대칭형 멀티 코어 아키텍처를 위한 효율적인 부하 분산 알고리즘

홍석일○ 국중진 안양근 홍지만
숭실대학교 컴퓨터학과
hongsukil@gmail.com○, tipsiness@gmail.com, ykahn@keti.re.kr, jiman@ssu.ac.kr

An Efficient Load Balancing Algorithm for Symmetric Multi-Core Architectures

Sukil Hong○ Joongjin Kook Yangkeun Ahn Jiman Hong
Soongsil University

요 약

컴퓨터의 성능을 향상하기 위해 주로 프로세서의 성능을 높여 왔으나 발열 및 집적도 등의 문제로 인하여 한계를 가지게 되었고, 이를 해결하기 위하여 멀티 프로세서와 멀티 코어 시스템이 등장하였다. 이러한 시스템은 두 개 이상의 처리기를 사용하여 단일 처리기 시스템보다 높은 성능을 갖으며 비교적 낮은 전력을 소모하기 때문에 점차 사용이 증가하고 있다. 운영체제도 이러한 다중 처리기 시스템을 위한 기능이 추가 되어 효율적으로 사용하여 성능을 높이기 위해 변화하고 있다. 부하 분산 알고리즘 역시 예전의 스케줄러에는 들어있지 않는 기법이었으나 멀티 프로세서가 등장한 이후로 추가 되었다. 본 논문에서는 이전 방식의 부하 분산 알고리즘에 유동적인 기준점을 추가하여 성능을 개선하고자 한다.

1. 서 론

시스템의 성능을 높이기 위하여 프로세서 성능 개발을 끊임없이 진행하고 있다. 몇 년 전까지의 개발 방법이 클록 스피드를 통한 속도 향상과 고집적을 통한 대용량화였다면, 이제는 처리기의 개수를 늘림으로써 성능을 향상시키고 있다. 기존의 단일 처리기 시스템에서 다수의 처리기를 사용한 멀티 프로세서 시스템과 멀티 코어 시스템이 주된 방법으로 사용되고 있다. 단일 처리기의 발전이 한계점을 가지는 몇 가지 근거가 있다.

첫 번째 문제는 메모리 장치와의 속도 차이이다. 데이터 전송대역폭 측면에서 볼 때 처리기와 메모리 장치사이에는 커다란 속도 차이가 존재한다. 평균적으로 처리기 작업의 75% 정도가 메모리 장치에 대한 응답을 기다리기 위해 사용된다. 이러한 속도 차이에 의한 대기시간을 줄이기 위해 캐시나 TCM(Tightly Coupled Memory)를 사용하지만 가격이 고가이고, 사이즈에 비례하여 성능향상이 되지 않기 때문에 비효율적이다.

두 번째 문제는 소비전력과 발열에 대한 문제이다. 성능 향상을 위해 처리기 및 주변 장치의 클록 스피드를 올리게 되면 성능 향상 효과는 그렇게 크지 않은 반면 소비전력은 큰 폭으로 증가하게 된다. 또한 주변 장치를 추가할수록 집적도가 높아지면서 높은 발열을 발생하게 된다.

세 번째 문제는 파이프라인이다. 처리기의 성능이 향상됨에 따라 파이프라인이 복잡해지고 길어진다. 프로그램 내에서 분기를 수행할 경우 파이프가 깨지거나, 운영체제에서 태스크 문맥 교환이 이루어지면서 파이프라인이 깨질 가능성이 높아진다. 특히 응용 프로그램과 시스템 관리를 위한 태스크가 점점 늘어남에 따라 이러한 현상이 많아지고 성능저하를 가져오게 된다.

이러한 문제를 해결하기 위해 멀티 프로세서 시스템과 멀티 코어 시스템이 등장하였고 이미 데스크탑 PC나 임베디드 시스템에 사용이 늘어가고 있다. 멀티 프로세서 시스템과 멀티 코어 시스템은 다수의 처리기를 가지고 있다는 점에서 공통점을 가지지만 시스템 형태와 동작면에서 차이점을 가지고 있다[1].

단일 처리기 시스템을 관리하던 운영체제 역시 다수의 처리기 시스템을 위해 몇 가지 기능이 추가되었다. 그중의 하나가 처리기에 할당된 태스크들의 부하 분산을 위한 기능으로 시스템에서 운영 중인 다수의 태스크들의 효율적인 처리를 위하여 운영체제가 관리를 하는 것이다. 시스템에 존재하는 많은 태스크들이 다수의 처리기가 존재하면서도 하나의 처리기에만 부하가 집중되는 것을 피하고 여러 처리기를 이용하여 태스크 처리를 분산함으로써 태스크 처리 효율을 높이고 있다. 하지만 리눅스 스케줄러에는 이러한 부하 분산을 처리함에 있어 태스크 부하 정도를 비교하는 기준에 문제가 있다. 전체 태스크의 수, 실행 큐의 태스크 분산 량, 편중 정도 등 주변 상황이 모두 다르지만 현재에는 이러한 상황을 고려하지 않고 태스크 수만 비교하여 부하 분산 알고리즘

을 수행하도록 한다. 결국 부하 분산을 수행하는 오버헤드를 생각할 경우, 수행하지 않아도 될 경우에도 특정한 기준 때문에 수행하도록 되어있다.

본 논문에서는 부하 분산 알고리즘을 수행하면서 비교하고자 하는 실행 큐들의 태스크 비율을 전체 태스크 수에 따라 고려하는 가변적인 기준을 정의하고 이 기준에 따라 부하 분산을 수행하는 효율적인 부하 분산 알고리즘을 제안하고자 한다.

2. 관련 연구

본 장에서는 논문에서 언급한 멀티 프로세서 시스템과 멀티 코어 시스템의 특성과 차이점을 확인하고 부하 분산 알고리즘 사용 및 종류에 대해 알아보도록 한다. 또한 리눅스 스케줄러에 정의된 일반적인 부하 분산 알고리즘에 대해 조사하고 수행 시점 및 수행 흐름에 대해 파악한다.

2.1 멀티 프로세서 및 멀티 코어 시스템

멀티 프로세서 시스템과 멀티 코어 시스템이 등장하기 이전에는 네트워크를 이용하여 다수의 처리기를 사용하는 분산 시스템이 있었다. 분산 시스템은 단일 처리기 시스템보다 뛰어난 성능을 가지고[2], 분산 시스템을 이용하여 단일 처리기에서 동작하기 힘든 작업이나 서버급의 용도로 사용하였다. 또한 논문에서 언급하는 태스크 부하 분산 알고리즘을 사용하여 네트워크 환경에서 처리기의 효율성을 증가시켰다[3].

점차 단일 처리기의 발전에는 한계가 드러나면서 멀티 프로세서 시스템이 활성화되었다. 멀티 프로세서 시스템이 단일 처리기 시스템보다 성능이 뛰어나고[4][5][6], 발열 문제와 집적도 및 비용 등에서도 좋은 효과를 얻었다. 또한 멀티 코어 시스템은 이미 주요 서버 및 PC 공급 업체들에 의해 활용되고 있다. 멀티 코어란 두 개 이상의 독립 코어를 단일 집적 회로로 이루어진 하나의 패키지로 통합한 것이다[7]. [표 2-1]은 멀티 코어 시스템과 멀티 프로세서 시스템의 가장 큰 차이점을 비교한 것이다.

[표 2-1] 멀티 코어와 멀티 프로세서 비교

구 분	멀티 코어	멀티 프로세서
다이	통합	통합
캐시	독립적	독립적
인스트럭션 메모리	공유함	공유하지 않음
프로세스	마스터에 종속적	독립적

멀티 코어는 시스템 메모리를 다수 개의 코어가 공유하는 시스템 인스트럭션 메모리를 공유하기 때문에 동시에 두 개의 인스트럭션 실행이 불가능하다. 그에 반해 멀티 프로세서는 시스템 메모리를 공유하지 않고 인스트럭션 메모리를 공유하지 않기 때문에 동시에 서로 다른 인스트럭션 수행이 가능하다.

2.2 부하 분산 알고리즘

2.2.1 Dynamic Thermal & Power Management

2000년 초부터 발열에 대한 관심이 높아졌다. 다수의 처리기 중 하나의 처리기에 부하가 생기면 높은 열이 발생하게 되고 주위의 처리기에 영향을 미침으로써 결과적으로 전체 처리기에 나쁜 영향을 미치기 된다. 이러한 발열을 방지하기 위하여 부하 분산 알고리즘을 사용하고 태스크의 처리에 효율성을 높이도록 한다[8][9][10]. 또한 각 처리기의 소비전력을 확인하고 태스크의 부하가 집중되는 것을 방지하고자 한다[11]. 태스크의 부하가 많은 처리기는 평균적으로 소모하는 소비전력 이상을 사용하기 때문에 에너지 사용면에서 비효율적이다.

2.2.2 Asymmetric Multicore / Multiprocessor Scheduler

본 논문에서 사용하는 대칭 구조의 멀티 코어 시스템이 아닌 비대칭 구조의 멀티 코어 시스템에서도 부하 분산 알고리즘을 연구하였다[12][13]. 성능이 다른 처리기가 존재할 경우 태스크의 처리 효율을 높이기 위해 성능이 좋은 처리기에 더 많은 태스크를 분배하도록 한다. 이러한 기법을 사용하기 위해 NUMA 시스템을 사용하여 태스크의 이동에 대한 오버헤드를 최소화하도록 하고 처리기의 성능에 따라 태스크 수를 조정하도록 한다.

2.3 리눅스 스케줄러의 부하 분산 알고리즘

2.3.1 실행큐

Linux 운영체제는 2.6 커널 중반 이후부터 스케줄러에 태스크 부하 분산 알고리즘을 추가하여 처리기의 실행 큐의 태스크 부하를 분산시킴으로써 태스크 처리에 대한 효율을 높이고자 한다[14].

Linux에서 사용하는 스케줄러는 실행 큐를 기반으로 구성되어있다. 실행 큐란 어떤 처리기에 있는 실행 가능한 태스크의 목록이고, 스케줄러에서 가장 중요한 자료구조이다. 또한 시스템 내의 모든 처리기는 자신의 실행 큐를 가지고 모든 실행 큐 자료 구조는 처리기 마다 할당되어있는 변수 *runqueues* 에 저장된다. [표 2-2]는 실행 큐 자료구조의 주요 멤버 변수를 나타내고 있다.

[표 2-2] 실행 큐 (runqueues) 구조체 멤버 변수

Data Type	Name	Description
spinlock_t	lock	실행큐를 보호하는 스핀락
unsigned long	nr_running	실행 가능한 태스크의 수
unsigned long	cpu_load	실행 큐 내 태스크의 평균수에 기초한 cpu 부하 인자
unsigned long long	nr_switches	cpu에 의해 수행된 태스크 전환 횟수
unsigned long	nr_uninterruptible	TASK_UNINTERRUPTIBLE 상태에서 자고 있는 태스크의 수
unsigned long	expired_timestamp	리스트 내 가장 오래된 태스크의 삽입 시간

2.3.2 부하 분산 알고리즘

실행 큐는 각 처리기에 종속되어 구분되므로 스케줄링 역시 각 처리기에 종속되어 이루어진다. 또한 각 처리기의 실행 큐에는 임의적으로 태스크가 존재하기 때문에 여러 처리기의 실행 큐 가운데 실행 큐의 태스크 분산 부하가 불균형할 수 있다. 로드 밸런서는 불균형한 실행 큐의 균형을 맞추는 역할을 한다. 로드 밸런서는 현재 처리기의 실행 큐를 시스템의 다른 처리기의 실행 큐와 비교하여 균형이 맞지 않을 경우 바쁜 실행 큐, 즉 태스크의 수가 많은 실행 큐에서 태스크를 빼내어 현재 실행 큐로 이동시킨다. Linux의 로드 밸런서는 kernel/sched.c 소스 안에 구현되어 있고 기본적인 수행 절차는 [그림 2-1]과 같다.

1. find_busiest_group() 함수와 find_busiest_queue() 함수를 호출
- 부하가 집중된 처리기 그룹과 실행 큐 검색

2. 가장 바쁜 실행 큐의 우선순위 배열에서 태스크 선택
- 우선순위 배열은 활성(active) 배열과 비활성(expired) 배열 두 가지로 나뉘어 있음
- 처리기 캐시에 없을 가능성이 높은 비활성 배열 선호
- 비활성 배열이 비어있다면 활성 배열에서 선택

3. 가장 높은 우선순위 태스크를 선택
- 높은 우선순위의 태스크를 공평히 나누어 조금이라도 빨리 처리하기 위함

4. 태스크 검색 조건
- 실행중이 아니고, 다른 처리기로 이동이 금지 되지 않은 태스크

5. pull_task() 함수를 호출하여 바쁜 실행 큐로부터 현재 실행 큐로 태스크 이동

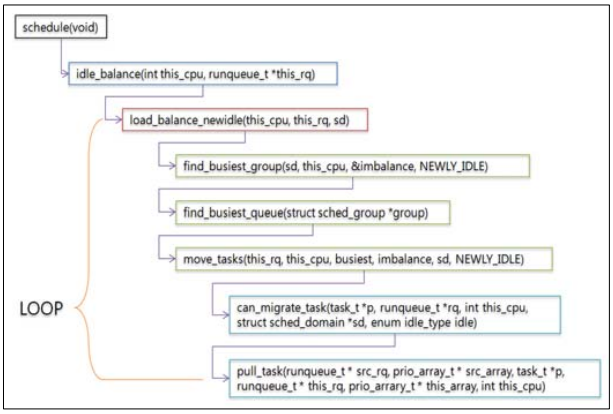
6. 실행 큐의 균형 상태를 확인하고 균형이 이루어 질 때까지 3 ~ 5번 과정 반복

7. 실행 큐의 불균형이 해제되면 현재 실행 큐의 잠금을 해제하고 알고리즘 종료

[그림 2-1] 일반적인 부하 분산 알고리즘

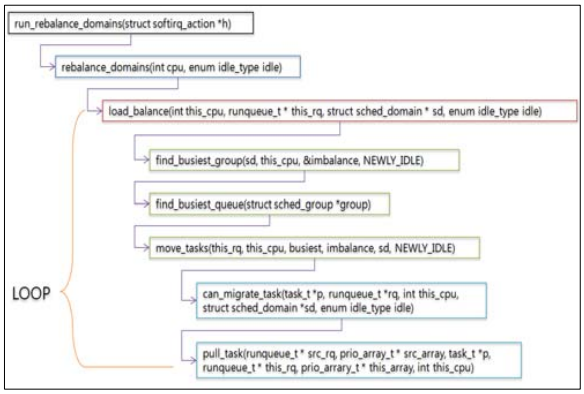
2.3.3 부하 분산 알고리즘의 수행 흐름

부하 분산 알고리즘은 세 가지 수행 경로가 있다. 첫 번째 방법은 schedule() 함수 내에서 함수 호출로 인한 실행이다. [그림 2-2]는 schedule() 함수로부터 수행되는 과정을 요약한 함수 흐름도이다. idle_balance() 함수를 호출한 이후 프로세서들의 실행 큐 부하 정도를 체크하여 load_balance_newidle() 함수를 호출하며 루프를 돌면서 처리한다. 호출된 load_balance_newidle() 함수에서 find_busiest_group() 함수와 find_busiest_queue() 함수를 통하여 가장 바쁜 실행 큐를 찾아낸다. 이때 조건에 맞는 실행 큐가 존재한다면 move_tasks() 함수를 호출하고 can_migrate_task() 함수를 통하여 이동 가능한 태스크를 체크하고 pull_tasks() 함수를 통하여 태스크를 이동시킨다.



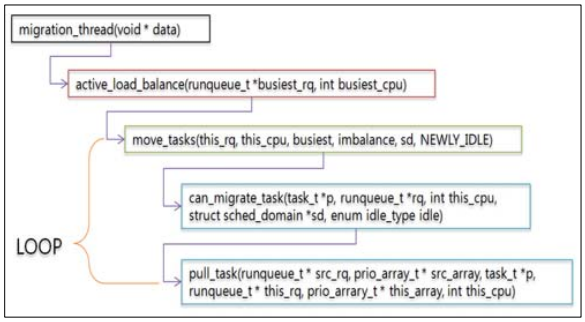
[그림 2-2] schedule() 함수를 통한 부하 분산 알고리즘 수행 루틴

두 번째 방법은 커널 타이머 핸들러로 등록되어 실행하는 것이다. CPU가 idle 상태일 경우는 1ms 마다 수행되고 그 외의 경우엔 200ms 마다 수행한다. [그림 2-3]은 타이머 핸들러로부터 수행되는 과정을 요약한 함수 흐름도이다. 타이머 핸들러로 등록된 run_rebalance_domains() 함수로부터 rebalance_domains() 함수를 호출하고 이후에 load_balance() 함수를 호출한다. 그 이후의 과정은 첫 번째 schedule() 호출에서의 과정과 동일하다.



[그림 2-3] 타이머 핸들러를 통한 부하 분산 알고리즘 수행 루틴

[그림 2-4]는 세 번째 방법으로 커널 쓰레드로의 호출로 인한 실행이다. 커널 쓰레드 함수인 migration_thread() 함수에서 active_load_balance() 함수를 호출하여 부하 분산 알고리즘을 수행한다. 이 경우는 앞의 두 가지 방법과 find_busiest_group() 함수와 find_busiest_queue() 함수를 사용하지 않는 차이점이 있다. active_load_balance() 함수에서는 sched_group 구조체 안에 있는 cpumask 멤버 변수를 통하여 group과 실행 큐를 선택한다. 선택한 실행 큐를 이용하여 move_tasks() 함수를 호출한 다음은 앞의 두 가지 방법과 동일하다.

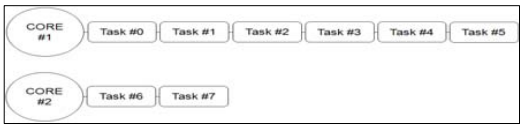


[그림 2-4] 커널 쓰레드로 호출되는 부하 분산 알고리즘 수행 루틴

3. 효율적인 부하 분산 알고리즘

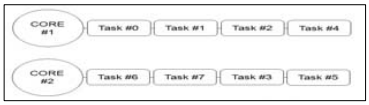
3.1 부하 분산 알고리즘의 수행 기준

리눅스 2.6커널은 다수의 처리기에 대하여 도메인과 그룹이라는 단위를 사용하여 관리한다. 태스크 부하 분산 알고리즘은 여러 처리기의 실행 큐를 모두 검색하여 가장 바쁜 실행 큐를 찾아내고 선택한 실행 큐에서 이동할 태스크를 선택한다. 예를 들어, [그림 3-1]은 부하 분산 알고리즘을 수행하기 전의 실행 큐 상황을 가정하고 있다. 각 코어의 task #0번은 현재 수행중인 태스크이고 나머지 태스크들은 대기 상태이다. 코어 1번의 실행 큐에는 1개의 실행중인 태스크와 5개의 대기 중인 태스크가 있고, 코어 2번의 실행 큐에는 1개의 실행중인 태스크와 1개의 대기 중인 태스크가 있다.



[그림 3-1] 부하 분산 알고리즘 수행 전

[그림 3-2]는 부하 분산 알고리즘을 수행 한 후의 실행 큐 상황이다. 코어 1번에서 task #3번과 task #5번이 이동하였고, 순서대로 이동하지 않고 임의의 태스크가 옮겨진 이유는 앞에서 언급했던 수행중이 아닌 태스크 중에서 높은 우선순위와 이동 금지되지 않은 태스크 등의 기준에 따르기 때문이다.



[그림 3-2] 부하 분산 알고리즘 수행 후

기존 알고리즘에서는 가장 바쁜 실행 큐를 선택할 때 실행 큐의 태스크의 수를 비교하여 실행 큐를 선택하도록 되어있다. 이러한 비교 기준은 상황에 따라 바뀌어야 하지만 현재의 알고리즘은 단순히 가장 많은 태스크를 가진 실행 큐를 선택하도록 되어있다. 하지만 태스크의 수만 가지고 실행 큐를 비교하는 것은 효율적이지 않다. 가령 태스크의 수가 별로 차이가 없거나 너무 적을 경우엔 부하 분산 알고리즘을 수행하는 오버헤드와 태스크를 이동하면서 생기는 오버헤드가 시스템의 성능에 오히려 나쁜 영향을 줄 수 있다.

[그림 3-3]은 기존 알고리즘에 있는 실행 큐에 존재하는 태스크 수를 비교하여 바쁜 실행 큐를 선택하는 방법에 대한 소스이다. 이 부분의 소스에 전달된 'group' 인자는 이전의 수행에서 프로세서 그룹 단위에서 태스크의 수, 전력 등의 비교를 통하여 부하가 집중된 그룹을 선택한다. 전달된 'group' 인자를 통하여 그룹 내의 모든 처리기의 실행 큐에 접근 가능하고 각 실행 큐의 로드된 태스크 수 비교 후 태스크의 수가 하나라도 많은 실행 큐를 바쁜 큐로 선택한다.

```
for_each_cpu_mask(i, group->cpumask) {

        if (!cpu_isset(i, *cpus))
                continue;

        rq = cpu_rq(i);

        if (rq->nr_running == 1 &&
            rq->raw_weighted_load
          > imbalance)
                continue;

        if (rq->raw_weighted_load > max_load) {
                max_load = rq->raw_weighted_load;
                busiest = rq;
        }
}
```

[그림 3-3] 태스크 수의 비교를 통한 실행 큐 선택 알고리즘

3.2 실행 큐 선택의 가변적인 기준

논문에서 제안하는 태스크 부하 분산 알고리즘은 전체 태스크 수에 따라 태스크 부하의 비율을 결정하고 비율에 따라 실행 큐의 부하 정도를 결정한다. 전체 태스크 수에 대한 정보는 커널에 전역 변수를 선언하고 태스크의 생성 시 1씩 증가하고 태스크의 종료 시 1씩 감소시키며 유지한다. 전체 태스크 수에 따라 10%부터 90%까지 태스크 부하 비율을 선택 하도록 되어있으며 선택한 비율을 통하여 태스크 수의 부하 정도를 계산한다. 만약 태스크의 수가 많은 실행 큐가 있더라도 비율보다 적은 수가 있다면 그 실행 큐는 선택하지 않는다. [그림 3-4]는 본 논문에서 제안하는 태스크 부하 분산 알고리즘이다.

1. 전체 태스크 수 확인
2. 태스크의 부하 비율 결정
3. 현재 실행 큐의 태스크 비율을 계산
4. 다른 실행 큐의 태스크 수와 비교
5. 비율보다 많은 태스크를 가진 실행 큐 선택
6. 선택할 실행 큐가 없을 경우 알고리즘 종료

[그림 3-4] 논문에서 제안하는 태스크 부하 분산 알고리즘

가령 전체 태스크 수가 100개 일 경우 부하 비율이 30%라고 가정한다면, 현재 실행 큐의 태스크 수가 10개 일 경우 부하 비율에 따라 최소한 13개 이상의 태스크를 가지고 있는 실행 큐를 선택하여야 한다. 기존의 알고리즘은 11~12개의 태스크를 가진 실행 큐가 있을 경우 태

스크 이동을 위한 실행 큐로 선택하지만 비율을 적용함으로써 적은 수의 태스크 차이는 무시하도록 한다.

4. 성능 평가

4.1 실험 환경

4.1.1 시스템 환경

[표 4-1]은 본 논문에서 사용한 시스템 환경이다. 페도라 코어 6을 설치한 데스크탑 PC에서 실험을 수행하였다.

[표 4-1] 시스템 환경

H/W	
CPU	Intel Core 2 CPU 6400
RAM	2 G
S/W	
OS	Fedora Core #6

4.1.2 커널 수정

실험을 수행하고 결과를 측정하기 위해 커널을 수정하였다. 실험을 위하여 수정한 부분은 실험 수행 플래그 변수를 두어 설정할 수 있도록 하였다. [표 4-2]는 실험을 위하여 커널에 수정한 내용이다.

[표 4-2] 커널 수정

플래그 변수	
int experiment_flag	실험 수행 루틴에 대한 활성 / 비활성 설정변수 - #define EXP_ON 1 - #define EXP_OFF 0
카운팅 변수	
unsigned long CNT[3]	load balancing 수행 빈도수 체크
int balance_type	load balancing 발생 위치 - #define SCHDULE_ROUTINE 0 - #define TIMER_TICK_ROUTINE 1 - #define KERNEL_THREAD_ROUTINE 2
시스템 콜 작성	
sys_exp_flag_on()	실험의 활성화 및 데이터 공간 할당 및 초기화
sys_exp_flag_off()	실험의 비활성화 및 데이터 공간 해제
sys_read_cnt()	load balancing 수행 빈도수 확인
sys_read_data()	실험에서 저장한 태스크 정보 확인
sys_process_check()	task_struct에 추가한 플래그 변수 설정

커널 부팅 후 기존의 리눅스로 동작하고 실험을 시작하기 위한 sys_exe_flag_on() 시스템 콜을 호출하면 실험을 위한 부분이 수행한다. 우선 부하 분산 알고리즘의 빈도수 측정을 위한 데이터 영역과 태스크들의 정보를 저장하기 위한 데이터 영역에 대해 메모리를 할당하고 초기화한다. 이후 실험용 테스트 사용자 프로그램에서 fork()를 이용하여 태스크들을 생성하고 sys_process_check() 시스템 콜을 이용하여 생성한 태스크들의 task_struct 자료구조에 추가한 플래그 변수를 설정한다. 이후 이 플래그가 설정된 태스크의 생성 시간, 대기 시간, 종료 시간 등의 정보를 저장한다. 생성한 전체 태스크의 수행이 종료되면 저장한 데이터를 확인하기 위해 sys_read_cnt() 시스템 콜과 sys_read_data() 시스템 콜을 호출한다. 반환한 자료를 정리하여 부하 분산 알고리즘의 발생 빈도와 전체 태스크의 평균 대기 시간을 측정한다. [그림 4-1]은 task_struct 자료구조 안에 추가한 플래그 변수이다.

```
struct task_struct {
        ...
        /* 실험에서 측정할 태스크를
           구분하는 플래그 변수 추가*/
        int exp_check;
}
```

[그림 4-1] task_struct 자료구조 수정

[그림 4-2]는 실험에서 측정할 태스크의 정보를 저장하기 위해 정의한 자료구조이다. 프로세스 번호, 처리기 번호, 프로세스 생성 시간, 프로세스 수행 시작 시간, 프로세스 수행 종료 시간, 실행 큐 간의 이동 횟수에 대한 정보를 저장하며 이를 이용하여 태스크 대기시간을 계산한다. 이 자료구조를 통하여 해당 태스크의 대기 시간과 처리 시간, 이동 횟수를 측정한다.

```
struct exp_ps_data {
    pid_t pid;          /* 프로세서 ID */
    u32 cpu_id;         /* 프로세스 할당 CPU ID */
    u64 create_time; /* 프로세스 생성 시간 */
    u64 start_time;  /* 프로세스 수행 시작 시간 */
    u64 end_time;    /* 프로세스 수행 종료 시간 */
    u32 move_cnt;  /* 실행 큐 이동 횟수      */
}
```

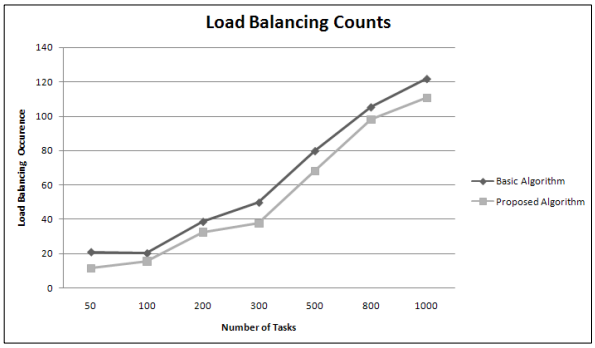
[그림 4-2] 실험에서 저장할 태스크 정보 구조체

4.1.3 구성 요소

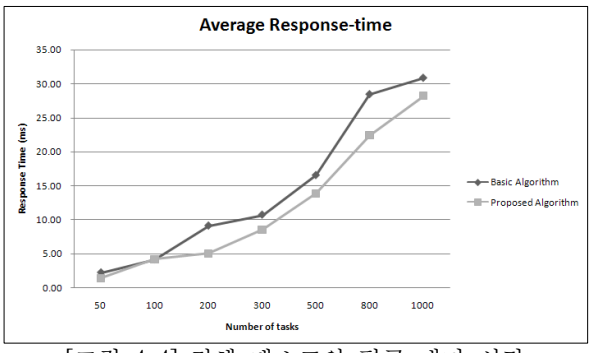
본 논문에서 제시한 알고리즘의 성능을 실험하기 위해 네 가지의 태스크 부류를 이용하여, 전체 태스크의 수, 실행 큐 간의 태스크 수 비율을 실험 요소로 사용한다. 우선 태스크 종류의 다양성을 위하여 각 처리하는 내용을 달리하여 구분 지었다. 첫 번째 태스크 종류는 처리기 중심의 태스크로써 주로 처리기의 연산을 사용하는 태스크이다. 두 번째 태스크 종류는 I/O 장치를 사용하는 태스크로써 처리기의 사용은 거의 없고 I/O 장치를 사용하도록 하였다. 세 번째 태스크 종류는 메모리 할당을 요청하는 태스크로써 커널에 힙 영역의 메모리를 요청하고 종료 시에 해제하는 작업을 하는 태스크이다. 네 번째 태스크 종류는 소켓을 이용하는 태스크로써 일반적인 서버 클라이언트 모델의 채팅 프로그램을 작성하여 사용하였다. 전체 태스크 수는 50, 100, 200, 300, 500, 800, 1000 단위로 사용하였고 네 가지 종류의 태스크를 임의로 비율로 분배하였다. 또한 실행 큐 간의 태스크 수 비교 비율은 10%부터 90%까지 10% 단위로 설정하였다.

4.2 실험 결과

[그림 4-3]과 [그림 4-4]는 기존의 부하 분산 알고리즘과 본 논문에서 제안한 부하 분산 알고리즘의 성능 비교 평가 그래프이다.



[그림 4-3] 부하 분산 알고리즘의 수행 횟수



[그림 4-4] 전체 태스크의 평균 대기 시간

5. 결 론

본 논문에서 제안한 알고리즘의 실험 결과를 보면 태스크 부하 분산 알고리즘의 수행 횟수와 전체 태스크의 평균 대기시간이 감소되었음을 알 수 있다. [표 5-1]은 전체 태스크 수에 따른 부하 비교 비율을 나타내고 있다. 전체 태스크의 수가 약 800개 이하일 경우 60% ~ 70% 비율에서 가장 높은 성능을 보이고 더 많은 태스크의 수가 있는 경우 30% 정도의 비율이 높은 성능을 가진다.

[표 5-1] 태스크 부하 비교 비율

전체 태스크 수	태스크 부하 비교 비율
0 - 200	60%
200 - 800	70%
800 - 1000	30%

결론적으로 태스크 부하 비교 비율을 적용함으로써 부하 분산 알고리즘의 수행 횟수가 감소되었고 태스크의 수행 대기 시간이 감소하였다. 이러한 결과는 알고리즘을 수행함으로써 발생하는 오버헤드가 태스크 처리에 있어 큰 영향을 미치고 있음을 알 수 있고, 태스크 비교 비율을 적용함으로써 오버헤드를 줄이는 효과를 얻었다.

[참고문헌]

[1] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis, Evaluating MapReduce for Multi-core and Multiprocessor Systems, Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture

[2] Songnian Zhou, A Trace-Driven Simulation Study of Dynamic Load Balancing, IEEE Transactions on software engineering VOL. 14, NO. 9, SEPTEMBER 1988.

[3] Mor Harchol-Balter, Allen B. Downey, Exploiting Process Lifetime Distributions for Dynamic Load Balancing, ACM Transactions on Computer Systems, Vol. 15, No. 3, August 1997, Pages 253-285.

[4] Maurice Herlihy, Beng-Hong Lim, Nir Shavit, Low Contention Load Balancing on Large-Scale Multiprocessors, SPAA '92 - 6/92/CA.

[5] Marc H. Willebeek-LeMair, Anthony P. Reeves, Strategies for Dynamic Load Balancing on Highly Parallel Computers, IEEE Transactions on parallel and distributed systems, VOL. 4, NO. 9, September 1993.

[6] Paul E.West, Yuval Peress, Sally A.McKee, Gary S. Tyson, Core Monitors: Monitoring Performance in Multicore Processors, May 18-20, 2009, Ischia, Italy, 2009 ACM.

[7] Multi-core, http://en.wikipedia.org/wiki/Multi_core/

[8] Eren Kursun, Glenn Reinman, Suleyman Sair, Anahita Shayesteh, Tim Sherwood, Low-Overhead Core Swapping for Thermal Management, Power-aware computer systems. International workshop No4, Portland OR , ETATS-UNIS (05/12/2004) 2005, vol. 3471, pp. 46-60.

[9] James Donald and Margaret Martonosi, Techniques for Multicore Thermal Management : Classification and New Exploration, Proceedings of the 33rd International Symposium on Computer Architecure (ISCA'06).

[10] Enric Musoll, A thermal-friendly load-balancing technique for multi-core processors, 9th International Symposium on Quality Electronic Design, IEEE 2008.

[11] Ravishankar Rao, Sarma Vrudhula, Chaitali Chakrabarti, Throughput of Multi-core Processors Under Thermal Constraints, ISLPED'07, August 27-29, 2007, Portland, Oregon, USA, ACM 2007.

[12] Tong Li, Dan Baumberger, David A. Koufaty, Scott Hahn, Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures, SC07 November 10-16, 2007, Reno, Nevada, USA ACM 2007.

[13] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, Sergey Blagodurov, A Comprehensive Scheduler for Asymmetric Multicore Systems, Aprial 13-16, 2010, Paris, France, 2010 ACM.

[14] Suresh Siddha, Multi-core and Linux Kernel, Intel Open Source Technology Center 2007.