리눅스의 쓰기 입출력 응답성 향상을 위한 동적 쓰기 캐시 할당 기법

김성진○ 박찬익
포항공과대학교
go4real@postech.ac.kr, cipark@postech.ac.kr

A dynamic allocation of Write Cache to Improve I/O Real-Time Responsiveness in Linux

Sungjin Kim○ Chanik Park
Pohang University of Science and Technology

요 약

대부분의 운영체제에서 쓰기 요청은 지연쓰기를 통하여 쓰기 성능을 향상시키고 있다. 운영체제의 메모리 관리자는 주기적, 더티 페이지 양이 특정 임계치를 초과 하였을 때 또는 명시적으로 비우기 동작이 요청되었을 때 관련 커널 스레드를 통하여 디스크에 더티 페이지를 비우는 작업을 한다. 이 과정에서, 기존의 비우기 관련 정책은 더티 페이지의 전체적인 분포에 근거하여 동작을 수행 한다. 그렇기 때문에 쓰기 요청을 발생시킨 프로세스의 중요도에 대한 고려가 되어 있지 않다. 일부 입출력 스케줄러의 경우는 이에 대한 고려가 있기는 하지만 비우기 정책 상에는 이를 구분하지 않기에 혼잡 발생시 실제적인 효과는 보기 힘든 상황이다. 이로 인해 높은 우선 순위의 태스크의 쓰기 작업이 낮은 우선 순위를 가지는 태스크의 쓰기 작업으로 인해 영향을 받는 현상이 발생하게 된다. 본 논문에서는 우선순위 그룹별 쓰기 캐시의 동적 할당방법을 통해 높은 우선 순위의 쓰기 요청이 낮은 우선순위 태스크의 쓰기에 방해를 받지 않고 안정적으로 이루어 지는 것을 보여주고 있다.

1. 서 론

쓰기 I/O 성능 향상을 위해 대부분의 운영체제는 지연쓰기를 하고 있다. 이는 주 저장장치와 다른 시스템 장치간의 속도 차이에서 발생하는 문제를 개선하여 쓰기성능을 개선할 수 있는 방법으로 여러 번 발생할 수 있는 쓰기 연산을 모아서 처리함으로써 디스크의 접근 횟수를 줄일 수 있게 한 방식이다[1]. 쓰기 과정에서 발생한 더티 페이지들은 가상 메모리 관리자가 디스크에 해당 페이지들을 쓰기 위한 커널 쓰레드를 수행시키기 전까지 쓰기 캐쉬 공간에 머무르게 된다. 리눅스에서는 가상 메모리 관리자가 커널 쓰레드 (pdflush)를 통해 주기적, 설정된 임계치를 초과하였을 경우, 또는 명시적으로 비우기 동작이 요청되었을 경우 더티 페이지를 디스크에 쓴다[1][2].

대부분의 운영체제에서 이와 같은 방법으로 쓰기를 비동기적으로 처리하지만, 읽기는 동기적인 방식으로 처리가 된다. 이런 이유로 기존 연구들에서 쓰기 보다는 읽기 성능을 향상 시킬 수 있는 방법이 많이 진행되어 왔다[3][4]. 하지만 프로세서와 디스크 간의 성능 차이가 커짐에 따라 디스크의 쓰기 성능이 시스템 성능에 영향을 미치는 상황이 발생하게 된다. 예를 들면, 시스템이 더티 페이지를 디스크에 기록할 수 있는 속도보다 빠르게 더 페이지가 생성될 경우 문제가 될 수 있다. 이런 문제점을 개선하기 위해 쓰기와 관련된 연구가 진행되었다[5][6]. 기존의 블록 디바이스의 쓰기 관련 연구들은 쓰기와 관련된 페이지가 실제 디스크에 쓰이는 시점을 조절하거나 지연쓰기 과정에서 블록 재배치를 하여 성능을 개선하고 있다. 하지만 쓰기 요청을 발생시킨 프로세스의 중요도에 대한 고려가 되지 않기 때문에 높은 우선 순위의 태스크의 쓰기 요청이 낮은 프로세스가 발생시킨 쓰기 요청에 의해 영향을 받는 상황이 발생하게 된다.

그림1의 경우는 다른 경쟁 쓰기 프로세스가 없는 경우의 응답시간이고, 그림2의 경우는 다른 경쟁 쓰기 프로세스가 존재하는 상황에서 높은 우선순위를 가지는 태스크의 쓰기 응답시간 지연을 보여주고 있다. 이는 기존의 쓰기 정책이 프로세스의 우선순위에 대한 고려가 없기 때문이다. 본 논문에서는 높은 우선 순위 태스크의 쓰기 응답시간을 보장하기 위해 더티 페이지 및 태스크의 분포와 태스크 그룹별 가중치를 적용하는 방식을 통해 쓰기 캐시의 동적인 할당 기법을 구현하고자 한다.

"본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음" (NIPA-2009-C1090-0902-0045)

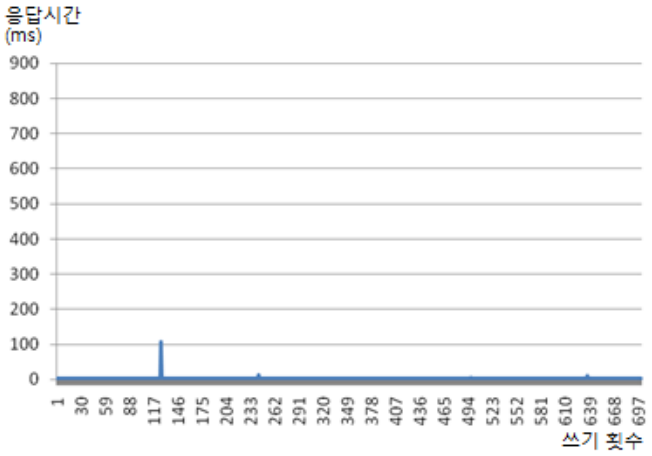

그림 1 vanilla 커널의 방해가 없는 상황에서 쓰기 응답 시간. 512KB 데이터를 20ms주기로 700번 쓰기수행.

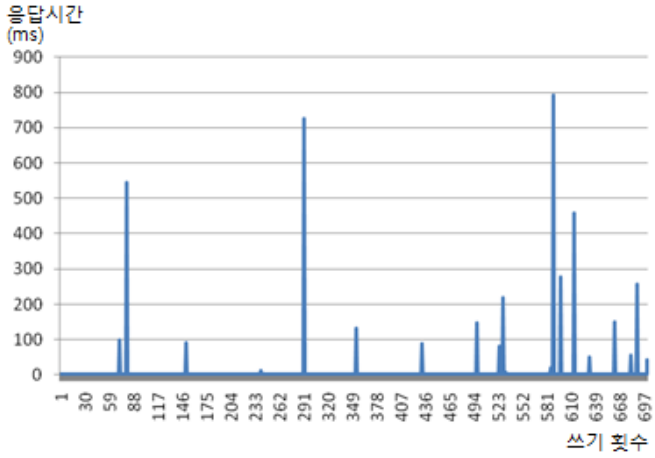

그림 2 쓰기가 방해 받는 상황에서 vanilla 커널의 쓰기 응답시간. 다른 우선순위의 두 프로세스가 동시에 512KB 데이터를 20ms 주기로 700번 쓰기 수행.

본 논문은 다음과 같은 순서로 구성되어 있다. 2장에서는 배경지식과 관련 연구들에 대해 알아보고, 3장에서는 제안한 기법에 대한 내용과 동적 쓰기 캐시 할당 알고리즘의 구조 및 디자인에 대해 알아본다. 그리고 4장에서는 실험을 통하여 제안한 기법을 평가하고 마지막 5장에서 결론을 기술한다.

2. 배경 지식 및 관련 연구

이 장에서는 리눅스에서 기존의 쓰기 시점, 쓰기 정책 및 관련 연구에 대해 설명한다.

2.1. 리눅스의 더티 페이지 실제 쓰기 시점

리눅스는 지연쓰기를 통해 쓰기를 효율적으로 수행하게 된다. 실제 디스크에 비우는 동작은 다음 네 가지 경우에 수행된다. 첫째로, 페이지가 너무 오랫동안 더티 상태로 유지되는 것을 방지하기 위해 디스크에 쓰기를 담당하는 커널 쓰레드에 의해 주기적으로 비워지게 된다. 둘째로 페이지 캐시의 여유가 없는 상태에서 새로운 페이지의 할당을 요청할 경우 새로운 페이지를 할당하기 위해서 커널 쓰레드에 의해 디스크에 비우는 작업이 수행된다. 셋째로 sync(), fsync() 같은 명시적인 시스템 콜이 수행이 될 때 페이지 캐시내의 더티 페이지들을 디스크에 비우게 된다. 마지막으로 write() 시스템 콜 등을 통해 쓰기 요청이 들어올 때 더티 페이지의 비율이 특정 임계치 이상일 경우 이를 조절하기 위해 비우기 작업을 수행한다.

2.2. 리눅스의 쓰기 캐시 비우기 정책

리눅스의 지연 쓰기는 이를 담당하는 커널 스레드 pdflush 에 의해 이루어진다. pdflush 의 동작은 다음 네 개의 제어 변수들을 통해 조절이 가능하다.

- vm_dirty_ratio : 더티 페이지로 사용될 수 있는 최대 허용 비율을 정의 한다. 더티 페이지의 비율이 이 임계치를 넘어서게 되면 해당 write요청은 지연되고, 기존 더티 페이지를 일정 기간 동안 디스크에 비우는 동작을 하게 된다.
- dirty_background_ratio : pdflush가 백그라운드로 동작을 하며 더티 페이지를 비우기 시작하는 시점의 비율을 정의 한다.
- dirty_writeback_interval : pdflush가 동작중인 상태에서 각 수행간의 주기를 정의한다.
- dirty_expire_interval : 더티 페이지로 존재할 수 있는 최대 시간을 정의 한다.

pdflush 는 메모리 내의 더티 페이지의 비율이 dirty_background_ratio 를 넘어가게 되면 더티 페이지를 디스크에 비우는 작업을 시작하여 dirty_writeback_interval 에 정의된 주기로 플러싱 작업을 수행한다. 더티 페이지의 비율이 vm_dirty_ratio 에 도달하게 되면 더 이상 더티 페이지가 생성이 안되고, 프로세스에서 강제적으로 더티 페이지를 비우는 작업을 수행하게 된다. 이 시점에 쓰기 응답 시간은 매우 큰 폭으로 증가를 하게 된다. 그래서 대부분의 쓰기 관련 연구들은 이 시점의 발생을 줄이고자 하고 있다[5][8].

이런 기존 리눅스의 플러싱 정책은 시스템 내의 총 더티 페이지의 분포를 기준으로 동작을 하게 된다. 그렇기 때문에 한정된 쓰기 캐시를 여러 프로세스가 공유하는 상황에 있어서 특정 프로세스에 의해 다른 프로세스의 쓰기가 방해 받는 상황이 발생하게 된다. 즉 낮은 우선 순위의 프로세스가 발생한 쓰기 요청으로 인해 높은 우선 순위 프로세스의 쓰기 요청의 지연이

발생하게 된다. 이는 리눅스의 쓰기 실시간 응답 성능을 저하시키는 원인이 된다.

2.3. 입출력 스케줄러

각 블록 장치는 파일 시스템이 요청한 I/O 를 처리하기 위해서, 각 요청을 저장하고 있는 입출력 요청 큐를 유지 하고 있다. 이러한 상황에서 입출력 스케줄러는 큐에 존재하는 입출력 요청들을 디스크의 접근 위치에 따라서 새롭게 정렬 하거나, 혹은 인접한 위치를 접근하는 입출력 요청들을 합치기도 한다. 이러한 입출력 스케줄러 메커니즘에 의해서 실제 블록 장치는 디스크의 접근 횟수와 시간을 줄일 수 있다. 또한, 입출력 스케줄러는 총 네 가지의 스케줄링 방법을 통해서 입출력 요청을 처리할 수 있다.

- No-Op : 가장 간단한 입출력 스케줄링 알고리즘이며, 정렬된 큐가 없고 새로운 요청은 항상 큐의 머리 또는 꼬리에 붙는다.
- Deadline : 표준 Elevator 알고리즘으로써 요청의 첫 섹터 번호를 기준으로 정렬 또는 병합을 수행한다. 또한 요청에 대해서 starvation 을 방지하기 위해 deadline 을 두고 있다.
- Anticipatory : 기본 알고리즘은 deadline 과 같지만, 더 나은 sequentiality 를 구성하기 위해 일정 시간 동안 요청을 기다리는 방식이다.
- CFQ : Multi-process 환경에서 주로 쓰이는 알고리즘이며, round-robin 방법을 사용하여 모든 경쟁관계의 프로세스들의 입출력 요청을 처리할 동일한 기회를 준다.

이중 CFQ 의 경우는 I/O 요청에 대해 real-time, best-effort, idle 로 구분하여 별도의 큐와 스케줄링 방법을 구현하였지만, 시스템내의 혼잡 제어 알고리즘은 요청 큐의 구분 없이 해당 큐의 혼잡도만을 확인하기 때문에 best-effort 프로세스가 많은 양이 I/O 를 발생시킬 경우 real-time 태스크의 I/O 응답 시간을 보장해 줄 수 없다[7].

3. 쓰기 캐시의 동적 할당 기법 설계 및 구현

그림 3 는 본 논문에서 제안한 쓰기 캐시 동적 할당 기법의 전체 구조(이후 WCALL 이라 정의)를 보여준다. WCALL 은 Monitoring Module 로부터 페이지 캐시와 태스크 정보를 수집하여 Write Cache Allocation Module 에 전달하게 된다. Write Cache Allocation Module 은 모니터링된 정보에 기반하여 각 프로세스 그룹별로 가중치를 적용하여 쓰기 캐시를 동적으로 할당한다. Write Back Control Module 은 할당된 쓰기 캐시 정보를 토대로 디스크에 더티 페이지를 쓰는 타이밍을 조절한다.

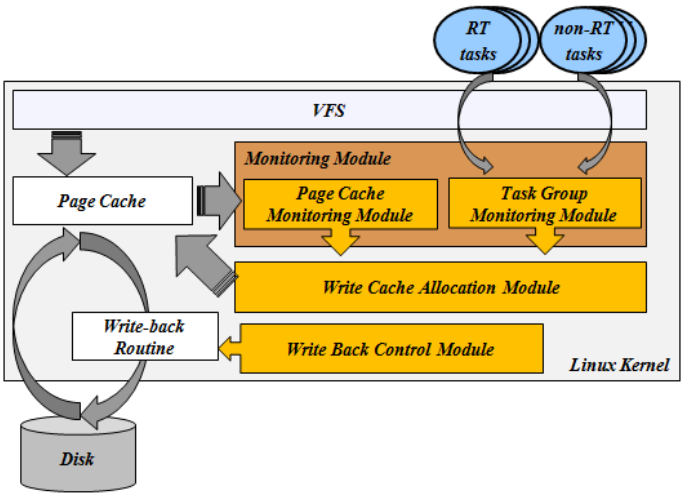

그림 3 쓰기 캐시 동적 할당 기법 전체 구조

3.1 모니터링 모듈

모니터링 모듈은 페이지 캐시 동적 할당에 필요한 정보를 수집하기 위해 페이지 캐시와 동작중인 태스크의 정보를 수집하게 된다. 각각의 정보를 수집하기 위해 태스크 그룹 모니터링 모듈과 페이지 캐시 모니터링 모듈로 구성된다.

3.1.1. 태스크 그룹 모니터링 모듈

페이지 캐시를 동적으로 할당하는데 있어서 모든 프로세스의 개별 우선 순위를 고려하여 쓰기 캐시를 분배하는 것은 한계를 가지게 된다. 본 논문에서는 실시간 태스크의 쓰기 응답시간 보장을 목표로 하고 있기 때문에 다음과 같이 세 가지의 그룹으로 태스크들을 구분하고 있다.

- RT-HIGH : 우선순위가 50 보다 큰 실시간 태스크들
- RT-LOW : 우선순위가 0 이상이고 50 이하인 실시간 태스크들
- NON-RT : 실시간 태스크가 아닌 태스크들

쓰기 요청이 들어오면 이를 발생시킨 태스크의 우선순위를 확인하여 우선순위 그룹별로 참조 정보를 유지한다.

3.1.2. 페이지 캐시 모니터링 모듈

더티 페이지가 생성되는 시점에서 해당 쓰기 요청을 발생시킨 태스크의 우선 순위를 확인하여, 해당 태스크 그룹의 더티 페이지 정보를 갱신하게 된다.

3.2. 쓰기 캐시 할당 모듈

```
WEIGHT_H_RT 4 // 높은 우선순위 RT 태스크 그룹의 가중치
WEIGTH_L_RT 2  // 낮은 우선순위 RT 태스크 그룹의 가중치
WEIGHT_NORMAL 1 // 일반 태스크 그룹의 가중치

//n_h_rt        높은 우선순위 RT 태스크 그룹의 태스크 개수
//n_dp_h_rt     높은 우선순위 RT 태스크 그룹의 더티 페이지 개수
//n_l_rt        낮은 우선순위 RT 태스크 그룹의 태스크 개수
//n_dp_l_rt     낮은 우선순위 RT 태스크 그룹의 더티 페이지 개수
//n_normal      일반 태스크 그룹의 태스크 개수
//n_dp_normal   일반 태스크 그룹의 더티 페이지 개수

1   total_rt_h     = n_h_rt * WEIGHT_H_RT * n_dp_h_rt
2   total_rt_l     = n_l_rt * WEIGTH_L_RT * n_dp_l_rt
3   total_normal  = n_normal * WEIGHT_NORMAL * n_dp_normal

4   total_base  = total_rt_h + total_rt_l + total_non_rt;

5   if(현재 프로세스가 높은 우선 순위의 RT 태스크 일 경우){
6      background_ratio  = background_ratio * total_rt_h /total_base;
7      dirty_ratio = dirty_ratio * total_rt_h /total_base;
8   }

9   else if(현재 프로세스가 낮은 우선 순위의 RT 태스크 일 경우){
10     background_ratio  = background_ratio * total_rt_l /total_base;
11     dirty_ratio = dirty_ratio * total_rt_l /total_base;
12  }
13  else if(현재 프로세스가 Normal 태스크일 경우){
14     background_ratio  = background_ratio * total_normal /total_base;
15     dirty_ratio = dirty_ratio * total_normal /total_base;
16  }
```

알고리즘 1 쓰기 캐시 동적 할당 알고리즘

알고리즘 1 은 쓰기 캐시 할당 모듈이 모니터링 모듈을 통해 받은 정보를 기반하여 쓰기 캐시를 어떻게 그룹별로 할당하고 있는 지를 보여준다.

모니터링 모듈로부터 받은 더티 페이지 정보와 태스크 그룹별 구성 정보를 확인하여 쓰기 캐시를 동적으로 할당한다. 또한 각 프로세스 그룹별로는 가중치가 다르게 적용하여 높은 우선 순위를 가진 프로세스 그룹이 더 안정적인 쓰기 공간 확보를 할 수 있도록 하고 있다.

3.3. 쓰기 제어 모듈

알고리즘 1 을 통해 구분된 쓰기 캐시 공간에 기반하여, 2.1 절에서 언급했던 리눅스의 더티 페이지 실제 쓰기 시점의 네 가지 경우 중에 마지막 경우를 컨트롤하게 된다.

각 우선순위 그룹별로 할당된 공간을 초과하여 사용하고 있을 경우에는 해당 그룹의 속하는 태스크의 쓰기 요청은 일정기간 블록된 상태를 가지며 수행된다. 여기서 사용한 블록 기간은 기존 리눅스에서 사용하고 있는 혼잡 발생시 블록 주기(HZ/10)를 그대로 사용하였다. 예를 들어 높은 우선순위와 낮은 우선순위 그룹의 태스크가 동시에 쓰기 요청을 하고 있는 경우에 낮은 프로세스의 그룹에 속한 태스크들이 자신들에게 할당된 페이지 이상을 소모하게 되면 일정 시간의 블록 상태가 섞여서 나타내게 되기 때문에 높은 우선순위 그룹의 태스크들의 쓰기 응답시간의 방해를 최소화 할 수 있다. 만약에 한 우선순위 그룹에서만 쓰기 요청이 발생하였을 때에는 거의 대부분의 쓰기 공간을 해당 그룹이 차지하고 있지만, 다른 그룹의 쓰기 요청에 들어옴에 따라 가중치의 적용에 따라 빠르게 다른 그룹에 공간을 할애해 주게 된다.

각 그룹별 전체 더티 페이지 합계가 전체 쓰기 공간에 할당된 최대치(리눅스 2.6.26 커널의 경우, 더티 페이지로 사용 가능한 메모리 공간 중 10%)를 넘게 될 경우에는 기존과 같은 방식으로 디스크에 더티 페이지를 강제적으로 비우는 동작이 수행된다. 하지만 개별적인 구역에 따른 분포에 따라서 강제 쓰기는 발생하지 않는다. 즉, 한 영역에서 자신에게 할당된 공간을 다 소모했을 때 강제 비우는 동작이 발생하는 것이라, 해당 영역에 쓰기 요청은 블록 상태로 대기상태가 되게 되고 시스템 전체의 더티 페이지 비율에 근거하여 강제 비우는 동작을 수행하게 된다. 이는 잦은 강제 비우기로 인해 불필요한 응답시간 증가를 줄이고 지연쓰기 효과의 감소를 줄이기 위해서이다.

또한 비우는 동작에 있어서는 프로세스 그룹간의 구별이 무의미 하기 때문에 이에 대한 별도의 처리는 하지 않았다. 그 이유는 실제적 메모리 상에서의 각 그룹별 영역이 따로 나누어져 있는 것이 아니기 때문에, 전체적으로 여유 페이지가 존재한다면 어느 영역이든지 필요에 따라 할당이 가능하기 때문이다. 즉 가상메모리 관리자의 관점에서도 어떤 우선순위 그룹의 더티 페이지를 비우느냐 보다는 얼마나 빨리 더티 페이지를 감소시키는가 하는 것이 중요하기 때문이다.

4. 성능 평가 및 분석

성능 평가를 위해서, 리눅스에 WCALL 을 구현하였으며, 비교를 위해 vanilla 커널과 WCALL 이 적용된 커널에서 동일한 실험을 수행하였다.

성능 평가를 위해 사용된 장비와 리눅스 커널의 정보는 다음과 같다.

- Intel Pentium IV 2.66 GHz
- 1GB SDRAM
- Linux Kernel 2.6.26

실험은 높은 우선 순위 태스크의 쓰기 응답시간이 낮은 우선 순위 태스크의 쓰기 동작으로 인해 지연을 받는 상황과 비실시간 태스크만이 존재하는 상황을 구성하였다.

4.1. 쓰기가 경쟁상태에 있는 경우

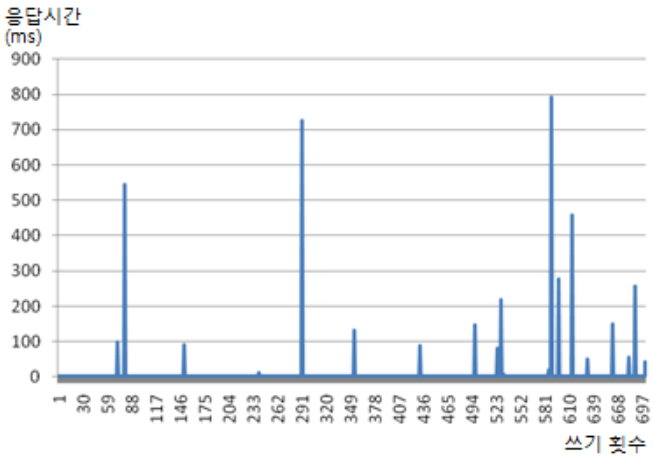

그림 4 vanilla 커널의 응답시간

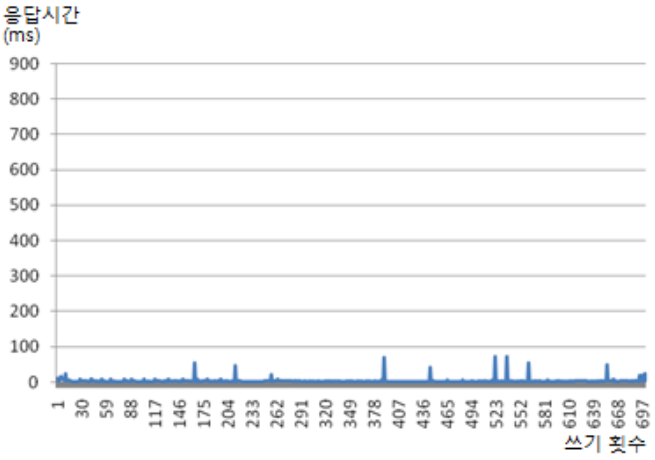

그림 5 쓰기 캐시 동적 할당 기법을 적용한 커널의 응답시간

실험에 사용한 워크로드는 우선 순위가 55 인 실시간 프로세스가 512KB 의 데이터를 20ms 주기로 700 번 쓰기 동작을 수행하고 있고, 동시에 우선 순위가 45 인 실시간 프로세스 두 개가 같은 디스크에 역시 512KB 의 데이터를 20ms 주기로 700 번 수행하고 있다.

그림 4 에서 보듯이 낮은 우선 순위 태스크들의 쓰기 작업으로 인해 높은 태스크의 응답 시간 지연 회수가 그림 1 에 비해 현저하게 증가한 것을 알 수가 있다.

또한 응답 시간이 200ms 를 넘어서는 구간이 많이 존재하는 이유는, 2.2 절에서 설명했던 것처럼 기존 리눅스의 쓰기 정책이 프로세스에 대한 구별이 이루어지지 않고 글로벌한 관점에서 관리가 되기 때문이다. 그림 5 의 응답 시간에서 확인 할 수 있듯이 본 논문의 접근 방법에서는 쓰기 응답시간이 많이 향상된 것을 알 수 가 있다. 이는 프로세스 그룹별로 모니터링 되어진 정보와 그룹별 가중치를 적용한 WCALL 방식이 안정적으로 쓰기 캐시를 관리해 주고 있기 때문이다.

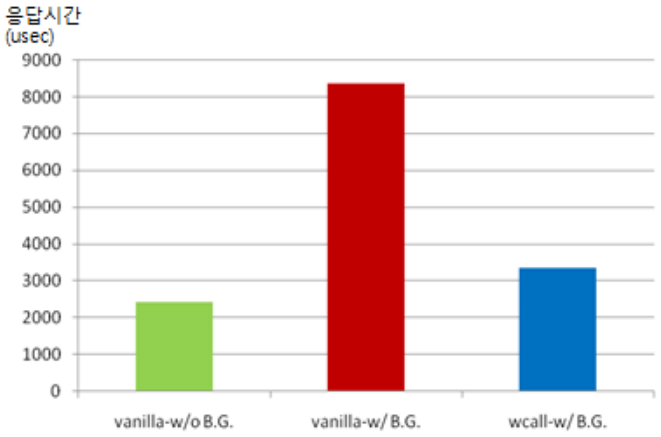

그림 6 평균 응답시간.

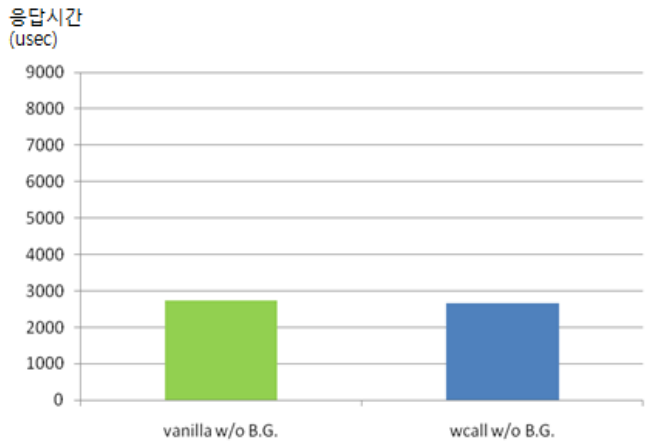

그림 7 비실시간 태스크만이 쓰기 요청을 할 경우 평균 응답시간.

그림 6 에서 볼 수 있듯이 쓰기 요청의 방해가 있을 때 평균 응답시간이 현저하게 증가 하는 현상을 확인할 수 있고(vanilla-w/o background 와 vanilla-w/ backgrounds 를 통해 확인), 본 논문이 제안한 방식으로 인해 큰 폭의 향상을 가져온 것을 알 수가 있다(vanilla-w/ backgrounds 와 walloc-w/ backgrounds 를 통해 확인).

4.2. 쓰기 경쟁이 없는 경우

실험에 사용한 워크로드는 nice 4 인 비실시간 태스크 하나가 512KB 의 데이터를 20ms 주기로 1000 번 쓰기 동작을 수행하고 있다.

그림 7 의 결과를 통해 경쟁이 없는 경우에 별도의 큰 오버헤드가 존재하여 응답시간이 지연되는 현상은 없는 것을 볼 수 있다. 오히려 작은 폭으로 평균 응답시간이 개선되는 것을 볼 수가 있다. 이는 WCALL 의 쓰기

제어모듈에서 조금 더 빈번하게 쓰기 요청을 진행하여서 이다.

5. 결론

본 논문에서는 기존 리눅스의 정적인 쓰기 캐시 관리 정책의 문제점을 분석 및 제시하였다. 그리고 문제점을 해결하기 위해, 더티 페이지 및 태스크의 분포와 태스크 그룹별 가중치를 적용하는 방식을 통해 쓰기 캐시의 동적인 할당 기법을 제안하였다. 제안한 기법이 문제가 되는 상황에서의 쓰기 I/O 응답시간을 향상 시키는 것을 보였다.

참고 문헌

[1] Bovet and Cesati, “Understanding Linux Kernel 3rd Edition”, O’REILLY, 2006.

[2] Norm Murray and Neil Horman, “Understanding Virtual Memory In Red Hat Enterprise Linux 4”, redhat.com, 2005.

[3] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In File and Storage Technologies Conference, 2003.

[4] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In ACM SIGMETRICS, 2002.

[5] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey, “AWOL: An Adaptive Write Optimization Layer”, 6th USENIX Conference on File and Storage Technologies(FAST), 2008.

[6] Binny S. Gill and Dharmendra S. Modha, WOW : Wise Ordering Writes Combining spatial and Temporal Locality in Non-Volatile Caches, 4th USENIX Conference on File and Storage Technologies(FAST), 2005.

[7] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First class support for interactivity in commodity operating systems. In Proc. of the OSDI, 2008.

[8] Binny S. Gill, Michael Ko, Biplob Debnath, and Wendy Belluomini. STOW: A Spatially and Temporally Optimized Write Caching Algorithm. USENIX Annual Technical Conference, 2009.