SSD의 등장에 따른 OS의 변화

염 헌 영
서울대학교 컴퓨터공학부

Ⅰ. 개 요

플래시 메모리에 기반을 둔 고성능 저장장치가 빠르게 등장하고 있지만 실제 응용프로그램의 성능은 그에 비해서 많은 차이가 나지 않고 있다. 본 논문에서는 그 이유에 대하여 살펴보고자 한다. 한마디로 말해서 이는 SSD와 같은 고성능 스토리지의 등장에도 불구하고 운영체제의 스토리지 시스템이 여전히 하드 디스크에 대하여 최적화를 하고 있기 때문이다. 하드 디스크의 응답시간은 수 millisecond 이상으로 그 동안에 OS에서는 많은 일을 할 수가 있었고 될 수 있는대로 I/O의 회수를 줄이는 것을 가장 우선과제로 삼아 왔다. 하지만 빠른 저장장치가 등장하면서 이러한 일들에 대한 우선순위가 바뀌어야 하는데 그런 일들이 어떻게 진행되어 왔으며 어떻게 발전해나갈지를 살펴보고자 한다.

본 논문에서는 SSD기반인 스토리지 장치가 I/O 요청을 처리 할 때 커널 I/O Stack에서 발생한 병목 현상에 대하여 실험을 통하여 알아보고 분석하고자 한다. 여기에서 언급하는 I/O Stack 병목 현상은 현재 운영체제 환경에서 SSD같은 고속저장장치가 스토리지 장치로 사용될 때 소프트웨어 적인 원인으로 인해 최대성능이 나오지 않는 것을 이야기한다. SSD 같은 새로운 장치가 기존의 운영체제 환경에서 사용될 때 최대성능이 발휘되지 못하는 문제점이 있는데, 이를 시스템 차원에서 실험을 통해 알아보고 이를 해결할 수 있는 각종 방안을 제안하고 분석하기로 한다.

Ⅱ. 배 경

초창기의 운영체제의 이름은 대부분 DOS(Disk Operating

System)[1]이다. 초기상태로 컴퓨터를 시작하기 위해서는 뭔가 프로그램이 필요한데 이것을 저장할 수 있는 장치가 디스크밖에는 없었다는 것이다. 물론 메모리가 있기는 했지만 양도 적었고 전원이 꺼지면 저장하고 있던 내용이 모두 없어지기 때문에 디스크는 메모리에 비해서 아주 느리긴 했지만 중요한 역할을 해왔다. 메모리를 읽고 쓰는 것보다 디스크를 읽고쓰는 데는 시간이 엄청나게 오래 걸린다. 디스크는 기계적인 장치여서 매번 다스크 암이 움직여서 필요한 위치로 가야하고 디스크 판이 돌면서 기록된 부분이 암 밑에 올때까지 기다려야 했다. 이러한 기계적인 움직임은 보통 천분의 일초 단위의 시간이 걸린다. CPU의 속도에 비교하면 작게는 수십만에서 수천만배에 이르기까지의 시간이다. 당연히 디스크에서 읽고쓰는 것을 될 수 있는대로 줄이는 것이 운영체제의 첫번째 과제가 될 수 밖에 없었다. 그동안 운영체제에도 많은 발전이 있어왔지만 아직까지도 디스크가 저장장치의 근간이라는 사실은 과거 오십년간 변화가 없었다.

최근에 와서 여기에 변화가 생기기 시작했다. SSD(Solid State Disk)의 등장이다. 플래시 메모리에 기초한 디스크 나아가서는 DRAM(Dynamic Random Access Memory)이나 PCM(Phase Change Memory)[2]에 기반한 고성능 저장장치가 등장하기 시작하면서 운영체제가 바라보는 저장장치의 개념을 바꾸기 시작한 것이다. 특히 일부 특화된 SSD는 초당 수백만번의 I/O요청을 해결할 수 있다[3]. DRAM이나 PCM은 매우 빠르지만 Flash Memory만 해도 읽는데 50ms, 쓰는데는 200ms 정도 걸리고 기계적인 움직임이 없기 때문에 어떤 위치에 읽고 쓰던 비슷한 시간이 걸린다는 하드디스크와는 전혀 다른 특성을 가지고 있다. 기존의 하드디스크에 비교하면 거의 백배 이상의 속도를 낼 수 있는 것이다. 이에 따라서 당연히 운영체제도 바뀌어야 하지만 그 속도가 좀 더디다는 것이 이 논문의 요점이다. 운영체제에서 스토리지를 담당하는 부분은 I/O Stack이다. 이 I/O Stack이 어떻게 바뀌어 왔으며 어떻게 발전해나가야하는지 살펴보도록 하자.

I/O Stack은 OS 스토리지 스텍의 일부로서 I/O 요청를 다루는 역할을 한다. I/O Stack을 설계할 때 하드웨어의 특성을 반영해야 해당 디바이스의 최대 I/O 성능을 끌어올릴 수 있다. 현재 Linux는 최근 수십년 동안에 하드디스크에 대하여 최적화를 해 왔는데, 이 최적화 중 가장 중요한 것은 하나의 I/O operation에 최대한 데이터 양을 들어갈 수 있도록 연속된 I/O 요청을 merge시키는 것이다. 이 과정에서 운영체제는 하드웨어 seek 시간을 줄이기 위해 I/O 요청의 실행순서를 바꾸고, CPU를 더 효율적으로 쓰기 위해 I/O subsystem에 I/O 요청을 하드웨어에 던지고나서 CPU 자원을 다른 job에 넘기고 자신이 I/O 처리가 끝나기를 기다린다.

SSD를 비롯한 고성능 스토리지 디바이스들이 시장에 등장하여 기계적인 데이터 액세스는 더 빠른 전기적인 액세스로 바뀌어 액세스 latency는 수천배 줄어들었다. 이 상황에서는 하드웨어 seek에 대한 고려를 하지 않아도 되는데, 앞서 진행된 I/O요청에 대한 최적화는 빠르게 처리되어야 할 I/O요청 처리를 지연시켜 SSD의 최고 성능을 내는 데 오히려 방해가 된다. 가장 먼저 눈에 띈 것은 I/O sleep() 이다. 기존의 하드디스크에서는 한번 I/O가 일어나면 최소 수 milisecond가 걸리기 때문에 무조건 CPU를 다른 쓰레드나 프로세스로 넘겨주는 것이 정상이었다. 하지만 빠른 스토리지에서는 이 시간이 수 microsecond에 불과하다. 따라서 끝날때까지 기다리는 것이 훨씬 현명한 선택일 수 있게 된다[4]. 또한 I/O요청을 병합함에 있어서도 기존의 하드디스크에서는 물리적으로 인접한 것들만 병합하는 것이 가능했지만 이제는 물리적인 위치에 상관없이 병합하여 요청의 회수를 줄이는 것이 훨씬 효과적이 되었다[5]. 또한 이런 병합된 요청들에 대한 결과를 한꺼번에 돌려보내는 것보다 빨리 끝난 순서대로 알려줌으로 좀더 빠른 대응이 가능하게 되었다[6]. 이런 기본적인 최적화 이외

운영체제에서 스토리지를 담당하는 부분은 I/O Stack 이다. I/O Stack을 설계할 때 하드웨어의 특성을 반영해야 해당 디바이스의 최대 I/O 성능을 끌어올릴 수 있다.

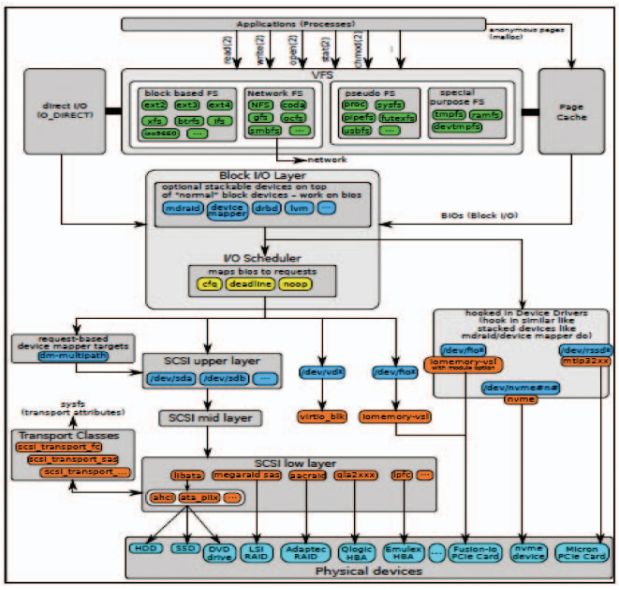
에도 다른 여러가지가 가능할텐데 이러한 요소의 영향을 알아보기 위해 기존 I/O subsystem에서 I/O요청의 처리과정에 대하여 소개하고 분석하도록 하겠다.

Ⅲ. I/O Path에 대한 분석

I/O 요청(I/O request)은 Linux 시스템에서 I/O를 수행하기 위해 만드는 구조체이다. 이러한 I/O요청안에 하드웨어 디바이스에서 I/O가 일어나기 위해 필요한정보들이 (예로 들면 sector 주소, 메모리 segment, transfer 사이즈 등) 저장되어 있다. 본 보고서에서는 I/O 요청의 방향에 따라 I/O path를 두 가지로 정의한다. 1). Descending I/O path는 애플리케이션에서 하드웨어 디바이스까지 거치는 커널 함수들이며, 2) Ascending I/O path는 하드웨어 디바이스에서 어떤 애플리케이션까지 올라오는 커널 함수들을 말한다.

> I/O path는, descending I/O path와 ascending I/O path로 정의될 수 있다.

Linux 스토리지 Stack은 Block layer, SCSI subsystem, Device driver등을 포함한다. 〈그림 1〉에서는 Linux I/O stack과 I/O subsystem를 기술하고 있다.



〈그림 1〉 Linux Kernel I /O Subsystem

응용 프로그램에서 I/O요청이 일어나면, VFS를 거쳐 page cache에 필요한 페이지를 요청한다. VFS가 page cache에서 원하는 page를 찾지 못하거나 버퍼 write할 때 page수가 모자라면 직접 I/O request를 만들고 block layer로 I/O를 요청한다. 이렇게 descending path를 초기화 되고 I/O 요청이 block layer로 보내어지게 되는데, 이후 아래의 두 가지 일이 수행된다.

1) I/O scheduling: I/O를 더 효율적으로 처리하기 위해 I/O 요청을 이슈하는 순서를 바꾸는 동작
2) I/O 요청 merge: random workload를 고려하여 연접한 작은 요청을 큰 요청으로 만드는 동작

일정한 조건을 만족하면 block layer의 큐에 쌓였던 I/O요청들은 한번에 SCSI subsystem으로 보내어진다. 이후 SCSI subsystem에서는 DMA를 세팅하고 마지막으로 디바이스 드라이버의 콜백 함수를 호출하여 DMA전송을 시작한다.

스토리지 디바이스가 I/O처리를 끝내면 디바이스 드라이버에게 인터럽트를 걸어 그 사실을 통보한다. 다바이스 드라이버가 해당 인터럽트를 받고 나서 SCSI 시스템이 할당해준 자원들을 풀고 나머지 completion처리는 local SoftIRQ 핸들러로 넘기는 식으로 ascending path를 마친다.

Ascending path에 있어서 I/O요청에 대한 completion처리가 일어나는 동안 CPU는 다음 4가지의 콘텍스트에 속할 수 있다. 이는 1) 사용자 프로세서, 2) 커널 스레드, 3) SoftIRQ 핸들러 또는 4)인터럽트 핸들러를 말한다. 여기서 CPU콘택스트가 위에 언급하는 4가지의 콘택스트사이에서 스위치할 때마다 지연 시간이 발생하게 된다. 이러한 콘텍스트의 변화는 I/O가 비 동기 적으로 처리되기 때문에 나타나는 현상이다.

Ⅳ. I/O 병목현상에 대한 실험 및 분석

가. 실험설계 및 실험환경

본 절에서는 기존 커널의 I/O subsystem 에 대하여

〈표 1〉 I/O 부하의 환경설정

Benchmark	Configuration
dd	1 thread, RecSize {0.5,1,2,4,8 KB}, I/O=direct
IOZONE	1 thread & 32threads, RecSize=4KB, I/O buffered, FileSystem=ext2

소프트웨어 대역폭, 함수별 지연, 또는 인터럽트 처리 시간등 3개의 실험을 통해서 병목현상 있는지 평가하고 그의 원인을 분석한다. 본 실험에서 사용한 장비는 4개의 Xeon E5630 2.53GHz 쿼드코어(총 16 코어)와 16GB RAM를 갖고 있다. 운영체제는 Linux이고, 2.6.32 vanilla 커널을 사용하였다. 측정을 위해 사용한 스토리지는 DRAM-SSD[7]이다. 기타 본 실험에서 사용하는 I/O 부하는 다음과 같다[8].

이 실험에서 I/O Subsystem의 각 함수를 실행할 때 시간이 얼마나 필요하는지를 더 쉽게 보기위해 Systemtap라는 커널레벨 프로파이링 툴을 사용였다. 여기서 우리는 I/O path의 latency를 분석하고 각 층에서 걸리는 시간에 대하여도 timeline을 분석해 보기로 한다.

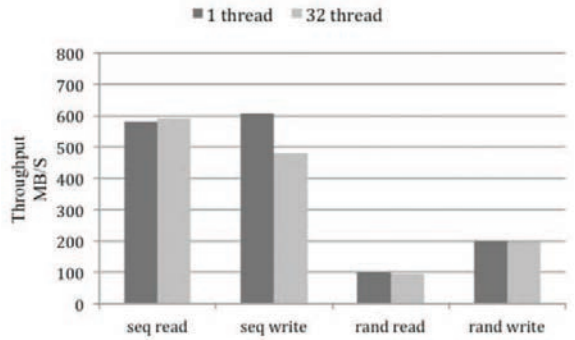
DRAM-SSD는 순수한 소프트웨어 적인 병목현상을 확인하고자, 스토리지 technology의 영향을 최소화 하기 위한 의도로 채택되었다. 즉, SSD내부 컨트롤러의 내부동작에 대한 영향을 배제하는 차원에서 DRAM-SSD가 사용된 것이다. 해당 DRAM-SSD의 하드웨어 latency는 4KB 페이지를 접근할 때 약 7us이고, 대역폭은 최대 700MB/s 이다.

나. 실험결과

디스크 기반인 I/O subsystem이 Fast 스토리지에 영향을 얼마나 미치는지 보기위해 기존 Linux I/O 시스템에서 사용하는 DRAM-SSD에 대해 IOZONE을 통해 실험한다. 실험결과는 다음과 같다.

여기서 우리는 〈그림 2〉을 통하여 두 가지의 성능 차이가 존재한다는 것을 알 수 있다.

- 소프트웨어 대역폭은 하드웨어 대역폭보다 작다



〈그림 2〉 스레드수 별 I/O 부하에 대한 성능 측정

- Random workload 의 대역폭은 Sequential workload 의 대역폭보다 작다

HDD같은 경우 하드웨어 latency가 대부분의 I/O 처리 시간을 차지하기 때문에 (1ms~10ms) 소프트웨어 latency가 별로 중요하지 않으며 하드웨어 대역폭도 거의 그대로 애플리케이션에서 반영될 수 있다. 그렇지만 고성능 스토리지 디바이스 같은 경우는 작은 소프트웨어 latency 이어도 애플리케이션 측에서 큰 성능 저하를 초래할 수 있다. 이 현상을 알아보기 위해 스토리지 스택의 각층에서 일어나는 오버헤드가 얼마인지 분석해 보기로 한다.

SSD와 같은 고성능 스토리지 디바이스에서는 작은 소프트웨어 latency라도 애플리케이션 측에서 큰 성능 저하를 초래할 수 있다.

I/O 요청당 latency 분석을 더 쉽게 하기 위해 싱글 스레드 I/O 부하(dd ,1 thread, seq. read, 4KB direct I/O)를 사용하여 I/O Stack의 각 함수가 실행할 때 latency가 얼마인지를 실험한다. 해당 I/O부하는 두 번째 I/O 요청이 반드시 첫번째 I/O 요청이 끝나야 I/O subsystem에 들어갈 수 있는 특성이 있다. 〈표 2〉는 이 실험에서 하나의 I/O request에 대한 latency breakdown이다.

〈표 2〉의 내용을 보면 소프트웨어 latency(53us)가 하드웨어 latency(7us,테이블에서 R로 표시한다)보다 훨씬 크다는 사실을 알게 되고 I/O 병목은 더 이상 하드웨어 latency가 아니라는 것도 알 수 있다.

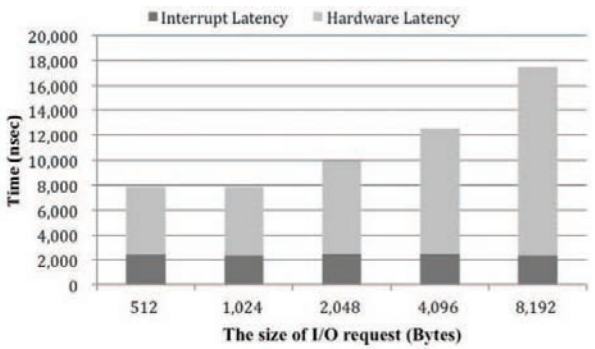
I/O 병목 현상을 만들 수 있는 또 하나의 원인은 인터럽트 latency이다. 인터럽트 latency가 얼만인지 보

〈표 2〉 Linux Kernel Latency Breakdown

Layer	Context	Function	In/Out	Time
VFS	dd	do_sync_read	in	0
BLK	dd	generic_make_request	in	4
BLK	dd	generic_make_request	out	7
SCSI	dd	scsi_request_fn	in	9
SCSI	dd	scsi_request_fn	out	16
VFS	dd	io_schedule	in	18
DEV	interrupt	SSD_intr	in	R+18
DEV	interrupt	SSD_intr	out	R+27
BLK	softirq	blk_done_softirq	in	R+36
BLK	softirq	bio_endio	in	R+39
BLK	softirq	bio_endio	out	R+41
SCSI	softirq	scsi_run_queue	in	R+45
SCSI	softirq	scsi_run_queue	out	R+46
BLK	softirq	blk_done_softirq	out	R+47
VFS	dd	io_schedule	out	R+51
VFS	dd	do_sync_read	out	R+53
VFS	dd	do_sync_read	in	R+56

기 위해 다음 그림과 같이 I/O 요청의 사이즈별 인터럽트 latency와 전체 하드웨어 latency를 비교한다.

〈그림 3〉에 나온 바와 같이 인터럽트 latency는 2-3us 되고 DRAM-SSD인 경우 전체 하드웨어 latency에 차지한 비율은 작지 않다. 하드 디스크를 사용할 때는 하드웨어 latency가 매우 커서 인터럽트 latency가 전체 하드웨어 latency에서 차지한 비율이 0.1%도 미만



〈그림 3〉 I/O request 크기 별 Interrupt Latency 및 Hardware Latency

하지만 Fast 스토리지 디방이스인 경우 그림에서 나오는 것 처럼 I/O 사이즈가 4KB가 될 때 거의 20%를 차자하고 512byte인 경우는 40%를 넘는다.

DRAM-SSD가 아니라 상용화된 flash SSD에서도 위에서 측정된 latency는 큰 영향을 끼칠 수 있다. 이러한 장치는 4KB I/O 요청을 처리 할 때 latency가 50us-100us 정도로 나타나는데 이 때 인터럽트 처리시간은 4%정도를 차지하여 작은 비중이 아니는것으로 보일 수 있다. 더불어 위에서 언급한 소프트웨어 latency (53us)는 flash SSD에서도 매우 큰 비중을 차지한다.

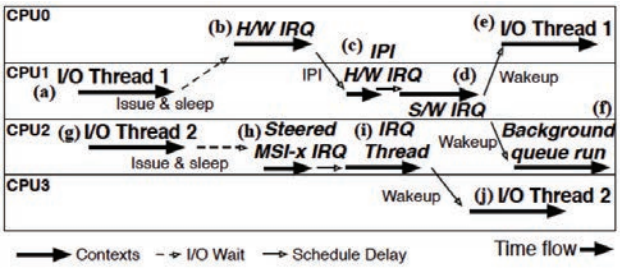
다. Descending Path에 대한 분석

Descending Path에서 발생하는 I/O 오버헤드는 현재 커널의 I/O subsystem이하드디스크 기반 최적화때문에 생기는 것으로 보인다. 이중에 제일 심각한 문제는 I/O 스케줄러이다. 현재 커널의 I/O 스케줄러가 요청 큐에서 일정한 개수의 요청을 갖고 있으며 만약에 큐에 쌓이는 요청의 개수가 임계치를 넘으면 스토리지 장치에 한번에 플러시한다. 그러지만 기존 시스템이 디스크기반 최적화를 하기 때문에 이 임계값이 너무 작다. SSD같은 고성능 디바이스가 I/O요청을 처리하기가 빠르니까 I/O 스케줄러가 빈번하게 플러시 해야 하는데, 이에 따라서 I/O 스케줄러의 제일 중요한 역할인 spatial merge의 효과가 거의 없는 것으로 보인다.

I/O 병목 현상을 만들 수 있는 또 하나의 원인은 인터럽트 latency 이다.

라. Ascending Path에 대한 분석

Ascending Path에서 발생하는 소프트웨어 latency는 주로 I/O path의 비동기적인 설계 때문에 생기는 것으로 보인다. 〈표 2〉를 보면 CPU가 interrupt 컨택스트에서 softIRQ 컨텍스트로 스위치 할 때 9us를 걸리고 softIRQ 컨텍스트에서 사용자 컨택스트로 스위치 할 때도 역시4us를 걸린다. Linux I/O 스택에서 하나의 I/O요청을 처리할 때 일반적으로 3,4번 컨택스트 스위치가 필요해서 전체 latency에 대하여 큰 영향을



〈그림 4〉 Interrupt handling routine

미친다.

여기에서는 인터럽트 오버헤드가 어디서 존재하는지에 대해 분석한다. 〈그림 4〉에서 나온 바와 같이 새로운 컨텍스트가 I/O path에 추가될 때 스케줄링 오버헤드도 들어간다. 〈그림 4〉에서 (a-f)는 I/O completion path의 각 콘텍스트이다. 이 중에 (c)는 inter-processor 인트럽트이고, (d)는 소프트웨어 인터럽트이다. f는 배경 큐 실행과정이다. 이 I/O path(a-f) 는 AHCI 컨트롤러를 사용한 SATA 또는 SAS SSD들의 표준 I/O path이다. 이 I/O path는 block layer과 SCSI subsystem이 사용된다. 이 I/O path에서는 I/O post-processing시 CPU 캐시 사용을 더 효율적으로 하기 위해 softIRQ를 특정 CPU로 유도하는데, 이 때 inter processor interrupt(IPI)을 사용한다(c). 그러므로 I/O 요청을 issue하는 스레드(thread 1)와 softIRQ(d)을 같은 CPU에서 돌릴 수 있어서 컨택스트 스위치로 인한 과도한 캐시 무효화를 피할 수 있는 것이다. 그렇지만 I/O 요청을 issue하는 쪽의 스레드(a)와 완료가 되는 쪽(e)의 스레드가 같은 CPU에서 돌리는 것은 보장되지 않는다.

고성능 PCI-Express 기반 SSD을 위한 NVM-E 표준에서도 비슷한 일이 벌어진다. 이 표준에서는 호스트 인터페이스를 확장하여 기존 I/O path을 단순화 한다. NVM-E의 인터페이스는 많은 (64k) 그리고 깊은 (64k) 큐를 제공하며 PCI-E 3.0 MSI-x 인터럽트도 지원한다. 이 경우에는 I/O 요청을 issue한 후에 Block layer와 SCSI subsystem 을 바이패스하고 직접 디바이스 드라이버에 들어가는데, 〈그림 4〉에서 (g-j)는 이 과정을 보여준다. 이 과정에서 깊은 큐(64k)를 쓰기 때문에 위에서 f로 표시한 배경 실행이 필요 없어진다. 또한 MSI-x 인터럽트가 특정 CPU에 고정될 수 있으므로 IPI없어도 인터럽트 처리와 I/O 요청이 같은 CPU에서 수행할 수 있다. 그러나 이 경우에는 softIRQ를 없앨 수 있지만 IRQ 스레드 들을 스케줄 할 때 역시 오버헤드가 들어가 있고 I/O 요청을 완성하는 스레드와 이슈 하는 스레드가 여전히 다른 CPU에서 실행될 수 있다[9].

Ⅴ. 최적화 방안

위에 분석한 바와 같이 고성능 스토리지를 사용할 때 생기는 병목 현상은 주로 I/O 스케줄러와 인터럽트로 인해 생기는 컨택스트 스위치때문에 발생하는 것이다. 따라서 이 문제를 해결하기 위하여 우선 I/O 스케줄러 문제를 해결 해야한다. Block layer에서 특정한 함수들을 등록하면 I/O 요청들이 I/O 스케줄러에 들어가는 것 대신 직접 디바이스로 issue할 수 있는데, 기본적으로 이렇게 하게 되면 I/O 스케줄러로 생기는 문제 해결할 수 있을 것이다. 더불어, I/O가 요청되는 시점으로부터 디바이스로 실제로 issue되는 지점까지의 소프트웨어적인 거리를 줄이는 것도 도움이 될 것이다. 특히 SCSI 레이어로 인해 발생하는 in-direction은 지양되어야 할 필요가 있다.

기존 커널의 I/O subsystem은 SSD등 고성능 스토리지 디바이스에 대해 최적화 고려가 되어 있지 않다.

그러나, 본격적으로는 Post I/O processing할 때 생기는 오버헤드를 줄이기 위해 컨택스트 스위치 지연 문제를 해결해야 한다. 구체적으로 말하면 softIRQ와 background 큐 실행 컨택스트를 먼저 제거하여 발생하는 추가적인 콘텍스트 스위치로 인한 문제를 해결하는 것이 필요하다고 생각된다.

Ⅵ. 결 론

기존 커널의 I/O subsystem은 SSD등 고성능 스토리지 디바이스에 대해 최적화 고려가 되어 있지 않다. 이런 환경에서 SSD을 사용하면 디바이스의 최적 성능을 발휘할 수 없고 I/O 병목 현상이 생길 수 있다. 본 논문에서는 고속 스토리지 디바이스를 사용할 때 발생하는 병목 현상에 대하여 실험을 통해 분석하였다. 이러한 원인에 대한 대안으로 과도한 콘텍스트 스위치를 방지하고자 하는 목표를 설정하였다. 앞으로는 이 부분을 중심으로 Linux의 I/O subsystem에 대한 최적화 작업을 진행할 예정이다.

참 고 문 헌

[1] Disk Operating System, Wikipedia

[2] R.C. Sousa, I.L. Prejbeanu Non-volatile magnetic random access memories (MRAM) C. R. Physique, 6 (2005), p. 1013

[3] Fusion-IO, ioDrive, http://www.fusionio.com/products/iodrive/

[4] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. FAST'12,.

[5] YU, Y. J., SHIN, D. I., SHIN, W., SONG, N. Y., EOM, H., AND YEOM, H. Y. Exploiting peak device throughput from random access workload. USENIX HotStorage 2012.

[6] Dong In Shin, Young Jin Yu, Hyeong S. Kim, Jae Woo Choi,Do Yung Jung, Heon Y. Yeom, "Dynamic Interval Polling and Pipelined Post I/O Processing for Low-Latency Storage Class Memory", USENIX HotStorage 2013

[7] TAILWINDSTORAGE. Extreme 3804, http://www.taejin.co.kr

[8] J.Axboe,"Fiobenchmark, http://freshment.net/projects/fio."

[9] Woong Shin, et al, "OS I/O Path Optimizations for Flash Solid-state Drives", USENIX ATC, June 2014.

염 헌 영

1984년 서울대학교 계산통계학과 학사
1986년 텍사스 A&M 대학교 전산 석사
1992년 텍사스 A&M 대학교 전산 박사
1993년~현재 서울대학교 컴퓨터공학부 교수

〈관심분야〉
클라우드 컴퓨팅, 스토리지 시스템