_ max)로 사이에 두 개의 임계값을 두어 총 3가지 구간으로 구분하였다. 스케줄러는 각 구간에 따라 스케줄링 대상이 되는 코어의 실행 큐에서 적절한 태스크를 선정하게 되며 아래 그림 4에 그 알고리즘을 소개하였다.

4. 실험 및 평가

4.1 실험 환경

본 논문에서 제안한 스케줄링 기법과 동적 태스크 프로파일러는 모두 ARM11 MPCore 시스템을 대상으로 하였으며, 이 시스템에서 동작하는 리눅스 운영체제 커널에 구현하여 실험 및 평가 하였다.

Paramemter	Value	
Processor	4-way CMP (210 MHz)	
Cache	L1 Inst/Data	32KB, 4-way, 2 cycle latency
	Shared L2	1MB,16-way, 8 cycle latency
OS Kernel	Linux-2.6.17-arm1-smp28	

표 2. 실험 파라미터

4.2 공유 자원 경합을 고려한 스케줄러의 성능 평가

제안한 스케줄링 기법은 공유 자원의 경합이 전혀 존재하지 않는다고 가정한 이상적인 수행 시간과 비교함으로써 그 성능을 평가하였다. 시스템이 포함하고 있는 코어의 개수의 N배에 해당하는 태스크 수를 실험 진행 동안 항상 유지 시킴으로 스케줄러가 처리하는 공유 자원 경합의 정도를 항상 일정하게 하였고, 이 환경에서의 공유 자원 경합이 전혀 없는 이상적인 수행 시간은 단독으로 수행했을 때의 수행 시간에 N배로 정하였다. 다양한 특성의 마이크로 벤치마크에서의 성능 평가를 위하여 세가지 특성의 메모리 접근 정도를 갖는 3개의 태스크군 12개의 태스크에 대하여 실험하였다. 3.1절에서 언급하였던 것과 같이 태스크군3이 가장 공유 자원에 대한 접근 정도가 큰 집단이며 태스크군1은 공유 자

원에 거의 접근하지 않는 집단이다. 본 실험에서의 N 값은 3이므로, 각 태스크군의 태스크를 단독으로 수행시켰을 때의 3배에 해당하는 수행 시간이 이상적인 수행 시간이다.

그림 5는 기존 리눅스 스케줄러와 제안 스케줄러에서 수행된 벤치마크의 수행 시간이 이상적인 수행 시간 대비 얼마나 증가하였는지를 나타내고 있다. 많은 공유 자원 경합을 발생시키는 태스크군 3에서 5.96%의 성능 하락 현상을 줄였으며, 공유 자원 경합을 상대적으로 많이 일으키지 않는 태스크군 2의 성능 하락 또한 2.50% 감소시켰다. 또한 공유 자원 경합을 많이 일으키는 태스크 군3이 제안 스케줄러에서 서로 같이 수행되는 것이 효과적으로 회피되고 있는지 확인하기 위하여 각각 6개의 태스크로 이루어진 태스크군1과 태스크군3의 조합을 실험하였다. 그 결과는 그림 6에서 확인할 수 있다.

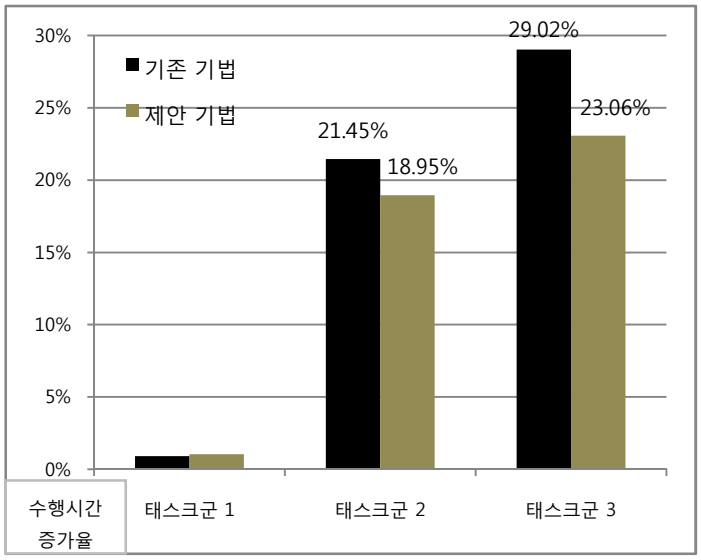

그림 5. 이상적인 수행 시간 대비 각 스케줄러의 태스크군별 수행 시간 증가 비율 (1)

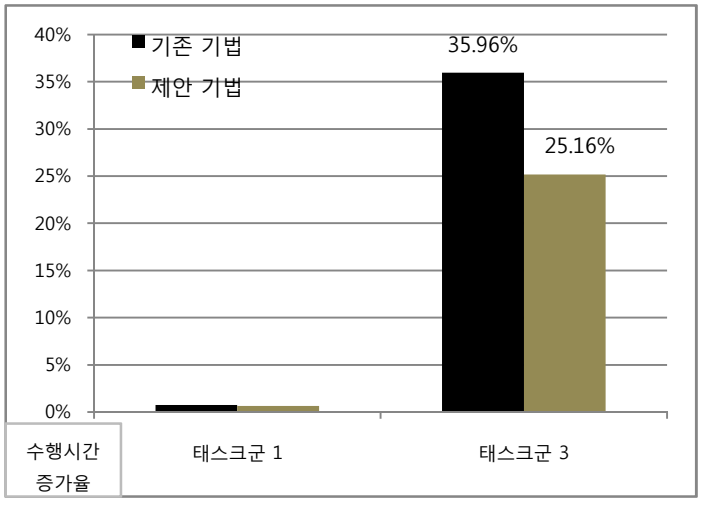

그림 6. 이상적인 수행 시간 대비 각 스케줄러의 태스크군별 수행 시간 증가 비율 (2)

5. 결론 및 향후 연구

CMP 환경은 하나의 칩에 여러 개의 마이크로프로세서를 집약함으로써 성능 향상의 이득을 얻고자 하는 시스템이다. 이러한 환경에서의 각 코어는 최하위 수준의 캐시와 시스템 공유 버스 등의 자원을 공유하게 되므로 자원 경합의 문제가 발생하게 되고 이것은 성능의 하락으로 이어진다. 본 논문에서는 이러한 문제를 인식하고 그에 대한 해결책으로 공유 자원의 경합을 고려한 CMP용 스케줄러를 제안하였다. 제안된 공유 자원 경합을 고려한 스케줄러는 기존의 리눅스 스케줄러에 동시에 수행되는 태스크들의 상호 영향 정도 모델을 적용한 것으로 공유 자원 경합으로 인한 성능 하락을 최대 10.8% 감소시켰다. 향후에는 본 논문에서 주요 문제로 삼았던, 공유 자원 경합으로 인한 성능 하락 현상에 추가로 각 태스크의 QoS와 fairness에 공유 자원 경합이 미치는 영향을 분석하여 모델링하고 그 모델을 적용한 스케줄링 알고리즘을 연구를 계속할 방침이다.

감사의 글

이 연구를 위해 연구장비를 지원하고 공간을 제공한 서울대학교 컴퓨터 연구소에 감사드립니다. 이 논문은 2009년도 두뇌한국21사업에 의하여 지원되었으며, 2009년도 정부(교육과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구(No. R0A-2007-000-20116-0, R33-2008-000-10095-0)입니다.

참고 문헌

[1] Performance data standard API. http://icl.cs.edu/projects/papi/, 2000.

[2] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling Algorithms for Effective Thread Pairing on Hybrid Multiprocessors. In Proceedings of Parallel and Distributed Processing Symposium, 2005.

[3] A. Fedorova, M. Seitzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In Proceedings of Parallel Architecture and Compilation Techniques, 2007.

[4] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. IEEE MICRO, 2008.

[5] SPEC CPU2000 Benchmark, http://www.spec.org/.

[6] 송욱, 송지석, 김지홍. 하드웨어 성능 카운터에 기반한 커널 수준의 동적 태스크 프로파일링 모듈의 설계 및 구현, 2008 정보통신분야학회 합동학술대회 논문집, 2008.