http://dx.doi.org/10.5392/JKCA.2016.16.05.251

고속 저장 장치를 위한 입출력 스택 최적화

Optimizing I/O Stack for Fast Storage Devices

한혁
동덕여자대학교 컴퓨터학과

Hyuck Han(hhyuck96@dongduk.ac.kr)

요약

최근 클라우드 컴퓨팅, 사회 관계망 서비스 등의 분야에서 고속 저장 장치에 대한 수요가 크게 증가하고 있다. 성능이 우수한 고속 저장 장치가 개발되고 있지만 현재 리눅스 운영체제의 입출력 스택은 하드 디스크 드라이브를 고려해서 설계되었기 때문에 고속 저장 장치를 충분히 활용하고 있지 못하다. 이 논문에서는 고속 저장 장치의 입출력 대역폭과 입출력 지연시간을 최대로 활용할 수 있는 최적화된 입출력 스택을 제안한다. 이를 위해 기존 리눅스의 블록 입출력 계층을 새로운 인터페이스를 가지는 입출력 계층으로 대체하고 최적화한다. 제안된 입출력 계층은 기존의 하드 디스크 드라이브를 고려한 블록 계층을 우회하고 디바이스 드라이버를 최적화하여 고속 저장 장치의 성능을 최대한 이용할 수 있게 해준다. 또한, 리눅스의 ext2/ext4 파일 시스템을 제안된 입출력 계층 위에서 동작할 수 있도록 최적화하였고, 벤치마크 실험 결과를 통해서 제안하는 입출력 스택은 기존 리눅스 입출력 스택과 비교하여 1.7배 정도의 성능 향상이 있음을 확인할 수 있었다.

■ 중심어 : | 고속 저장 장치 | 입출력 계층 | 리눅스 운영체제 |

Abstract

Recently, the demand for fast storage devices is rapidly increasing in cloud platforms, social network services, etc. Despite the development of fast storage devices, the traditional Linux I/O stack is not able to exploit the full extent of the performance improvement since it has been optimized for disk-based storage devices. In this paper, we propose an optimized I/O stack which can fully utilize the I/O bandwidth and latency of fast storage devices. To this end, we design a new I/O interface to replace the current block I/O interface and optimize our I/O interface. Our optimized I/O interface bypasses operations/layers in block I/O subsystems of the current Linux I/O stack to fully exploit fast storage devices. We also optimize the Linux file systems such as ext2 and ext4 to run on our I/O interface. We evaluate our I/O stack with multiple benchmarks and the experimental results show that our I/O stack achieves 1.7 times better throughput compared to traditional Linux I/O stack.

■ keyword : | Fast Storage Device | I/O Stack | Linux Operating System |

I. 서 론

메인 메모리와 저장 장치의 커다란 성능 격차는 컴퓨터 시스템의 전체적인 성능을 향상시키는 것을 제한하는 주된 요소였다. 최근 비휘발성 메모리와 고속 저장 장치 기술의 발달은 메인 메모리와 저장 장치 간 성능

* 이 논문은 2015년도 동덕여자대학교 학술연구비 지원에 의하여 수행된 것임.
접수일자 : 2016년 01월 05일
수정일자 : 2016년 03월 10일
심사완료일 : 2016년 03월 21일
교신저자 : 한혁, e-mail : hhyuck96@dongduk.ac.kr

격차를 줄여주고 있다. 이러한 비휘발성 메모리 및 고속 저장 장치들은 슈퍼 컴퓨팅, 클라우드, 사회 관계망 서비스 등에서 폭넓게 사용되고 있다. 인텔이 2013년 Flash Memory Summit에서 발표한 자료에 따르면[1] PCIe 인터페이스 기반의 플래시 SSD는 읽기 지연 시간이 80us 정도이다. 그리고 PRAM (Phase-change Memory) 혹은 MRAM (Magnetoresistive Random Access Memory)과 같은 고속 비휘발성 메모리 기반 저장 장치의 경우 읽기 지연 시간이 10us 정도이다.

하지만 저장 장치가 고속으로 자료를 입출력할 수 있음에도 리눅스 운영체제의 입출력 스택은 하드 디스크 드라이브를 고려되어 설계되었기 때문에 고속 저장 장치의 이점을 충분히 활용하지 못 하고 있다[2]. [그림 1]과 같이 기존의 입출력 스택에서는 데이터에 접근하기 위해 입출력 요청은 가상 파일 시스템(VFS, Virtual File System), ext2/ext4와 같은 실제 파일 시스템, 블록 계층, 디바이스 드라이버 계층을 모두 통과하여야 한다. 저장 장치가 고속으로 입출력 요청을 처리하는 상황에서는 입출력 스택의 여러 계층에서 발생하는 소프트웨어 오버헤드를 무시할 수 없게 된다. 따라서 이러한 소프트웨어 오버헤드를 줄여서 고속 저장 장치의 성능을 최대로 활용할 수 있는 입출력 스택이 필요하다.

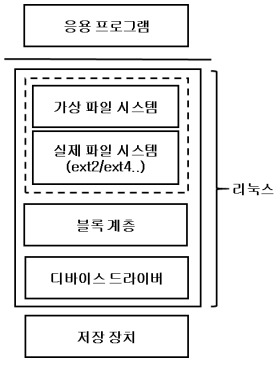

그림 1. 기존의 리눅스 입출력 스택

최근에 PCIe 인터페이스 기반 고속 저장 장치를 위한 소프트웨어/하드웨어 최적화에 관한 많은 연구들이 진행되었다[3-8]. 이러한 연구들은 운영 체제에서 발생하는 오버헤드를 회피하기 위해 응용 프로그램 수준에서 직접 고속 저장 장치에 접근하는 방식을 제안하였거나[3][4] 입출력 계층의 일부만을 최적화하였다[5-7]. 또한 메모리 버스에 연결된 고속의 비휘발성 메모리를 (SCM, Storage Class Memory) 위한 최적화에 관한 연구도 진행되었다[9][10]. 이러한 연구들은 메모리에 저장된 데이터에 대한 시스템 실패 후에 일관된 변경, 메모리 누수, 데이터 변형을 위한 추가적인 하드웨어가 필요하거나 메모리 관리 기법이 필요하다.

본 논문에서는 고속 저장 장치를 위한 리눅스 운영체제의 입출력 스택을 제안하고자 하며, 가상 파일 시스템, 실제 파일 시스템, 블록 계층, 디바이스 드라이버 계층 전반을 최적화한다. 제안하고자 하는 입출력 스택은 [그림 2]와 같이 최적화된 디바이스 드라이버, 실제 파일 시스템, 가상 파일 시스템의 세 계층으로 구성되어 있다. 이러한 입출력 스택의 장점은 다음과 같다. 첫째, 다른 연구에서 제안한 응용 수준의 파일 시스템과 비교했을 때 파일 입출력에 필요한 권한 검사 등을 위한 저장 장치의 하드웨어 수준 최적화를 필요로 하지 않는다. 둘째, 응용 프로그램이 응용 수준의 파일 시스템이 제공하지 못 하는 저널링과 같은 기존 파일 시스템의 기능을 그대로 사용할 수 있다. 셋째, 메모리 버스에 연결된 SCM을 위한 파일 시스템과 달리 메모리 누수, 데이터 변형을 위한 하드웨어 변경을 요구하지 않는다.

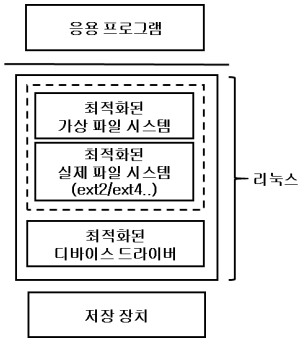

그림 2. 최적화된 리눅스 입출력 스택

본 논문의 최적화된 디바이스 드라이버는 기존의 블록 계층을 완전히 우회하며 무복사 입출력 (zero-copy I/O), 큐 교환 (queue swapping)과 같은 최적화 기법이

적용되었다. 그리고 파일 시스템은 최적화된 디바이스 드라이버를 이용하도록 최적화되었다. 실험 평가를 위해 ext2와 ext4를 제안된 디바이스 드라이버 상에서 동작하도록 구현하였다. 벤치마크 프로그램들을 이용하여 평가하였으며 제안된 입출력 스택이 기존 입출력 스택 대비 평균 1.7배 정도 성능이 우수한 것을 확인하였다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구를 소개하고 3장에서는 최적화 기법에 대해서 설명한다. 4장에서는 성능 평가를 설명하고 5장에서는 이 논문의 결론을 제시한다.

II. 관련 연구

최근에 고속 저장 장치의 성능을 최대로 활용하는 것에 대한 많은 연구가 수행되었다. Moneta[3] 연구에서는 모사된 PCM과 PCIe 인터페이스를 기반으로 하는 프로토타입 저장 장치를 제안하였다. Moneta-D[4] 연구에서는 Moneta 연구에서 개발된 저장 장치 환경에서 소프트웨어 오버헤드를 줄이는 방법들이 제안되었다. 운영체제의 오버헤드를 완전히 줄이기 위해 응용 수준의 디바이스 드라이버가 직접 저장 장치에 접근한다. 그러나 이러한 연구들은 본 연구와는 달리 하드웨어 수준의 최적화가 필요하다. 즉, 운영체제 수준에서 수행했던 권한/보안 검사 기능 등이 저장 장치 수준에서도 구현이 되어야 하는 등의 하드웨어 수준에서 변경이 필요하다. 이 밖에도 J. Coburn의 연구에서는 DRAM 기반 SSD와 데이터베이스 시스템 소프트웨어를 상호 최적화하여 성능을 개선시키고자 했다[5].

고속 저장 장치의 성능을 하드웨어 변경이 없이 소프트웨어 최적화로 개선하는 연구도 수행되었다. [6][7]의 연구에서는 고속 저장 장치 환경에서 폴링 (polling), 시간적인 병합(temporal merge)과 같은 블록 계층을 최적화하는 방법들을 제안하였다. [8]의 연구에서는 고속 저장 장치 환경에서 오히려 성능을 떨어뜨리는 SCSI, ATA 계층을 우회하는 것을 제안하였다. 이러한 연구들은 본 연구와는 달리 입출력 스택에서 블록 계층의 최적화만 다루고 있다.

메모리 버스에 연결된 고속의 비휘발성 메모리는 (SCM) PCIe/SATA 기반 저장 장치와 달리 인터페이스 오버헤드가 거의 없으며 CPU에서 직접 접근이 가능하다. SCM이 탑재된 컴퓨터 시스템에서는 SCM에 접근할 때 블록 계층, 디바이스 드라이버 등을 거치지 않기 때문에 성능 향상에 용이하다. SCMFS (File System for Storage Class Memory)[9] 및 BPFS (File System for Byte-Addressable NVM)[10] 연구는 메모리 버스에 연결된 비휘발성 메모리 환경에서의 파일 시스템 최적화에 관한 연구이다. 이러한 연구들은 본 연구와는 달리 파일 시스템 최적화에 초점이 맞추어져있다.

이 밖에도 [11]과 같은 연구에서는 고속의 네트워크 장치인 InfiniBand를 이용하여 원격 메모리를 블록 장치로 활용하여 메모리를 확장하였다. 이 연구에서는 운영체제의 가상 메모리 관리 계층을 최적화하였다. 본 논문은 기존의 연구와는 달리 PCIe 기반의 고속 저장 장치의 성능을 최대로 활용하기 위한 운영체제의 특정 계층이 아닌 입출력 스택을 전반적으로 최적화한다.

III. 리눅스 입출력 스택 최적화

이번 장에서는 고속 저장 장치를 탑재한 시스템에서 리눅스 입출력 스택을 분석하고 분석된 결과를 바탕으로 입출력 스택 최적화 기법을 제안한다.

3.1 리눅스 입출력 스택 분석

기존 리눅스 입출력 스택[1]을 분석하기 위해서 FIO 벤치마크를 이용하여 실험을 진행하였다. 실험 환경은 4절의 환경과 동일하다. [표 1]은 하나의 쓰레드를 가지는 하나의 프로세스가 ext2 파일 시스템의 4KB 데이터를 저장 장치로부터 읽어오는 데 걸리는 시간을 분석한 결과를 보여준다. FIO 프로세스 수준에서의 읽기 지연 시간[2]은 28.8us이다. 이 중에서 저장 장치의 읽기 지연

1. 실험에 사용된 디바이스 드라이버에는 기존 연구에서 제안된 폴링 및 시간적 병합 기법 등이 구현되었다.
2. 페이지 캐시, 저널링 효과를 배제하기 위하여 Direct I/O 모드에서 측정하였다.

시간은 5us이며 리눅스 입출력 스택의 지연 시간은 23.8us이다. 리눅스 운영체제의 입출력 지연 시간이 전체 입출력 지연 시간의 82.6%를 차지하고 있음을 알 수 있다. 또한 입출력 스택의 모든 계층에서 전반적으로 무시할 수 없는 오버헤드가 있음을 알 수가 있다.

표 1. 리눅스 입출력 스택 지연 시간 프로파일 결과(VFS: 가상 파일 시스템, FS: 실제 파일 시스템, BLK: 블록 계층, DEV: 디바이스 드라이버)

입출력 스택		시간(us)	비율(%)
SW 계층	VFS	5.8	20.1
	FS	8	27.7
	BLK	5	17.4
	DEV	5	17.4
총 S/W 지연시간		23.8	82.6
저장 장치 지연시간		5	17.4
전체 지연시간		28.8	100

3.2 디바이스 드라이버 수준 최적화

본 연구에서 제안하는 디바이스 드라이버는 기존의 블록 인터페이스가 아닌 파일 시스템에서 사용하는 정보들 혹은 자료구조들을 파라미터로 사용할 수 있다. 또한 기존 연구를 통해 잘 알려져 있는 폴링(polling)과 시간적 병합 (temporal merge) 기법을 기본적으로 사용하고 있다. 입출력 성능을 개선하기 위해 본 논문에서는 입출력 파라미터 무변환, 무복사 입출력 (zero-copy I/O), 큐 교환 (queue swapping) 기법을 제안한다.

기존 리눅스 입출력 스택은 계층적으로 구성되어 있으며 입출력이 각 계층을 통해 처리될 때마다 입출력 파라미터들이 각 계층에 알맞은 구조체로 변환되어 전달된다. 예를 들어, 직접 입출력(Direct I/O) 요청 처리는 실제 파일 시스템은 가상 파일 시스템으로부터 받은 입출력 요청 파라미터를 DIO 구조체로 만든다. 그리고 DIO 구조체로 관리되는 입출력 요청 정보는 블록 계층에 입출력 요청이 전달될 때 파일 시스템이 다시 BIO 구조체로 변환하여 블록 계층에 전달한다. 이렇게 여러 단계로 관리되는 입출력 요청 정보들은 고속 저장 장치 환경에서는 무시할 수 없는 소프트웨어 오버헤드를 보인다. 이러한 점을 개선하기 위해 BIO 구조체 대신 가상 파일 시스템에서 받은 입출력 요청 파라미터들을 (파일 포인터, 버퍼 주소 등) 혹은 파일 시스템에서 사용하는 자료구조를 디바이스 드라이버가 받을 수 있도록 디바이스 드라이버 인터페이스를 개선하였다.

무복사 입출력은 입출력 요청을 처리할 때 응용 프로그램의 버퍼와 리눅스 커널의 버퍼 사이의 메모리 복사 연산을 제거하는 것이다. 즉, 응용 프로그램이 입출력 요청을 할 때 전달하는 파라미터 중에서 버퍼 주소[3]를 가로채서 그 버퍼의 물리 주소를 디바이스 드라이버에 전달하여 저장 장치가 이 주소에 DMA를 통해 데이터를 쓰거나 데이터를 읽어갈 수 있게 한다. 이를 위해 디바이스 드라이버를 버퍼 주소를 물리 주소로 처리할 수 있게 드라이버 인터페이스를 구현하였다. 이러한 무복사 입출력은 커널과 응용 프로그램 사이의 데이터 복사 오버헤드를 줄이기 때문에 입출력 지연 시간을 단축시키고 CPU 사용률을 감소시켜 준다.

큐 교환은 시간적 병합의 락 경합을 줄여주는 방법이다. 기존의 시간적 병합은 다음과 같이 처리된다. 여러 CPU에서 동시에 입출력 요청을 처리할 때 입출력 요청들을 디바이스 드라이버에 존재하는 하나의 큐에 저장한다. 그리고 여러 CPU 중에서 하나의 CPU가 큐에 저장되어 있는 입출력 요청들을 한 번에 처리하고 처리가 끝나면 대기하고 있는 다른 CPU에 처리가 끝났음을 알려준다. 입출력 요청을 처리하는 중에는 큐에 배타적인 락이 선언되어 다른 CPU에서는 접근이 불가능하며, 새로운 입출력 요청을 하는 CPU는 락이 해제될 때까지 대기해야 한다. 이 경우에는 큐에 입출력 요청을 삽입하고 입출력 요청을 처리하기 위해 각각 1번씩 총 2회의 락을 얻기 위해 경합을 한다.

위와 같은 2회의 락 경합을 줄이기 위해 본 연구의 디바이스 드라이버는 입출력 요청 큐를 두 개를 할당한다. 하나의 큐는 현재 처리하고 있는 입출력 요청을 저장하고 있는 큐 자료구조이며 다른 하나는 새로운 입출력 요청을 삽입하기 위한 자료구조이다. 즉, 입출력 요청을 모두 처리하면 큐에 있는 입출력 요청을 모두 비우고 비워진 큐에 새로운 입출력 요청을 삽입하게 한다. 그리고 입출력 요청이 저장되어 있는 다른 큐에 배

3. 응용 프로그램이 전달한 버퍼 주소는 가상 메모리 주소이다.

타적인 락을 잡은 CPU가 입출력 요청들을 처리한다. 이러한 방법으로 기존 2회의 락 경합을 1회로 줄일 수 있게 된다.

3.3 파일 시스템 수준 최적화

3.2절에서 제안한 디바이스 드라이버는 기존의 블록 인터페이스 대신에 파일 시스템에서 사용하는 파라미터 혹은 자료구조를 직접 처리할 수 있는 인터페이스를 제안하고 있기 때문에 파일 시스템의 변경이 필요하다. 특히 저널링과 같은 파일 시스템의 고유한 기능에 영향을 주지 않으면서 입출력 관련된 부분에서 새로운 인터페이스를 사용할 수 있도록 리눅스 파일 시스템을 최적화하였고 본 논문에서는 크게 메타데이터 입출력 그리고 직접 입출력 부분을 최적화한다.

파일 시스템의 메타데이터 처리는 메타데이터 입출력에 관한 정보를 가지고 있는 buffer_head (bh) 구조체를 이용하여 이루어진다. 메타데이터 입출력 처리는 submit_bh 함수를 통해 수행되는데 이 함수의 파리미터는 bh 구조체이다. 이 함수를 수행하는 동안에 bh 구조체의 정보는 다시 BIO 구조체를 생성하는데 이용되며 생성된 BIO 구조체는 submit_bio 함수의 파리미터로 전달된다. 메타데이터 처리 부분을 최적화하기 위해서 submit_bio 함수에서 bh 구조체를 파리미터로 하여 최적화된 디바이스 드라이버에서 제공하는 ni_submit_bh 함수에 전달하여 메타데이터 입출력 요청을 수행하도록 한다. 이러한 최적화는 [그림 3]과 같이 기존 입출력 스택에서 블록 계층을 완전히 우회할 수 있으며 파라미터 변환에 따르는 오버헤드를 제거할 수 있다.

파일 시스템의 직접 입출력은 (Direct I/O) 가상 파일 시스템에서 입출력 요청을 위한 파라미터를 받아서 DIO 구조체를 생성한다. DIO 구조체에 저장되어 있는 입출력 정보는 블록 계층에 전달되기 위해 BIO 구조체를 생성한 후에 DIO 구조체에 저장되어 있는 정보를 BIO 구조체로 복사한다. 그리고 메타데이터 입출력과 마찬가지로 submit_bio 함수를 호출하여 블록 계층에 BIO 구조체로 되어 있는 입출력 파라미터를 전달한다. 직접 입출력 처리를 개선하기 위해 DIO 구조체를 생성

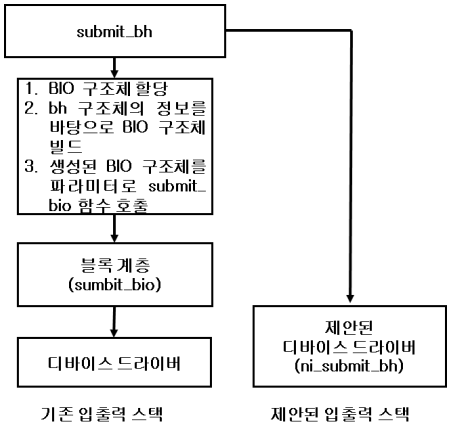

그림 3. 메타데이터 입출력 비교

하기 전에 가상 파일 시스템으로부터 주어진 입출력 요청 정보를 최적화된 디바이스 드라이버에 전달한다. 가상 파일 시스템으로부터의 입출력 요청 정보는 파일 포인터, 버퍼 주소, 입출력 양 등이며 디바이스 드라이버에는 버퍼 주소, 입출력 양, 그리고 파일 포인터 대신에 섹터 주소를 전달한다. 이 때 사용되는 파일 시스템 함수는 ext2(/4)_get_block 함수이며 파일 포인터를 파라미터로 하여 호출하면 섹터 주소를 얻어올 수 있다.

IV. 성능 평가

4.1 실험 환경

실험을 위해 Intel Xeon CPU E5630 2.53GHz를 장착한 서버를 사용하였다. 이 서버 시스템은 8개의 코어와 8GB 메인 메모리를 가지고 있으며, 본 연구의 입출력 스택은 Linux 커널은 2.6.32에 구현되었다. 현재 PRAM/MRAM과 같은 비휘발성 메모리 기반 고속 저장 장치를 쉽게 구할 수 없다. 하지만 Intel/삼성과 같은 반도체 기업들의 예측에 따라 개발될 비휘발성 메모리 소자의 성능이 현재의 DRAM의 성능과 비슷할 것이라 예측하고 있기 때문에 본 논문은 고속 저장 장치를 위해 DRAM을 저장 소자로 사용하는 [그림 4]와 같은 DRAM 기반 저장 장치를 활용하였다[12].

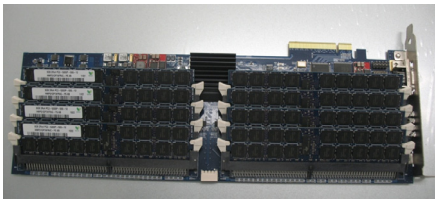

그림 4. 실험에 사용된 DRAM 기반 SSD

4.2 입출력 지연 시간 분석

3.1절의 실험과 같이 쓰레드가 하나인 하나의 프로세스가 한 개의 페이지에 대한 읽기 연산을 직접 입출력 모드로 수행했을 때 입출력 지연 시간을 분석하였다. [표 2]는 입출력 스택 내의 각 계층별 입출력 지연 시간을 표시한 결과이다. [표 1]과 비교했을 때 제안하는 입출력 스택은 블록 계층을 완전히 우회하였으므로 블록 계층의 지연 시간은 0이다. 그리고 무복사 입출력, 큐 교환 등의 디바이스 드라이버 수준의 최적화 기법 때문에 디바이스 드라이버의 입출력 지연 시간은 5us에서 4us로 줄었다. 그리고 파일 시스템의 직접 입출력 부분을 최적화를 통하여 파일 시스템 수준의 입출력 지연 시간은 8us에서 0.5us로 줄었다. 전체적으로 고려하면 전체 지연 시간은 28.8us에서 15.3us로 47% 개선되었으며 S/W 지연 시간은 23.8us에서 10.3us로 57% 개선되었다.

표 2. 최적화된 입출력 스택 지연 시간 프로파일 결과 (ext2 파일 시스템)

입출력 스택		시간(us)	비율(%)
SW 계층	VFS	5.8	37.9
	FS	0.5	3.3
	BLK	0	0
	DEV	4	26.1
총 S/W 지연시간		10.3	67.3
저장 장치 지연시간		5	32.7
전체 지연시간		15.3	100

4.3 실험 결과

이 절에서는 마이크로 벤치마크와 실제 응용인 메일 서버와 데이터베이스 시스템을 이용하여 수행한 실험 결과에 대해서 설명한다. 마이크로 벤치마크는 FIO 벤치마크를[13] 활용하였다. 실험은 하나의 FIO 프로세스가 8개의 쓰레드를 가지며 각 쓰레드는 4KB 단위로 3GB의 랜덤 읽기/쓰기 연산을 수행한다. 이 실험은 직접 입출력 모드에서 진행되었으며 [표 3]은 실험 결과를 보여준다. 읽기 연산에서는 최적화된 입출력 스택의 처리량은 1033MB/s이며 처리량이 531.6MB/s인 기존 입출력 스택과 비교했을 때 처리량은 1.9배 높다. 지연 시간은 기존 입출력 스택에서 66.7us이며, 최적화된 입출력 스택은 31.6us로 54% 정도 지연 시간이 개선되었다. 마찬가지로 쓰기 연산에서도 최적화된 입출력 스택은 885MB/s 그리고 기존 입출력 스택은 428MB/s의 성능을 보여주며 읽기 연산 결과와 유사한 성능 개선 효과를 볼 수 있었다.

표 3. 최적화된 입출력 스택의 처리량과 지연 시간 (ext2 파일 시스템)

		처리량 (MB/s)	지연시간(us)
읽기	기존 입출력 스택	531.6MB/s	66.7us
	최적화된 입출력 스택	1033MB/s	31.6us
쓰기	기존 입출력 스택	885MB/s	35us
	최적화된 입출력 스택	428MB/s	81us

실제 워크로드 환경에서 실험을 수행하기 위해 메일 서버 벤치마크와 데이터베이스 시스템에서 많이 사용하는 TPC-C 벤치마크를 이용하였다. 메일 서버 벤치마크는 FileBench를[14] 사용하였으며 1000개의 쓰레드가 총 10000개의 평균 6MB의 파일을 대상으로 하여 실험을 진행하였으며 [표 4]는 그 결과를 보여준다. 최적화된 입출력 스택은 ext2/ext4 파일 시스템에서 1000 MB/s 정도의 처리량을 보여주며 기존 입출력 스택과 비교하여 25% 정도의 성능 향상 효과를 보여준다.

표 4. 메일 서버 벤치마크 처리량 결과

	ext2	ext4
기존 입출력 스택	766MB/s	771MB/s
최적화된 입출력 스택	1008MB/s	1010MB/s

TPC-C 실험을 위해 데이터베이스 시스템은 MySQL을 그리고 벤치마크 소프트웨어는 benchmark SQL을[15] 사용하였다. 세션의 개수는 100개, 웨어하우스의 수는 50, 실험 시간은 1시간으로 하여 분당 트랜잭션 처리 수를 측정하여 [그림 5]에 표시하였다. 25000tpmC 정도의 성능을 보인 최적화된 입출력 스택은 기존 스택 대비 1.63배 정도 성능이 더 좋음을 알 수 있다.

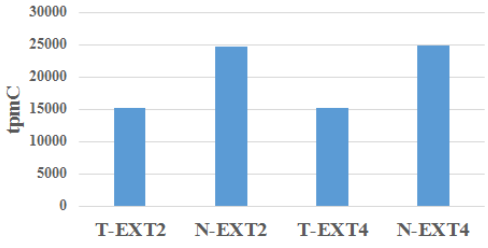

그림 5. TPC-C 결과(T:기존 입출력 스택, N: 최적화된 입출력 스택)

V. 결론

본 논문은 기존 리눅스의 입출력 스택을 고속 저장 장치에 최적화하는 것을 제안한다. 이를 위해 기존 리눅스의 블록 계층은 우회하고, 디바이스 드라이버를 파일 시스템에서 직접 사용할 수 있도록 인터페이스를 개선하였다. 그리고 디바이스 드라이버에 큐 교환, 무 복사 입출력 등의 최적화 기법을 구현하였다. 그리고 이러한 디바이스 드라이버를 파일 시스템이 직접 호출할 수 있도록 리눅스의 ext2/ext4 파일 시스템을 최적화하였다. 벤치마크 실험 결과를 통해서 본 논문에서 제안하는 입출력 스택은 기존 리눅스 입출력 스택과 비교하여 평균 처리량은 1.7배 정도 좋아지고 지연시간은 50%정도 감소함을 확인할 수 있었다.

향후에는 이 논문에서 다루지 않은 파일 시스템의 버퍼된 입출력 (buffered I/O) 및 저널 입출력을 (journal I/O) 고속 저장 장치에 최적화하는 연구와 ext2/ext4 외에도 JFS, XFS와 같은 다른 파일 시스템에도 최적화 기법을 적용하는 연구를 진행하려고 한다.

참 고 문 헌

[1] A. Huffman, “NVM Express Overview & Ecosystem Update,” In Proceedings of Flash Memory Summit 2013.

[2] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt., “Scheduling algorithms for modern disk drives,” In Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems (SIGMETRICS '94), 1994.

[3] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson, “Providing safe, user space access to fast, solid state disks,” ASPLOS 2012.

[4] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson, “Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories,” MICRO 2010, 2010.

[5] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson, “From ARIES to MARS: transaction support for next-generation, solid-state drives,” In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13), 2013.

[6] Dong In Shin, Young Jin Yu, Hyeong S. Kim, Jae Woo Choi, Do Yung Jung, and Heon Y. Yeom, “Dynamic interval polling and pipelined post I/O processing for low-latency storage class memory,” In Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems, 2013.

[7] J. Yang, D. B. Minturn, and F. Hady, “When poll is better than interrupt,” In Proceedings of the 10th USENIX Conference on File and Storage

Technologies, 2012.

[8] Eric Seppanen, Matthew T. O'Keefe, and David J. Lilja, "High performance solid state storage under Linux," In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, 2010.

[9] X. Wu and A. L. N Reddy, "SCMFS: A file system for storage class memory," In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, 2009.

[11] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda, "Swapping to remote memory over InfiniBand: An Approach using a High Performance Network Block Device," In Proceedings of the 2005 IEEE Cluster Computing.

[12] TailwindStorage, "Extreme 3804, http://www.taejin.co.kr"

[13] FIO Benchmark, "http://freecode.com/projects/fio"

[14] FileBench Benchmark, "http://filebench.sourceforge.net/"

[15] BenchmarkSQL, "http://sourceforge.net/projects/benchmarksql"

저 자 소 개

한 혁(Hyuck Han) 정회원

- 2003년 8월 : 서울대학교 컴퓨터공학부(공학사)
- 2006년 2월 : 서울대학교 컴퓨터공학부(공학석사)
- 2011년 2월 : 서울대학교 컴퓨터공학부(공학박사)
- 2011년 3월 ~ 2012년 8월 : 서울대학교 컴퓨터공학부 박사후 연구원
- 2012년 9월 ~ 2014년 2월 : 삼성전자 메모리 사업부 책임연구원
- 2014년 3월 ~ 현재 : 동덕여자대학교 컴퓨터학과 조교수

<관심분야> : 데이터베이스 시스템, 병렬 프로그래밍, 분산 시스템