저작자표시-비영리-변경금지 2.0 대한민국

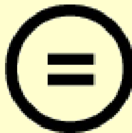
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.


변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

박 지 웅 교수 지도

석사학위 청구논문

CPU 자원 경쟁 환경에서 고성능 저장장치의 입출력 방식에 따른 성능 변화 분석

2024

성신여자대학교 대학원

컴퓨터학과

이 슬 아

CPU 자원 경쟁 환경에서 고성능 저장장치의 입출력 방식에 따른 성능 변화 분석

박 지 웅 교수 지도

이 논문을 석사학위논문으로 제출함

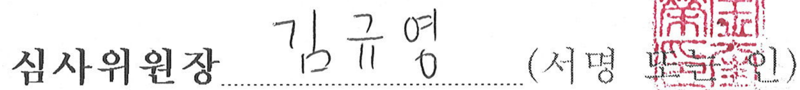
2023년 11월

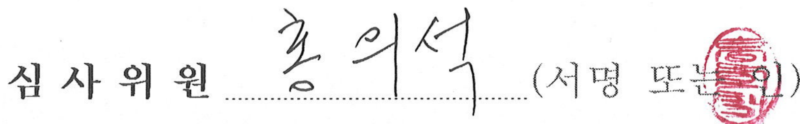
성신여자대학교 대학원
컴퓨터학과
이 슬 아

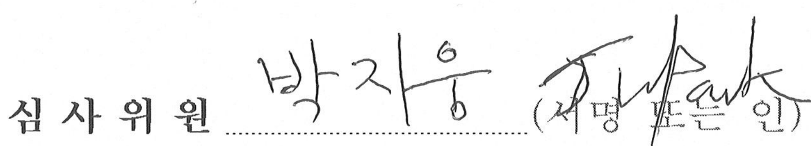
인 준 서

이슬아의 석사학위 논문으로 인준함

2023년 11월


심사위원장 ....(서명 또는 인)


심 사 위 원 ....(서명 또는 인)


심 사 위 원 ....(서명 또는 인)

성신여자대학교 대학원

논문개요

현대의 PCIe 4.0 기반 고성능 초저지연 NVMe 저장장치의 입출력 성능을 극대화하기 위해서는 인터럽트 기반의 입출력 인터페이스보다 폴링 기반의 입출력 인터페이스를 사용하는 것이 좋다고 알려져 있다. 하지만, 폴링 방식의 입출력 처리는 CPU 자원의 독점을 필요로 하기 때문에 CPU 자원 경쟁 상황에서 입출력 성능에 부정적인 영향을 미칠 수 있다. 일반적으로 분산 스토리지 시스템처럼 입출력 처리를 필요로 하는 애플리케이션들은 입출력 이외에도 스토리지 노드 간 일관성 유지 및 데이터 복제를 위해 많은 CPU 자원을 사용한다. CPU 사용량이 높은 분산 스토리지 시스템과 같은 환경에서는 CPU 자원을 독점적으로 사용하는 폴링 기반의 입출력 처리가 인터럽트 기반의 입출력 처리보다 효율적이지 않을 수 있다.

본 논문에서는 분산 스토리지 시스템과 같이 CPU 자원이 부족한 상황에서 고성능 저장장치에 효율적인 입출력 방식을 파악하고자 한다. 이를 위해 분산 스토리지 시스템에서 발생하는 CPU 자원 경쟁 환경을 모사하여 입출력 방식에 따른 성능 변화에 대한 비교 분석을 진행하였다. 저장장치 성능 특성, CPU 간섭 정도, 캐시 크기 대비 데이터셋 크기 비율 등 다양한 요소에 대한 성능 평가를 통해, 여러 작업이 CPU 자원을 두고 경쟁하는 상황에서는 폴링 기반 입출력 방식의 성능 저하가 크게 발생하여, 경우에 따라 인터럽트 기반 입출력 방식보다 오히려 낮은 성능을 보일 수 있음을 확인하였다. 이러한 결과를 바탕으로, 분산 스토리지 시스템에서 최대의 입출력 성능을 달성하기 위해서는 다양한 요소를 고려하여 동적으로 최적의 입출력 방법을 적용할 수 있는 폴링과 인터럽트의 하이브리드 입출력 메커니즘이 필요함을 입증하였다.

목　　차

논문개요

Ⅰ. 서론 ········· 1

Ⅱ. 배경 ········· 3
1. 리눅스 입출력 스택 ········· 3
2. 입출력 이벤트 처리 방식 ········· 5
1) 인터럽트 ········· 5
2) 폴링 ········· 6
3. 비동기적 입출력 인터페이스 ········· 7
1) libaio ········· 8
2) SPDK ········· 8

Ⅲ. 관련 연구 ········· 11
1. 고성능 저장장치를 위한 입출력 스택 최적화 연구 ········· 11
2. 고성능 저장장치에서 입출력 인터페이스에 따른 성능 분석 연구 ········· 15

Ⅳ. 성능 분석 ········· 17
1. 마이크로 벤치마크 ········· 17

1) 실험 환경 ······ 17
2) 실험 방법 ······ 18
3) 실험 결과 ······ 21
2. 매크로 벤치마크 ······ 27
1) 실험 환경 ······ 27
2) 저장장치 성능 특성의 영향 ······ 31
a) 실험 방법 ······ 31
b) 실험 결과 ······ 31
3) CPU 간섭 정도의 영향 ······ 36
a) 실험 방법 ······ 36
b) 실험 결과 ······ 36
4) 캐시 크기 대비 데이터셋 크기 비율의 영향 ······ 39
a) 실험 방법 ······ 39
b) 실험 결과 ······ 39

Ⅴ. 결론 및 향후연구 ······ 42

참고문헌
ABSTRACT

그림 목차

[그림 1] 리눅스 I/O 스택 구조 및 입출력 처리 과정 ························· 4
[그림 2] 이벤트 처리 방식에 따른 CPU 사용 다이어그램 ················ 5
[그림 3] SPDK RocksDB 계층 ·· 9
[그림 4] 하이브리드 폴링 CPU 사용 다이어그램 ······························ 13
[그림 5] PCIe 4.0 기반 NVMe SSD에서의 Random Read IOPS ······ 20
[그림 6] CPU 코어 제한 시 PCIe 4.0 기반 NVMe SSD에서의
Random Read IOPS ·· 20
[그림 7] 저장장치에 따른 입출력 인터페이스별 애플리케이션 성능
변화 ·· 21
[그림 8] 저장장치에 따른 입출력 인터페이스별 입출력 성능 변화 ··· 22
[그림 9] PCIe 3.0 기반 NVMe SSD에서 CPU 경쟁 상황 시
입출력 인터페이스별 성능 ·· 24
[그림 10] PCIe 4.0 기반 NVMe SSD에서 CPU 경쟁 상황 시
입출력 인터페이스별 성능 ·· 24
[그림 11] RockDB 실행 환경에 따른 캐시 사용 ······························ 29
[그림 12] PCIe 3.0 기반 NVMe SSD에서 데이터베이스 환경에 따른
Mixgraph 워크로드 입출력 지연시간 ································· 31

[그림 13] PCIe 3.0 기반 NVMe SSD에서 데이터베이스 환경에 따른 Mixgraph 워크로드 처리량 ······ 32
[그림 14] PCIe 4.0 기반 NVMe SSD에서 데이터베이스 환경에 따른 Mixgraph 워크로드 입출력 지연시간 ······ 33
[그림 15] PCIe 4.0 기반 NVMe SSD에서 데이터베이스 환경에 따른 Mixgraph 워크로드 처리량 ······ 34
[그림 16] CPU 간섭 정도에 따른 성능 변화 ······ 38
[그림 17] 데이터베이스와 캐시 크기 비율에 따른 성능 변화 ······ 41

표 목차

[표 1] 입출력 인터페이스 특징 ······ 7
[표 2] 입출력 벤치마크 실행 환경 ······ 17
[표 3] NVMe SSD 4KB Random Read 성능 ······ 18
[표 4] Mixgraph 워크로드 실행 환경 ······ 27
[표 5] Mixgraph 워크로드 구성 ······ 28

I. 서 론

최근 몇 년 동안 저장장치 기술은 엄청난 성능 향상을 이룩했고, Samsung Z-SSD, Intel Optane SSD, Toshiba XL-Flash와 같은 최첨단 NVMe (Non Volatile Memory Express) SSD는 입출력 지연시간을 이전보다 훨씬 빠르게 단축시켰다[1]. 저장장치의 속도는 수백만 개의 입출력 작업을 수행하면서 최대 수 GB/s의 대역폭을 제공할 수 있을 만큼 급격하게 발전하고 있는 반면, CPU의 발전은 크게 이루어지지 않고 있다[20]. 고성능 저장장치의 성능을 최대한으로 끌어내 사용하기 위해서는 CPU 코어를 효율적으로 사용하는 것이 매우 중요하다. 기존에 사용되던 인터럽트 기반의 입출력 처리 방식에서 발생하는 인터럽트 처리 및 컨텍스트 스위칭 오버헤드는 저장장치의 성능이 낮을 경우 입출력 처리 시간에 비해 무시할 만한 수준이었지만, 초저지연 저장장치 하에서는 더 이상 무시할 수 없는 수준에 이르렀다. 저장장치의 입출력 처리 속도가 빨라지면서 인터럽트 관련 오버헤드가 입출력 지연시간의 최대 15%를 차지한다고 알려졌으며 폴링 방식을 사용하면 이러한 오버헤드를 감소시키고 입출력 지연시간을 줄일 수 있다[5, 6].

일반적인 상황에서 지연시간이 낮은 저장장치를 사용할 때는 폴링 기반의 입출력 방식 사용이 권장된다[7, 18, 29, 43, 45, 46, 47]. 그중에서도 인텔에서 제시한 SPDK(Storage Performance Development Kit)가 높은 처리량을 가지며 다른 입출력 인터페이스와 비교했을 때도 좋은 성능을 보인다고 알려져 있다[7][8]. 그러나 폴링 기반의 입출력 처리 방식은 CPU 자원의 독점적 사용을 필요로 하기 때문에 CPU 자원이 부족한 상황에서는 폴링 방식이 인터럽트 방식보다 더 높은 성능을 보장한다고 확신할 수 없다. 스토리지 서버가 데이터의 저장 및 서비스뿐만 아니라 데이터의 일관성과 가용성을 보장하기 위해 다른 노드들과 상호작용해야 하는 분산 스토리지 시스템 환

경의 특성은 각 스토리지 서버의 CPU 사용률과 크기가 작은 입출력 요청에 대한 CPU 부담을 증가시킨다. 또한, 일반적인 분산 스토리지 시스템에서는 스토리지 서버를 효율적으로 활용하기 위해 하나의 스토리지 서버에 다수의 저장장치가 장착되고 관리된다. 관리하는 저장장치의 수가 늘어날수록 하나의 스토리지 서버에서 처리해야 하는 데이터가 많아지므로 CPU 사용량도 증가하게 된다[9]. 따라서 CPU 사용량이 높은, 즉 CPU 자원이 부족한 분산 스토리지 시스템 환경에서는 폴링 방식이 인터럽트 방식보다 높은 성능을 가질 것이라고 확신할 수 없다.

본 논문에서는 분산 스토리지 시스템에서 발생하는 CPU 자원 경쟁 환경을 모사한 실험을 통해, 어떠한 조건에서 폴링과 인터럽트 방식이 각각 효율적인지 파악하고자 한다.

본 논문의 주요 기여 내용은 다음과 같다.

첫째, 마이크로 벤치마크와 매크로 벤치마크를 통해 저장장치의 성능 특성에 따른 CPU 경쟁 상황에서의 성능 변화를 분석하였다.

둘째, 어떠한 조건에서 폴링과 인터럽트 방식이 각각 효율적일 수 있는지 분석함으로써 향후 폴링과 인터럽트의 하이브리드 입출력 연구를 위한 토대를 마련하였다.

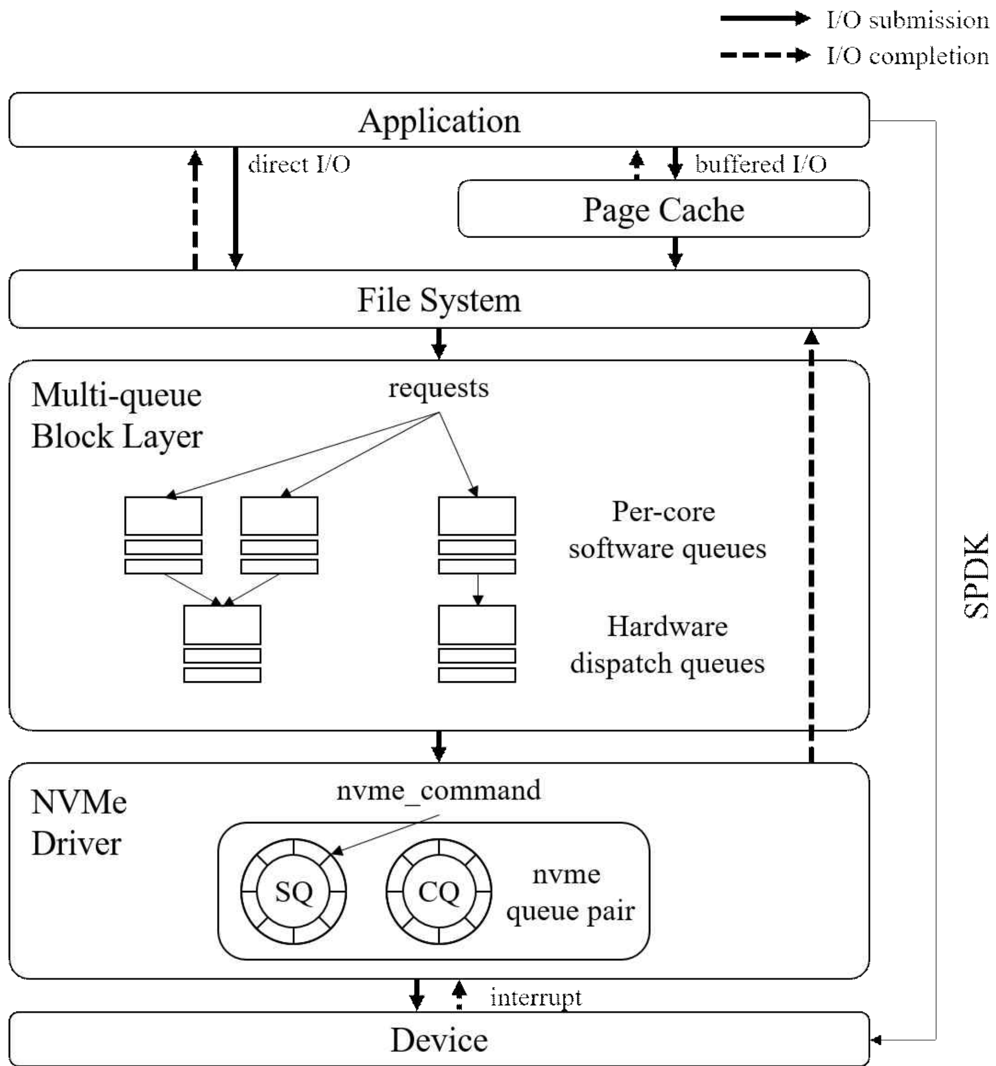
본 논문은 다음과 같이 구성된다. 2장에서는 입출력 방식과 실험에 사용된 각 입출력 인터페이스에 대해, 3장에서는 고성능 저장장치의 성능 향상을 위한 관련 연구들에 관해 서술한다. 4장에서 마이크로 벤치마크와 매크로 벤치마크를 통한 CPU 경쟁 상황에서의 성능 분석을 진행한다. 5장에서는 4장의 실험 결과를 기반으로 한 결론과 향후 연구를 제시하며 마무리한다.

II. 배 경

1. 리눅스 입출력 스택

[그림 1]은 리눅스 시스템의 입출력 스택 구조와 입출력 처리 과정을 보여준다. 이는 페이지 캐시, 파일 시스템, 멀티 큐 블록 레이어 및 NVMe 드라이버와 같은 다양한 레이어로 구성되어 있다. 실선은 I/O submission 경로를, 점선은 I/O completion 경로를 표시한다.

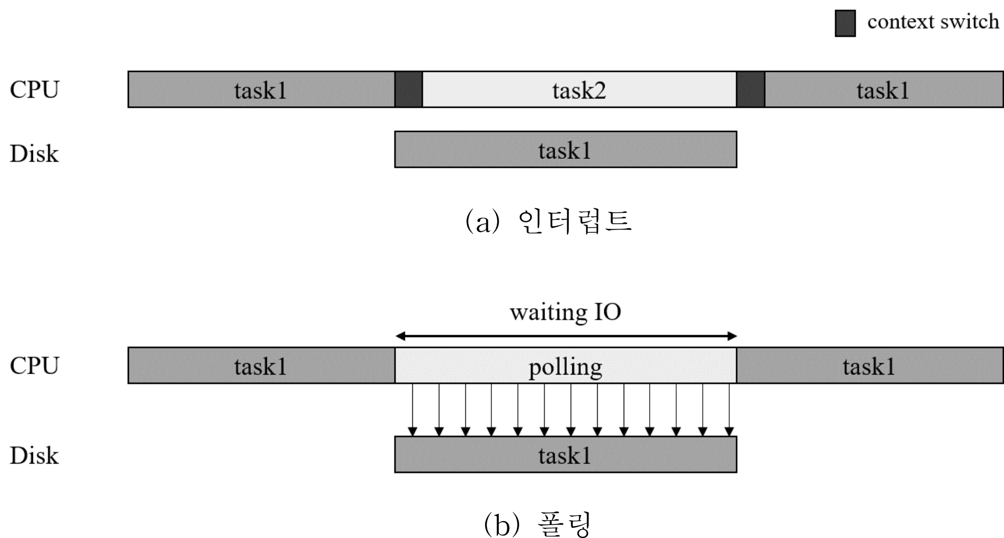
응용 프로그램이 POSIX read/write[21, 22], libaio[10], io_uring[17]과 같은 입출력 인터페이스에 입출력 요청을 제출하면 입출력 인터페이스는 VFS(Virtual File System)를 통해 파일 시스템에 해당 요청을 제출한다. 이때 buffered I/O에서는 페이지 캐시에서 먼저 데이터를 검색하거나 삽입하며 캐시 미스 발생 시, 빈 페이지가 입출력 요청을 버퍼링하는 데 사용된다. 파일 시스템은 페이지를 페이지 캐시에 삽입하고 논리 블록 주소를 검색한 후, bio 구조체를 생성해 블록 레이어에 제출한다. 리눅스의 멀티 큐 블록 레이어(blk-mq)[39]는 입출력 스케줄링 및 디스패치를 수행한다. bio 구조체를 request 구조체로 변환하고 각 코어의 소프트웨어 큐에 삽입한다. 해당 요청은 블록 I/O 스케줄러에 의해 처리되어 하드웨어 디스패치 큐로 넘어가 NVMe 드라이버에 의해 nvme_command 구조체로 변환되고 제출 큐(SQ)에 들어간다. 제출 큐에 들어간 nvme_command는 디바이스에서 처리된 후 완료 큐(CQ)에 기록되며 디바이스는 인터럽트를 보내 요청을 보낸 응용 프로그램에게 입출력 작업이 완료되었음을 알릴 수 있도록 한다. SPDK는 커널 I/O 스택을 우회하여 NVMe SSD에 직접 접근할 수 있으며 이를 통해 컨텍스트 스위치와 시스템 콜로 인한 오버헤드를 줄일 수 있다.



[그림 1] 리눅스 I/O 스택 구조 및 입출력 처리 과정

2. 입출력 이벤트 처리 방식

시스템에서 입출력 작업의 소요 시간은 기계적인 측면이나 발생하는 이벤트 등으로 인해 달라질 수 있어 예측하기 어려운 경우가 많다. 입출력 작업을 시작한 장치 드라이버는 입출력 작업이 종료되었거나 시간이 초과된 것을 신호로 하는 이벤트 처리 방식을 사용한다. 작업이 종료될 경우, 드라이버는 입출력 인터페이스의 상태 레지스터를 읽어 입출력 작업이 성공적으로 수행되었는지 판단한다. 입출력 작업 종료 모니터링에 사용되는 기술에는 인터럽트와 폴링이 존재한다.



[그림 2] 이벤트 처리 방식에 따른 CPU 사용 다이어그램

1) 인터럽트

인터럽트는 저장장치가 입출력 작업을 완료하면 CPU에게 신호를 보내 완료를 감지하고 처리하는 방식이다. 그림 2의 (a)는 인터럽트 동작 과정의 CPU 사용을 보여준다. 실행 중인 프로세스가 입출력 요청을 보내면 컨텍스트 스위치가 발생하며 해당 프로세스는 sleep 상태에 들어간다. 이때 실행 중

이던 프로세스의 상태는 저장되어 입출력 작업이 완료된 후에 이어서 진행될 수 있다. 입출력 작업이 완료되면 커널 내부의 인터럽트 서비스 루틴(ISR:Interrupt Service Routine)이 호출되고 sleep 상태에 들어간 프로세스가 깨어나 예약되며 저장했던 프로세스 상태를 이용해 중단된 부분부터 다시 실행된다. 인터럽트 방식은 시그널이 들어온 정확한 타이밍을 감지할 수 있고 반응 시간이 빠르며 입출력 작업이 진행되는 동안 다른 작업을 수행할 수 있다는 장점이 있지만, 입출력 작업의 소요 시간이 적을 경우, 컨텍스트 스위치 및 인터럽트 처리에 의한 오버헤드가 크게 발생할 수 있다.

2) 폴링

폴링은 CPU가 주기적으로 저장장치의 상태를 확인하여 입출력 작업의 완료 여부를 확인하는 방식이다. 그림 2의 (b)는 폴링 동작 과정의 CPU 사용 다이어그램이다. 입출력 작업이 완료되었다는 값이 표시될 때까지 CPU가 디바이스의 상태 레지스터를 점검한다. 구현이 쉽고, 우선순위 변경이 용이하다는 장점이 있지만, 시스템 자원을 많이 사용하고 폴링 주기에 따른 오차가 존재한다는 단점이 있다. 폴링 방식은 CPU가 입출력 작업이 완료될 때까지 대기하면서 CPU 사이클을 낭비하기 때문에 시간이 오래 걸리는 입출력 작업에서는 효율적이지 않다[15].

최근 몇 마이크로초 만에 입출력 작업을 완료하는 초저지연 저장장치의 도입으로 인터럽트 방식보다 폴링 방식의 입출력 이벤트 처리를 사용하는 것이 입출력 작업의 오버헤드를 완화하는데 효과적이라고 여겨지고 있다[4, 5, 28].

3. 비동기적 입출력 인터페이스

저장장치의 성능이 높아짐에 따라 입출력 처리에 발생하는 소프트웨어 오버헤드를 줄이기 위한 새로운 입출력 인터페이스들이 등장하고 있다. 기존의 인터럽트 기반 동기적 입출력 인터페이스의 경우 저장장치가 입출력 요청을 처리하는 동안 스레드가 블록 되기 때문에 별도의 입출력 스레드 풀을 생성하여 스레드가 블록 되더라도 다른 입출력 요청을 처리할 수 있도록 하는 방식을 사용하였다. 그러나, 이러한 접근 방식은 스레드 간 많은 컨텍스트 스위칭을 발생시켜 CPU 자원을 비효율적으로 사용하며 이는 고성능 저장장치의 성능을 모두 활용하지 못하도록 만든다.

위와 같은 동기적 입출력 인터페이스의 한계로 고성능 입출력 처리를 필요로 하는 상황에서는 입출력 처리를 기다릴 필요 없이 다른 작업을 처리할 수 있는 비동기적 입출력 인터페이스를 활용한다. [표 1]은 논문에서 사용하는 입출력 인터페이스별 특징을 보여준다.

[표 1]
입출력 인터페이스 특징

SW overhead	libaio	SPDK
System call	2 per I/O batch	0
Memory copy	O	X
Context switch	O	X
Interrupts	O	X

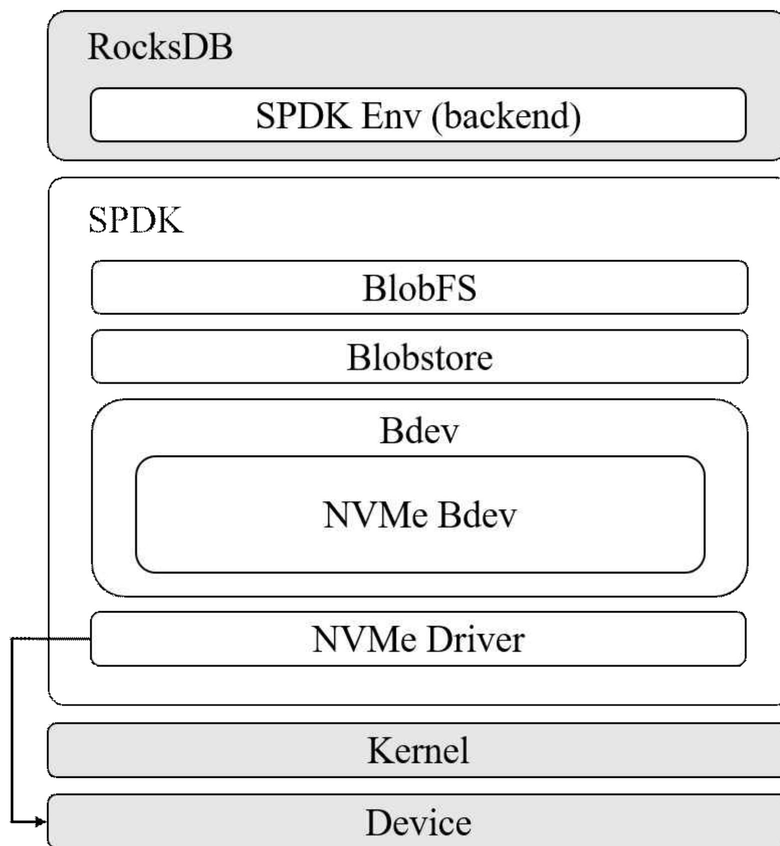
1) libaio

libaio(Linux Asynchronous I/O)[10]는 리눅스 커널 프로젝트의 일부로 비동기 입출력을 지원하기 위해 개발된 인터페이스이며 입출력 요청 후 컨텍스트 스위치를 발생시켜 다른 작업을 수행하다가 인터럽트가 발생하면 입출력 요청을 마저 처리하는 방식을 사용한다. libaio를 사용하면 응용 프로그램이 모든 블록 장치(HDD, SATA SSD, NVMe SSD)와 비동기 방식으로 상호작용할 수 있다. libaio는 io_submit과 io_getevents, 두 가지의 주요 시스템 콜을 중심으로 동작한다. io_submit 시스템 콜을 통해 입출력 요청을 커널에 제출하고 io_getevents 시스템 콜을 통해 완료된 입출력 요청을 검색한다. 기존의 블로킹 입출력 API에 비해 사용 편의성과 유연성이 높다는 이점이 존재하지만, 입출력 작업 당 두 개의 시스템 콜 호출을 필요로 하고 유저와 커널 간 메타데이터 전달을 위해 시스템 콜마다 메모리 복사가 필요하여 입출력 작업 당 성능 오버헤드가 발생한다는 단점이 존재한다. 이 외에도 페이지 캐시를 사용하지 않는 direct I/O인 O_DIRECT만 사용 가능하다는 한계점을 가지고 있어 활용률이 낮다[8, 16, 17].

2) SPDK

SPDK(Storage Performance Development Kit)는 인텔에서 제시한 유저 영역에서 사용하는 스토리지 입출력 가속 솔루션으로 zero-copy, 높은 성능, NVMe SSD에 대한 직접적인 접근을 제공한다[11]. 블록 장치 드라이버를 유저 영역에 구현함으로써 컨텍스트 스위치와 시스템 콜 호출 오버헤드를 제거하였다. 또한, 하드웨어 자원에 대한 Lock을 제거하여 동기화에 사용되는 오버헤드를 줄였다. 이와 같은 특징으로 인해 SPDK는 높은 확장성을 가지고 있다. 저장장치의 PCIe 레지스터는 디바이스와 응용 프로그램 간에 공유되는 제출 큐(SQ:Submission Queue)와 완료 큐(CQ:Completion Queue)를 구성하기 위해 유저 공간에 매핑된다. 입출력 요청은 SQ에 제출되고 입출

력 요청의 완료는 CQ에서 폴링 되어 인터럽트나 시스템 콜이 발생하지 않는다. SPDK는 커널에 의존하지 않기 때문에 폴링 기반의 입출력 완료 처리만 사용할 수 있고 인터럽트 기반의 입출력 완료 처리는 불가능하다. 또한, libaio와 비교했을 때 복잡성이 높고 커널을 우회하기 때문에 커널에서 지원하는 파일시스템과 커널 스토리지 서비스의 기능을 이용할 수 없다는 단점이 있다. 이러한 단점에도 불구하고 현재 SPDK는 워크로드의 최고 성능을 제공할 수 있는 최첨단 입출력 스택으로 여겨지며 많은 프로젝트에서 광범위하게 사용되고 있다[23, 24, 25, 26].



[그림 3] SPDK RocksDB 계층

[그림 3]은 SPDK가 적용된 RocksDB의 계층 구조로 응용 프로그램과 SPDK의 구성을 나타낸다. SPDK는 커널을 우회하여 커널 파일 시스템을 지원하지 않기 때문에 Blobstore 위에서 동작하는 간단한 유저 수준 파일시스템인 BlobFS를 제공한다. Blobstore는 로컬 스토리지 시스템으로 기존 파일 시스템이 담당하던 블록 할당 역할을 하며 스토리지 서비스를 효율적으로 동작시키기 위해 설계되었다. Blobstore에서 데이터는 'Blob'이라는 수 MB 크기의 큰 단위로 관리되기 때문에 NVMe 저장장치에 SPDK를 사용하기 위해서는 Hugepage 할당이 필수적이다. Bdev(Block Device Abstraction)는 블록 장치를 추상화하여 고정 크기의 논리 블록으로 read/write가 가능하도록 블록 I/O 인터페이스를 제공하는 추상화 계층이다. SPDK에서는 NVMe, malloc(ramdisk), Linux AIO, virtio-scsi, Ceph RBD 등 다양한 유형의 블록 장치 드라이버를 위한 인터페이스를 제공하고 있다 [7].

SPDK는 장치를 제어하는 드라이버를 유저 영역으로 옮겨서 사용한다. 이를 위해 저장장치의 기존 드라이버를 unbind 시키고 유저 영역 드라이버인 uio[41]나 vfio[42]를 사용해 커널에서 해당 장치를 분리한다. uio와 vfio 유저 영역 드라이버는 저장장치의 PCI BAR(Base Address Register)를 프로세스에 매핑하여 저장장치에 접근할 수 있도록 하는 역할을 한다.

III. 관련 연구

1. 고성능 저장장치를 위한 입출력 스택 최적화 연구

소프트웨어 계층의 오버헤드로 인해 고성능 저장장치의 성능을 최대로 활용할 수 없게 되면서, 기존 선행 연구들은 커널 입출력 스택의 재설계나 새로운 유저 레벨 입출력 스택의 개발을 통해 소프트웨어 오버헤드를 줄이고자 하였다.

AIOS[32]는 초저지연 SSD를 위해 설계된 저지연 커널 I/O 스택인 비동기 I/O 스택으로 비동기 연산을 활용하여 입출력 관련 CPU 작업을 줄이고 전반적인 성능을 향상시키는 스토리지 I/O 스택 설계에 대한 새로운 접근 방식을 제안하였다. AIOS는 I/O 경로에서의 동기 작업을 비동기 작업으로 대체하여 read 및 fsync와 관련된 CPU 작업을 장치 I/O와 중첩시키는 방식을 사용했다. Iris[40]는 control plane과 data plane을 분리하여 일반적인 I/O 경로에서 시스템 소프트웨어의 오버헤드를 최소화하는 응용 프로그램용 새로운 I/O 경로를 제시하였다. 이 외에도 I/O 스택과 I/O 경로에 대한 재설계를 통해 고성능 저장장치의 성능을 향상시키는 방향의 연구들이 이루어져 왔다[28, 38, 39].

SPDK와 같이 커널을 우회하는 방식의 유저 레벨 입출력도 많은 관심을 받고 있다. NVMeDirect[43]는 NVMe SSD에서 응용 프로그램별 최적화를 지원하는 유저 공간 I/O 프레임워크를 제안하였다. NVMeDirect는 응용 프로그램과 NVMe SSD 간의 직접적인 접근을 지원함으로써 커널 I/O 스택의 오버헤드를 방지하기 때문에 커널 I/O보다 높은 처리량을 달성할 수 있다.

저장장치의 입출력 지연시간 감소를 위한 방법으로 스토리지 인터페이스를 변경하는 방법도 존재한다. Vasudevan[37] 등은 고성능 네트워크 시스템

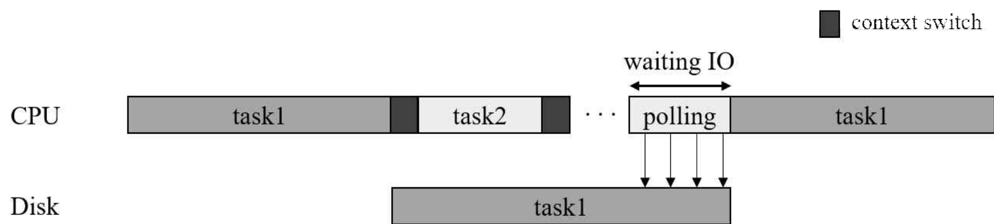
에서 여러 요청을 통합하여 처리하는 벡터 인터페이스를 제안하였다. 벡터 인터페이스는 단일 key-value 스토리지 서버에서의 지연시간을 14배 이상 단축시켰다. Son[36] 등은 빠른 저장장치의 특성 활용을 위한 블록 기반 파일 시스템 최적화를 제안하여 기존 파일 시스템보다 평균 32% 향상된 성능을 달성하였다. 이러한 연구들은 여러 I/O 요청을 하나의 명령어로 통합하여 스토리지 접근에서 round trip 횟수를 줄이는 방법을 사용해 입출력 성능을 향상시킨다.

FastResponse[35]는 클라우드 데이터 센터에서 지연시간에 민감한 워크로드에 초저지연 SSD를 사용하는 접근 방식의 I/O 스케줄러를 제안하였다. Intel Optane SSD와 같은 초저지연 SSD는 매우 낮은 지연시간을 제공할 수 있지만, I/O 스택을 통한 다양한 워크로드 간의 간섭은 여전히 지연시간을 크게 늘릴 수 있으며 이러한 간섭은 주로 SSD 장치의 자원 경합, 파일 시스템의 트랜잭션 커밋, 비용이 높은 프로세스 스케줄링에서 비롯한다는 사실을 밝혔다. FastResponse는 새로운 I/O 스케줄러를 이용해 리눅스와 비교하여 평균 응답 시간을 18~70% 단축시켰지만 처리량은 6% 감소되었다.

Shin[33] 등은 SCM(Storage-Class Memory)을 디스크 드라이브 및 확장 가능한 메인 메모리로 사용하기 위해 소프트웨어 스토리지 스택 최적화와 새로운 입출력 인터페이스에 관해 탐구하였다. 불필요한 폴링을 방지하고 스토리지 시스템 버스의 부담을 줄이기 위해 동적 인터벌 폴링이라는 새로운 폴링 방식을 제안하고 저장장치와 호스트 OS 간의 파이프라인 실행을 제안하여 전체 성능을 15% 향상시켰다.

기존의 인터럽트와 폴링 입출력 처리 방식의 장점을 이용하고 단점을 보완시키기 위해 폴링 중간에 타이머 기반 sleep을 삽입하는 하이브리드 폴링 방식이 등장했다[18]. 하이브리드 폴링은 폴링이 시작되기 전에 예상되는 입출력 작업 지연시간의 절반만큼 sleep 하여 기존의 클래식 폴링보다 CPU를

적게 사용하는 것을 목표로 한다[4]. [그림 4]는 하이브리드 폴링의 CPU 사용 다이어그램이다. 기존 클래식 폴링은 입출력 작업 실행시간 동안 CPU 사이클을 낭비하지만, 하이브리드 폴링은 입출력 작업 실행시간의 절반을 sleep 하며 다른 프로세스가 CPU 자원을 활용할 수 있도록 하기 때문에 클래식 폴링에 비해 CPU 사이클의 낭비가 줄어든다. 하이브리드 폴링은 리눅스 커널 4.10 버전에 추가되었다[27].



[그림 4] 하이브리드 폴링 CPU 사용 다이어그램

하이브리드 폴링은 스레드의 sleep 시간을 결정하는 옵션으로 fixed-time과 adaptive를 제공한다. fixed-time 하이브리드 폴링은 사용자로부터 직접 시간을 받고 adaptive 하이브리드 폴링은 입출력 요청에 대한 장치 시간을 측정하여 얻은 지연시간 데이터로 sleep 시간을 동적으로 적용한다[18].

두 개 레이어의 NVM 캐시를 사용하여 DRAM 캐시를 줄이는 MyNVM[29]의 설계 옵션으로 하이브리드 폴링 디자인이 사용되었고, 초저지연 저장장치를 위해 하이브리드 폴링에서 장치 시간 통계를 큐 단위로 관리하여 sleep 시간을 더 정확하게 추정할 수 있게 하는 최적화 연구도 등장하였다[5]. 그러나 다양한 크기의 데이터가 존재하는 실제 시스템에서는 실행되는 입출력 작업의 크기나 실행시간을 항상 정확하게 예측할 수 없어, 여전히 일부 입출력 요청이 설정된 sleep 시간 종료 이전에 도착할 위험이 있다.

이처럼 대부분의 기존 고성능 저장장치 성능 향상을 위한 연구는 단일 저장장치의 성능을 끌어올리는 데 집중하고 있다. 그러나 이러한 방식은 스토리지 노드에 여러 개의 저장장치가 연결되어 있어 CPU 자원이 부족한 분산 시스템 환경에서는 과도한 CPU 사용량으로 인해 오히려 전체적인 시스템의 성능 저하를 불러올 수 있다. 따라서 CPU 사용량이 높은 분산 시스템 환경을 위한 입출력 인터페이스에 관한 추가적인 연구가 필요하다.

2. 고성능 저장장치에서 입출력 인터페이스에 따른 성능 분석 연구

저장장치의 성능 향상과 그에 따른 새로운 입출력 인터페이스의 등장들로 인해 초저지연 고성능 저장장치에서의 입출력 인터페이스에 따른 성능 분석이 활발히 이루어지고 있다[1, 4, 8, 30, 31].

Koh[1] 등은 800GB의 초저지연 SSD 프로토타입을 사용하여 다양한 대기열 및 접근 패턴과 같은 광범위한 조건들을 고려하여 초저지연 SSD의 시스템을 분석하고 동작을 파악하였다. 또한, 리눅스 커널의 폴링 및 하이브리드 폴링 입출력 방식의 효율성과 직면한 문제들을 분석하고 이를 기존의 인터럽트 기반 I/O 경로의 효율성과 비교하였다. 인터럽트 기반 및 폴링 기반 I/O 서비스가 제공하는 읽기, 쓰기의 지연시간 차이는 각각 평균 2.2%, 11.2% 미만으로 나타났다. 입출력 성능 측면에서는 초저지연 저장장치에서 폴링 기반 입출력을 사용하는 것이 유리하지만, 높은 CPU 사이클 소비와 잦은 메모리 접근과 같은 시스템 수준의 오버헤드로 인해 CPU stall이 자주 발생하고 시스템의 전력 소비가 증가한다는 단점 또한 존재한다는 결과를 얻었다.

Harris[4] 등은 리눅스 커널의 폴링과 하이브리드 폴링, 인터럽트 입출력 방식에 대한 성능 분석을 제시하였다. 폴링은 인터럽트보다 성능적인 면에서 우수하지만, CPU 사용에 더 높은 비용을 필요로 하며 입출력 지연시간이 낮은 요청에 대해서는 폴링 방식의 입출력이 가장 에너지 효율적이라는 결과를 제공한다.

Didona[8] 등은 리눅스에서 쉽게 사용할 수 있는 최신 스토리지 API의 성능 특성에 초점을 맞추어 지연시간, 처리량, 확장성 등을 평가하여 체계적인 분석을 제공하며 최신 스토리지 API를 사용하는 고성능 응용 프로그램에 대한 설계 가이드라인을 제시하였다.

Ren[30] 등은 초저지연 저장장치인 Intel Optane SSD 환경에서 리눅스의 다양한 I/O 스택과 API에 대한 성능 분석을 진행하였다. 폴링 방식의 입출력 처리는 인터럽트 방식과 비교하여 성능을 1.7배 향상시켰지만, 2.3배의 CPU 명령을 소비하며 높은 부하와 규모에서 io_uring[17]은 SPDK보다 10배 이상 느려질 수 있음을 확인하였다.

앞선 고성능 저장장치에서의 성능 분석 연구의 결과를 종합하면 다음과 같은 결과를 얻을 수 있다.

(1) 고성능 저장장치의 경우 인터럽트 방식의 입출력 처리보다 폴링 방식의 입출력 처리를 사용하는 것이 더 유리하다.
(2) 폴링 방식은 성능을 향상시키지만 CPU 명령과 자원을 많이 소비한다.
(3) 최신 스토리지 API 중 SPDK가 가장 좋은 성능을 제공한다.

기존의 분석 연구들은 일반적으로 CPU 자원이 부족하지 않은 상황을 기준으로 진행되었으나, 본 논문에서는 여러 스토리지 노드가 CPU 자원을 공유하는 분산 스토리지 시스템 환경에서 CPU 자원이 부족한 상황을 고려하여 CPU 경쟁 상황을 모사하였다는 점에서 차별점을 가진다.

IV. 성능 분석

본 장에서는 많은 CPU 사용량과 그에 따른 CPU 자원 경쟁 상황이 발생하는 분산 스토리지 시스템 환경을 모방하여 CPU 자원 경쟁 상황 발생 시 입출력 인터페이스별 성능 변화를 비교 분석하고자 한다. 실험은 마이크로 벤치마크와 매크로 벤치마크로 나누어 진행했다.

1. 마이크로 벤치마크

1) 실험 환경

[표 2]
입출력 벤치마크 실행 환경

OS	Ubuntu 20.04
Kernel	Linux 5.4.0-128-generic
CPU	Intel Xeon Gold 6342 2.8Gz
Memory	128GB
fio	v3.29
SPDK	v18.07

[표 2]와 같은 환경에서 마이크로 벤치마크 실험을 진행했다. 실험에는 2개의 NUMA node를 가진 Intel Xeon Gold 6342 CPU를 사용했으며 각 NUMA 노드는 24개의 물리 코어를 가진다. 실험에 사용할 저장장치로는 PCIe 3.0 기반의 A와 PCIe 4.0 기반의 B, 두 가지의 NVMe SSD를 사용했다. [표 3]은 PCIe 3.0 기반 저장장치 A와 PCIe 4.0 기반 저장장치 B의 4KB Random Read 성능을 나타낸다.

[표 3]
NVMe SSD 4KB Random Read 성능

	A(PCIe 3.0)	B(PCIe 4.0)
IOPS	420,000	1,500,000
latency	15µs	8µs

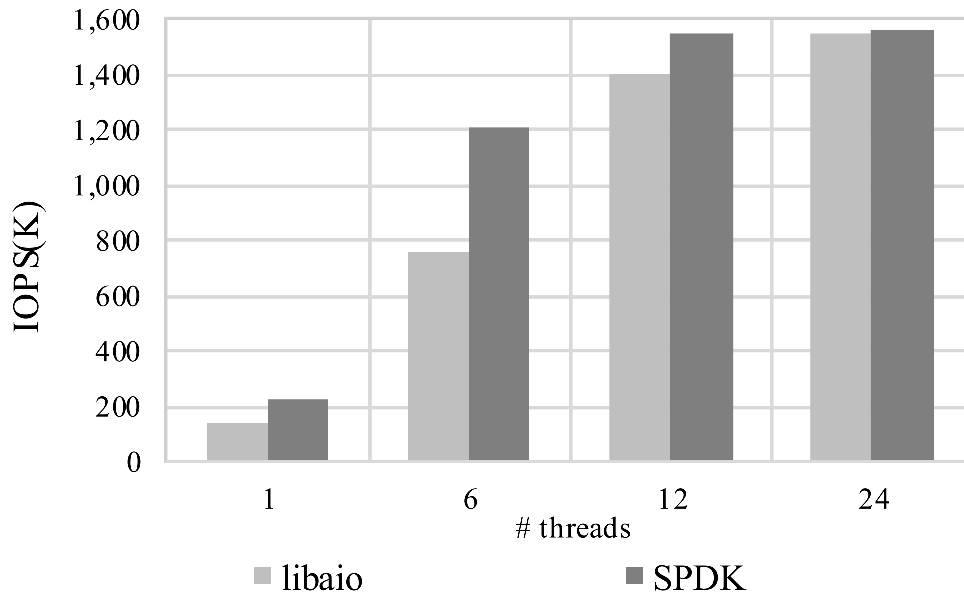
2) 실험 방법

저장장치의 성능 특성이 CPU 경쟁 상황 발생 시 입출력 인터페이스에 따른 성능 변화에 영향을 미치는지 확인하고자, [표 3]의 입출력 성능이 다른 두 종류의 NVMe SSD에서 CPU 경쟁 상황을 모사한 실험을 진행하였다. 입출력 인터페이스로는 libaio와 SPDK를 사용하였으며, 입출력 벤치마킹 도구로는 입출력 처리 방식에 따른 성능 비교를 위해 인터럽트 기반의 libaio와 폴링 기반의 SPDK를 모두 지원하는 FIO(Flexible I/O Tester)[12]를 사용했다. FIO는 유연한 I/O 워크로드 생성이 가능하고 오버헤드가 낮은 I/O 경로를 사용한다는 장점을 가지고 있다. 이 실험에서 FIO는 1GB 크기의 파일에 대해 4KB Random Read를 수행한다.

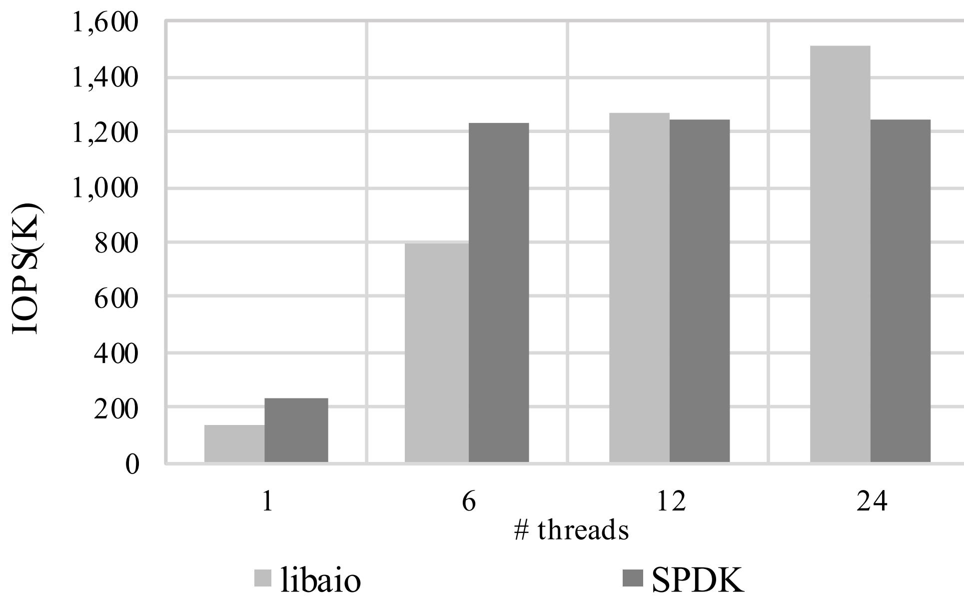
여러 개의 저장장치가 CPU 코어를 함께 사용해야 하는 분산 스토리지 시스템 환경에서는 하나의 저장장치가 전체 CPU 코어를 독점적으로 사용할 수 없으며 CPU 코어를 100% 입출력 작업에만 사용하는 것은 거의 불가능하다. 이러한 분산 스토리지 시스템의 특성을 모방하고자 CPU 코어 수를 제한하고 그 위에서 CPU 경쟁 상황을 조성하고자 했다. 저장장치의 최대 성능을 달성시킬 수 있을 만큼 CPU 자원에 여유가 있는 상황에서는 CPU 경쟁에 따른 성능 저하가 발생하지 않기 때문에 폴링 기반의 SPDK가 [표 3]에 서술된 두 NVMe SSD 모두에서 저장장치의 최대 입출력 성능을 달성하지 못하도록 코어 수를 6개로 제한하여 실험을 진행하였다.

CPU 코어를 6개로 제한한 이유는 다음과 같다. [그림 5]는 표 2에 B로 서술된 PCIe 4.0 기반의 NVMe SSD에서의 Random Read IOPS를 보여준다. [그림 6]은 [그림 5]와 동일한 환경에서 CPU 코어를 전체 24개 중 6개로 제한하여 측정한 결과이다. 그래프의 x축은 스레드 수를 의미하며 실행 시 I/O depth는 1로 고정하였다. CPU 자원이 여유로운 [그림 5]와 같은 상황에서는 폴링 기반의 SPDK가 인터럽트 기반 libaio 보다 먼저 저장 장치의 최대 성능인 150만 IOPS를 달성하지만, [그림 6]와 같이 CPU 자원이 부족한 상황에서는 폴링 기반 인터페이스인 SPDK가 저장 장치의 최대 성능을 달성하지 못한다. CPU 코어를 독점적으로 사용하는 폴링을 사용하는 SPDK에서는 입출력 작업을 수행하는 스레드 간의 CPU 경쟁이 발생하기 때문이다.

입출력 작업과 애플리케이션이 제한된 CPU 코어 위에서 자원을 두고 경쟁하는 상황을 모사하기 위해 입출력 작업으로는 FIO를, 애플리케이션으로는 SPEC CPU 2017[13] 벤치마크에서 제공하는 스케줄링 프로그램인 mcf_r 워크로드를 사용한다. 6개의 코어 위에 입출력 작업(FIO) 스레드 6개, 애플리케이션(SPEC CPU) 스레드 6개를 동시에 실행시켜 분산 스토리지 시스템에서 발생하는 CPU 자원 경쟁 상황을 모사하였다. 실험 진행 시에는 결괏값의 변동을 줄이기 위해 Intel Turbo Boost를 비활성화하였고, CPU Governor를 Performance로 설정했다. 또한 CPU의 NUMA 구조가 성능에 미치는 영향을 막기 위해 실험은 1개의 NUMA node에 고정해 진행하였으며, 이상치로 인한 영향을 제거하기 위해 각 실험은 5번씩 진행하여 평균값을 결괏값으로 사용하였다.

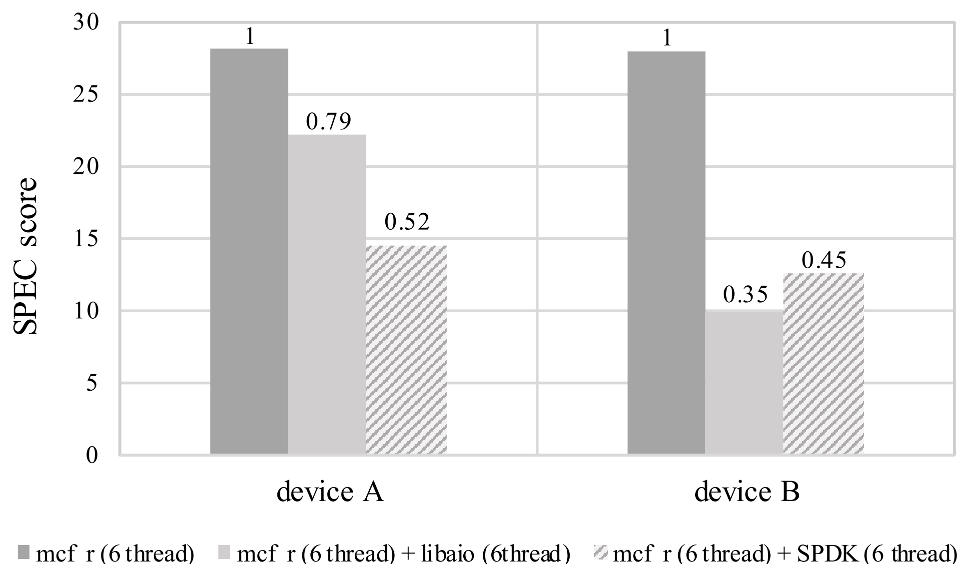


[그림 5] PCIe 4.0 기반 NVMe SSD에서의 Random Read IOPS



[그림 6] CPU 코어 제한 시 PCIe 4.0 기반 NVMe SSD에서의 Random Read IOPS

3) 실험 결과

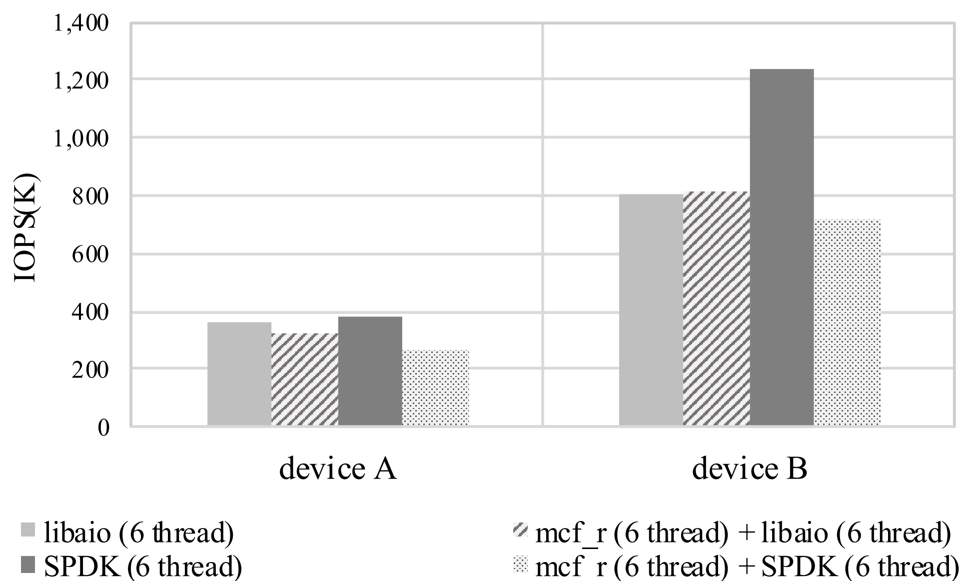


[그림 7] 저장장치에 따른 입출력 인터페이스별 애플리케이션 성능 변화

[그림 7]은 CPU 코어 6개 위에서 CPU 경쟁 상황 없이 SPEC CPU mcf_r 벤치마크 6개 스레드를 단독으로 실행했을 때와 대비해, SPEC CPU mcf_r 스레드 6개와 FIO 스레드 6개를 동시에 실행시켰을 때 SPEC CPU 성능인 SPEC score의 성능 저하를 보인다. Y축은 실제 측정된 SPEC score 값이며, 그래프에 표시된 데이터 레이블은 mcf_r 단독 실행 시 SPEC score를 기준으로 정규화한 값을 나타낸다.

PCIe 3.0 기반의 A 저장장치에서 애플리케이션 성능은 입출력 인터페이스로 인터럽트 기반의 libaio 사용 시 21%, 폴링 기반의 SPDK 사용 시 48% 저하되어 폴링 기반의 SPDK와 함께 실행했을 때 더 심각한 성능 저하를 보였다. 그러나 초저지연 저장장치인 PCIe 4.0 기반의 B 저장장치에서의 애플리케이션 성능은 인터럽트 기반의 libaio와 함께 실행했을 때 65%, 폴링

기반의 SPDK와 함께 실행했을 때 55% 저하되어, 인터럽트 기반의 libaio를 사용하는 입출력 작업과 병행할 때 더 많은 성능 저하가 나타났다. 이는 SPDK와 SPEC CPU mcf_r을 동시에 실행시켰을 때, 입출력 작업보다 애플리케이션 작업에 더 많은 CPU 자원이 사용되었기 때문으로 추정된다.

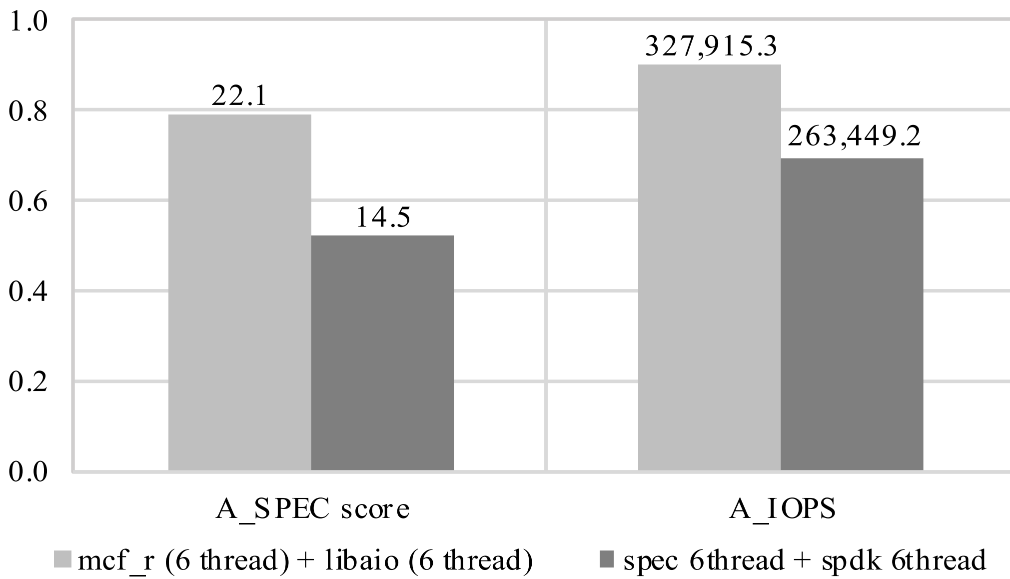


[그림 8] 저장장치에 따른 입출력 인터페이스별 입출력 성능 변화

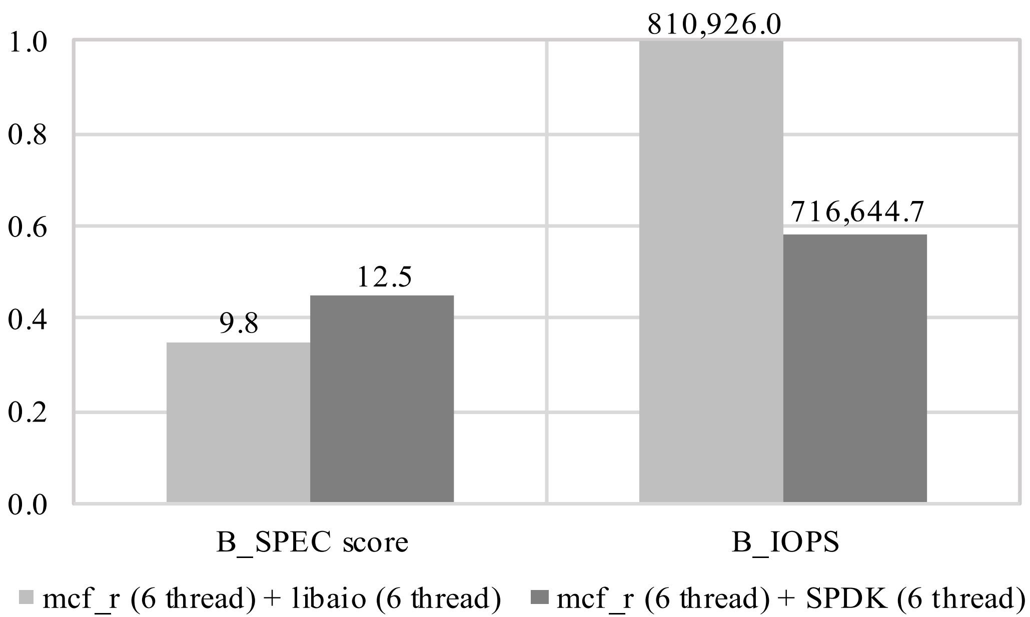
[그림 8]은 CPU 코어 6개 위에서 CPU 경쟁 상황 없이 SPEC CPU mcf_r 벤치마크 6개 스레드를 단독으로 실행했을 때에 대비해, SPEC CPU mcf_r 스레드 6개와 FIO 스레드 6개를 동시에 실행시켰을 때 입출력 작업의 성능 저하를 보인다. Y축은 실제 측정된 입출력 처리량인 IOPS(Input/Output Operations Per Second) 값이다.

CPU 경쟁 상황 발생 시 입출력 작업 단독 실행 성능 대비 입출력 성능은 PCIe 3.0 기반의 A 저장장치에서 입출력 인터페이스로 인터럽트 기반의 libaio 사용 시 10%, 폴링 기반의 SPDK 사용 시 31% 저하되었고, PCIe 4.0

기반의 B 저장장치에서는 입출력 인터페이스로 인터럽트 기반의 libaio 사용 시 성능 저하가 나타나지 않았지만, 폴링 기반의 SPDK 사용 시 42% 저하되었다. CPU 경쟁 상황에서 입출력 성능은 PCIe 3.0 기반 저장장치 A와 PCIe 4.0 기반 저장장치 B 모두에서 폴링 기반의 SPDK 인터페이스를 사용했을 때 더 많이 저하되었다.



[그림 9] PCIe 3.0 기반 NVMe SSD에서 CPU 경쟁 상황 시 입출력 인터페이스별 성능



[그림 10] PCIe 4.0 기반 NVMe SSD에서 CPU 경쟁 상황 시 입출력 인터페이스별 성능

[그림 9]와 [그림 10]은 저장장치의 성능 특성에 따른 성능 변화를 알아보기 용이하도록 [그림 7]과 [그림 8]의 결과를 저장장치의 종류에 따라 재구성한 결과이다. Y축은 각각 저장장치 A와 B에서 애플리케이션 작업인 SPEC CPU mcf_r 벤치마크 스레드 6개를 단독 실행했을 때와 입출력 작업인 FIO 벤치마크 스레드 6개를 단독 실행했을 때 성능을 기준으로 하여, 입출력 작업 스레드 6개와 애플리케이션 작업 스레드 6개를 함께 실행하였을 때 각 작업의 성능을 정규화한 값이다. 그래프에 표시된 데이터 레이블은 실제 측정된 결괏값을 나타낸다.

[그림 9]의 입출력 지연시간이 상대적으로 긴 PCIe 3.0 기반의 A 저장장치의 경우, SPEC CPU 벤치마크 성능인 SPEC score는 단독으로 실행했을 때와 비교하여 libaio와 병행 시 21%, SPDK와 병행 시 48%의 성능 저하가 발생하였다. 입출력 처리량은 입출력 인터페이스로 libaio 사용 시 10%, SPDK 사용 시 31% 저하되었다. 입출력 작업과 애플리케이션 작업 간의 CPU 경쟁 상황이 발생했을 때 애플리케이션 성능과 입출력 성능 모두 인터럽트 기반 입출력 방식인 liabio보다 폴링 기반 입출력 방식인 SPDK에서 더 심각한 성능 저하가 발생하였다.

[그림 10]의 PCIe 4.0 기반의 초저지연 저장장치 B의 경우, SPEC CPU 벤치마크 성능인 SPEC score는 단독으로 실행했을 때와 비교하여 libaio와 병행 시 65%, SPDK와 병행 시 55%의 성능 저하가 발생하여 폴링 기반의 SPDK를 사용했을 때, 인터럽트 기반의 libaio를 사용했을 때에 비해 애플리케이션 성능 저하율이 약 10% 적게 나타났다. 그러나 입출력 처리량 측면에서는 여전히 폴링 기반의 SPDK 사용 시 성능이 42% 감소하며 더 심각한 성능 저하가 발생하였다.

애플리케이션 성능의 경우, PCIe 3.0 기반 저장장치에서는 SPDK와 함께 실행시켰을 때 성능 저하가 심했지만, PCIe 4.0 기반 저장장치에서는 더 적

은 성능 저하를 보였다. 그러나 입출력 처리량 측면에서는 PCIe 3.0 기반 저장장치와 PCIe 4.0 기반 저장장치에서 공통으로 애플리케이션 작업과 libaio를 사용한 입출력 작업을 병행했을 때보다 SPDK를 사용한 입출력 작업과 병행했을 때 더 많이 감소하였다.

PCIe 3.0 기반의 저장장치 A에서는 입출력 작업을 단독 실행했을 때 libaio와 SPDK의 성능 차이가 거의 발생하지 않으므로 CPU 경쟁 상황에서 성능 저하가 적은 인터럽트 방식을 사용하는 것이 좋다. 그러나 PCIe 4.0 기반의 저장장치 B에서는 입출력 작업의 단독 실행 성능이 libaio보다 SPDK에서 50% 이상 좋기 때문에 CPU 경쟁이 없는 상황에서는 폴링을 사용하는 것이 훨씬 효율적이다.

2. 매크로 벤치마크

1) 실험 환경

[표 5]와 같은 환경에서 매크로 벤치마크를 통해 실제 데이터베이스 상에서의 워크로드를 통한 성능 분석을 진행하였다. 데이터베이스로는 SPDK가 적용된 RocksDB[34]를 사용했으며 데이터베이스 성능 평가 지표로 페이스북에서 제공하는 Mixgraph 벤치마크[19]를 사용하였다. RocksDB는 페이스북에서 개발된 임베디드 key-value 스토어이며, 대규모 분산 시스템을 대상으로 하고 SSD와 같은 빠른 저장장치에 최적화되어있다. RocksDB는 다양한 분야에서 응용되고 있으며 논문에서 목표로 하는 분산 스토리지인 Ceph에서도 RocksDB를 사용하고 있다.

[표 4]
Mixgraph 워크로드 실행 환경

OS	Ubuntu 20.04
Kernel	Linux 6.0.0
CPU	Intel Xeon Gold 6342 2.8Gz
Memory	128GB
RocksDB	6.15.fb

Mixgraph는 실제 key-value 스토어 워크로드에 대한 분석을 통해 쿼리 구성, 키와 밸류의 사이즈, 핫 데이터의 지역성 등을 고려하여 YCSB 벤치마크보다 실제 워크로드에 가깝게 만들어진 벤치마크이다. Key range 기반 모델링을 통해 실제 replay와 가장 유사한 결과를 제공하는 Prefix_dist 워크로드를 사용했으며 그 구성은 [표 4]와 같다[19]. SPDK의 캐시와 커널의

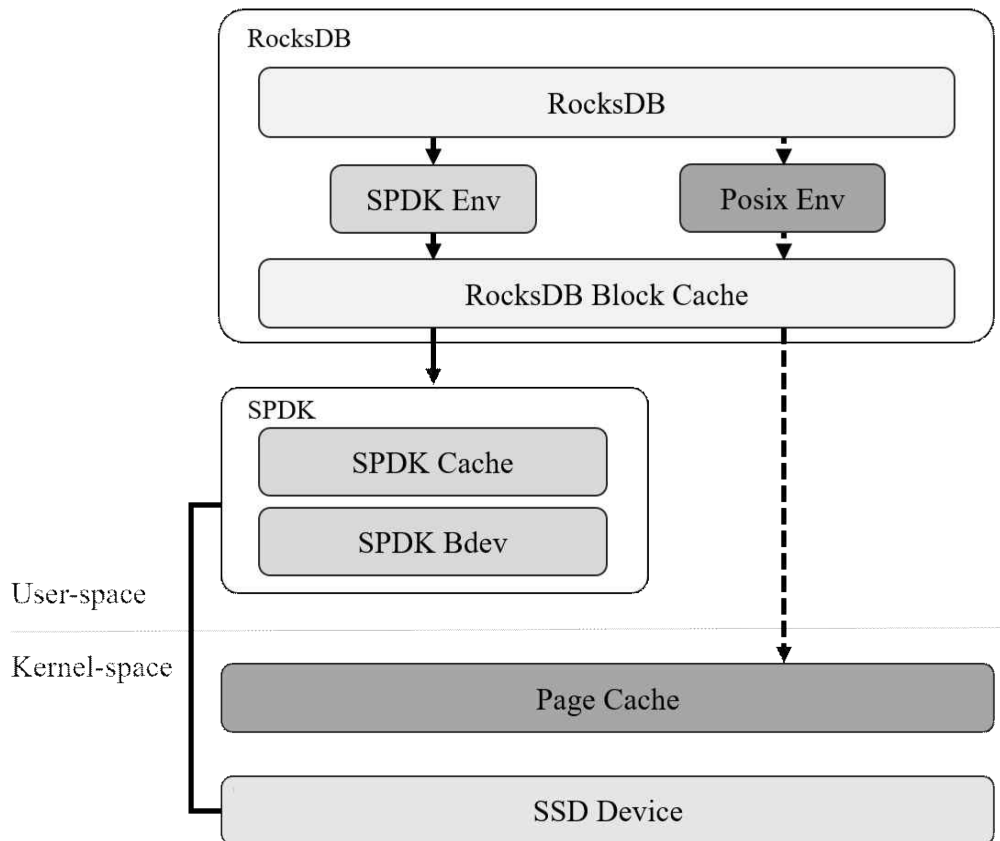
페이지 캐시를 활용하기 위해 Prefix_dist의 설정 중 direct I/O와 관련된 옵션은 해제하였으며 워크로드를 실행하기 전 5천만 개의 key-value 쌍을 데이터베이스에 임의의 순서로 삽입하였다.

[표 5]
Mixgraph 워크로드 구성

Key size	16 byte
Value size	1000 byte
# KV pairs	50,000,000
Database size	50GB
Cache size	10GB
SPDK cache size	10GB
Page cache size	10GB
Configuration	Get 83%, Put 14%, Scan 3%

입출력 인터페이스에 따른 성능 변화를 비교하기 위해 폴링 기반 입출력 인터페이스인 SPDK를 사용하는 SPDK 환경과 리눅스 커널의 인터럽트 기반 입출력을 이용하는 Posix 환경 두 가지를 사용했다. [그림 11]은 각 실행 환경에 따라 사용하는 캐시의 종류를 보여준다. SPDK 환경과 Posix 환경은 공통적으로 RocksDB에서 read 최적화를 위해 사용되는 메모리 기반 캐시인 블록 캐시를 사용한다. Posix 환경은 커널의 페이지 캐시를 활용하지만 SPDK 환경에서는 커널을 우회하여 커널 페이지 캐시를 사용하지 못하는 대신 SPDK 블록 디바이스(Bdev) 위에 구현된 SPDK 캐시를 활용한다. 두

환경의 캐시 용량을 동일하게 제한하기 위해 리눅스의 cgroup[44]을 사용하여 Posix 환경에서의 페이지 캐시 크기를 조정해 10GB의 RocksDB 블록 캐시와 더불어 각각 10GB의 SPDK 캐시와 페이지 캐시를 사용하도록 설정하였다.



[그림 11] RockDB 실행 환경에 따른 캐시 사용

여러 개의 저장장치가 CPU 코어를 함께 사용해야 하는 분산 스토리지 시스템 환경에서는 하나의 저장장치가 전체 CPU 코어를 독점적으로 사용할 수 없다. 이러한 분산 스토리지 시스템의 특성을 모방하고자 CPU 코어를 제한하여 CPU 경쟁 상황을 조성하고자 했다. CPU 자원 경쟁 상황을 모사

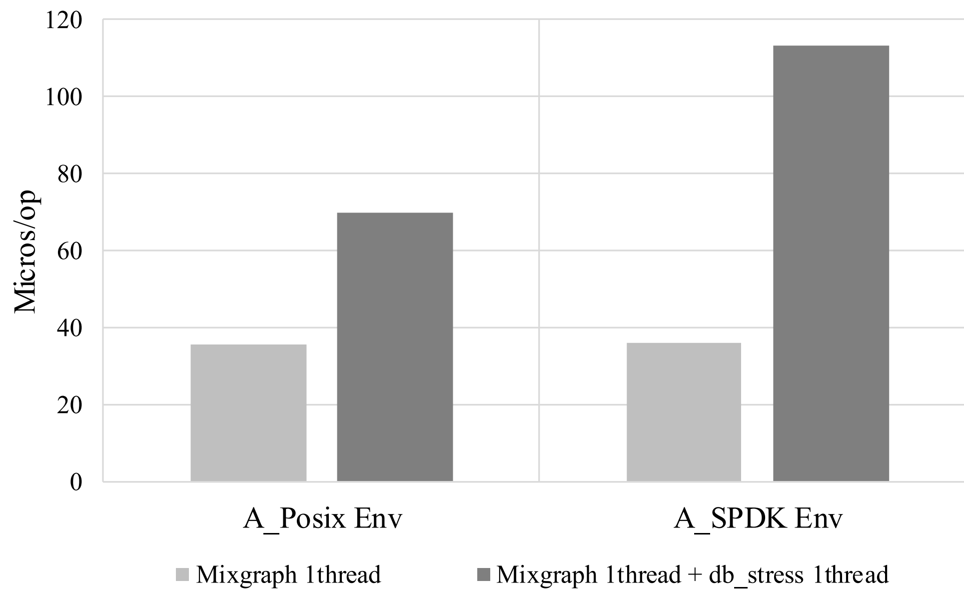
하기 위해 0번 노드의 0번 코어에서 Rocksdb Mixgraph 벤치마크 스레드 하나를 실행했고 동일한 코어에 RocksDB db_stress 스레드 하나를 실행하였다. SPDK가 적용된 RocksDB의 6.15.fb 버전의 Mixgraph에서는 멀티스레드 실행을 제공하지 않기 때문에 하나의 스레드만 사용하였다. 이상치로 인한 영향을 제거하기 위해 각 실험은 5번씩 진행하여 평균값을 결괏값으로 사용하였으며, 벤치마크 성능 평가와 마찬가지로 NUMA 구조가 성능에 미치는 영향을 막기 위해 실제 실험은 1개의 NUMA node에서 진행하였다. 결괏값의 변동을 줄이기 위해 Intel Turbo Boost를 비활성화하였고, CPU Governor를 Performance로 설정했다.

2) 저장장치 성능 특성의 영향

a) 실험 방법

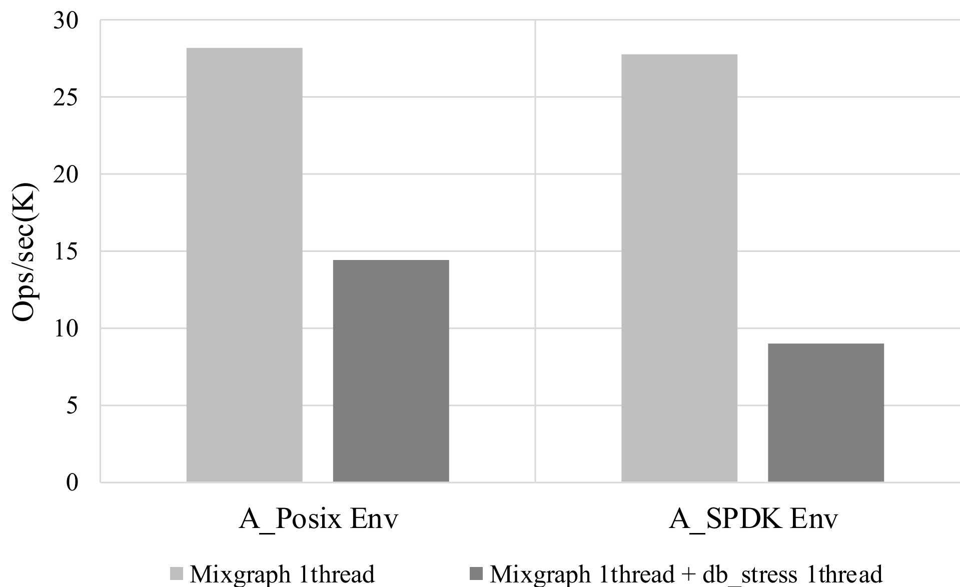
저장장치의 성능 특성에 따른 CPU 경쟁 상황 발생 시 입출력 인터페이스에 따른 성능 변화를 파악하고자, [표 3]의 입출력 성능이 다른 두 종류의 NVMe SSD에서 CPU 경쟁 상황을 모사한 실험을 진행하였다. CPU 경쟁 상황을 조성하기 위해 하나의 Mixgraph 스레드가 실행되는 코어 위에 db_stress 스레드 하나를 동시에 실행했다. Mixgraph 워크로드 설정은 [표 5]와 같다.

b) 실험 결과



[그림 12] PCIe 3.0 기반 NVMe SSD에서 데이터베이스 환경에 따른 Mixgraph 워크로드 입출력 지연시간

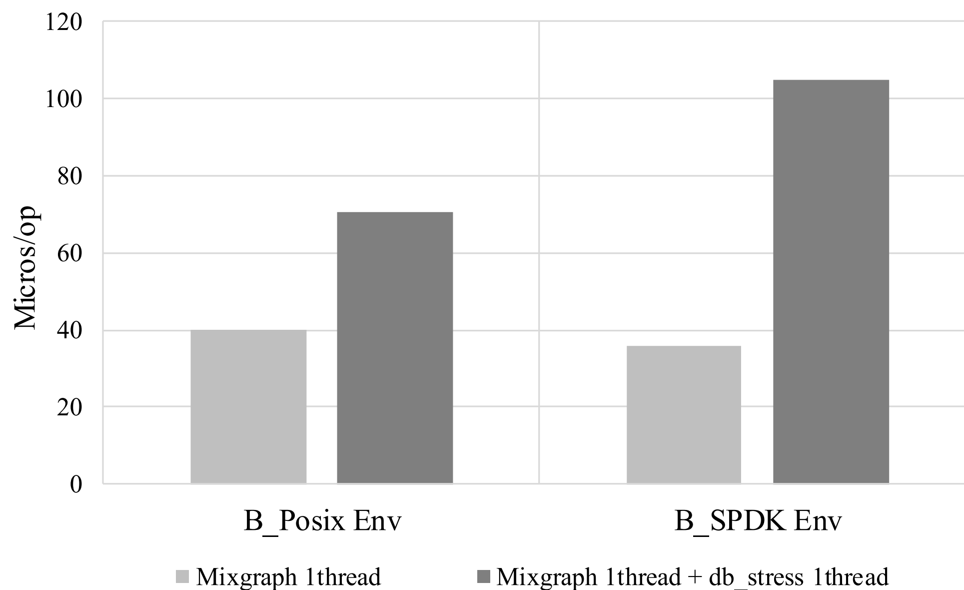
[그림 12]는 상대적으로 입출력 지연시간이 긴 PCIe 3.0 기반 NVMe SSD에서 Mixgraph 워크로드의 CPU 자원 여유 상황과 CPU 자원 경쟁 상황 시의 입출력 지연시간 변화를 나타낸다. Mixgraph 워크로드를 단독으로 실행시킨 상황에서는 폴링 기반 입출력을 사용하는 SPDK 환경과 커널의 인터럽트 기반 입출력을 사용하는 Posix 환경의 입출력 지연시간이 거의 유사하게 나타났다. 그러나 Mixgraph와 db_stress를 함께 실행시킨 CPU 자원 경쟁 상황 시, Mixgraph를 단독 실행했을 때와 비교한 입출력 지연시간이 Posix 환경에서는 97%, SPDK 환경에서는 215% 증가하며 폴링 기반 입출력을 사용하는 SPDK 환경의 입출력 지연시간이 매우 많이 증가했다.



[그림 13] PCIe 3.0 기반 NVMe SSD에서 데이터베이스 환경에 따른 Mixgraph 워크로드 처리량

[그림 13]은 PCIe 3.0 기반 NVMe SSD에서 Mixgraph 워크로드의 CPU 자원 여유 상황과 CPU 자원 경쟁 상황 시 처리량 변화를 나타낸다. 지연시

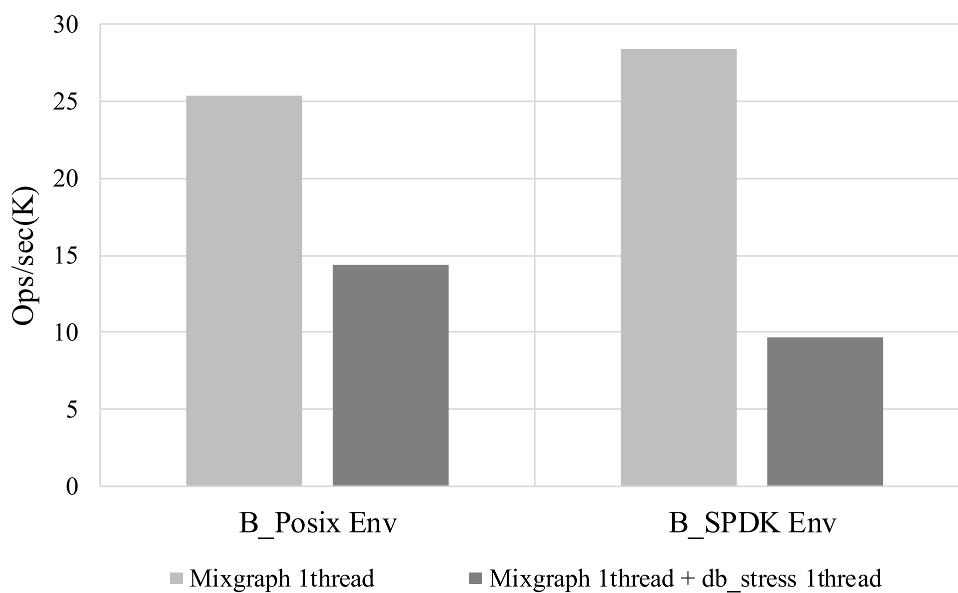
간과 마찬가지로 Mixgraph 워크로드를 단독으로 실행시킨 CPU 자원 여유 상황에서는 폴링 기반 입출력을 사용하는 SPDK 환경과 커널의 인터럽트 기반 입출력을 사용하는 Posix 환경의 입출력 지연시간이 거의 유사하게 나타났다. CPU 자원 경쟁 상황 발생 시 SPDK 환경의 처리량은 단독 실행했을 때와 대비하여 68% 감소하였고 이는 동일한 상황에서 Posix 환경의 처리량보다도 38% 적다.



[그림 14] PCIe 4.0 기반 NVMe SSD에서 데이터베이스 환경에 따른 Mixgraph 워크로드 입출력 지연시간

[그림 14]는 입출력 지연시간이 짧은 PCIe 4.0 기반 초저지연 NVMe SSD에서 Mixgraph 워크로드의 CPU 자원 여유 상황과 CPU 자원 경쟁 상황 시의 입출력 지연시간 변화를 나타낸다. Mixgraph 워크로드를 단독으로 실행시켜 CPU 자원을 독점하여 사용하는 상황에서는 폴링 기반 입출력을 사

용하는 SPDK의 입출력 지연시간이 더 낮게 나타나며 좋은 성능을 보인다. 그러나 Mixgraph와 db_stress를 함께 실행시킨 CPU 자원 경쟁 상황 시, Mixgraph를 단독 실행했을 때와 비교한 입출력 지연시간이 Posix 환경에서는 76% 증가한 데 비해 SPDK 환경에서는 194% 증가하며 2배 이상의 높은 성능 저하를 보였다.



[그림 15] PCIe 4.0 기반 NVMe SSD에서 데이터베이스 환경에 따른 Mixgraph 워크로드 입출력 처리량

[그림 15]는 PCIe 4.0 기반 NVMe SSD에서 Mixgraph 워크로드의 CPU 자원 여유 상황과 CPU 자원 경쟁 상황 시 처리량 변화를 나타낸다. 지연시간과 마찬가지로 Mixgraph 워크로드를 단독으로 실행시킨 CPU 자원 여유 상황에서는 SPDK 환경의 입출력 처리량이 높았지만 CPU 자원 경쟁 상황 발생 시 단독 실행했을 때와 대비하여 입출력 처리량이 66% 감소되며, 43% 감소된 Posix 환경보다 높은 성능 저하율을 보인다.

[그림 14]와 [그림 15]의 결과를 통해 PCIe 4.0 기반의 초저지연 저장장치를 사용할 때 CPU 자원이 여유로운 상황에서는 폴링 기반 입출력을 사용하는 SPDK 환경의 성능이 우수하지만, CPU 경쟁 상황에서는 지연시간과 처리량 모두 인터럽트 기반 입출력을 사용하는 Posix 환경보다 폴링 기반 입출력을 사용하는 SPDK 환경에서 더 심각한 성능 저하가 발생됨을 확인하였다.

PCIe 3.0 기반의 저장장치 A에서는 CPU 경쟁 없이 Mixgraph만 단독 실행했을 때 SPDK 환경과 Posix 환경의 성능 차이가 거의 발생하지 않기 때문에 CPU 경쟁 상황에서 성능 저하가 적은 인터럽트 기반의 입출력 방식을 사용하는 것이 좋다. 그러나 PCIe 4.0 기반의 저장장치 B에서는 폴링 기반 입출력을 사용하는 SPDK 환경에서의 Mixgraph의 단독 실행 성능이 인터럽트 기반 입출력을 사용하는 Posix 환경보다 우수하다. CPU 경쟁 상황에서는 폴링의 높은 성능 저하율로 인해 Posix 환경이 SPDK 환경보다 높은 성능을 가진다. 따라서 고성능 저장장치에서는 CPU 자원이 여유로운 상황에는 폴링을, CPU 자원 경쟁이 발생하는 상황에서는 인터럽트를 사용하는 것이 효율적이다.

3) CPU 간섭 정도의 영향

a) 실험 방법

Mixgraph 벤치마크 스레드가 받는 db_stress의 CPU 사용률을 조정하여 고성능 저장장치에서 CPU 간섭 정도가 성능 변화에 미치는 영향을 파악하고자 했다. 고성능 저장장치에서의 영향을 분석하기 위해 저장장치로는 [표 3]의 PCIe 4.0 기반 NVMe SSD인 B 저장장치를 사용하였다. CPU 경쟁 상황을 조성하기 위해 하나의 Mixgraph 스레드가 실행되는 코어 위에 db_stress 스레드 하나를 동시에 실행시켰으며 리눅스의 cpulimit을 통해 최대 CPU 자원 사용량을 제한하였다. Mixgraph의 CPU 사용량은 제한하지 않고 db_stress의 CPU 간섭 정도를 10%부터 50%까지 10% 단위로 조정하였다. 또한, 성능 저하가 시작되는 지점을 파악하기 위해 10% 미만의 CPU 간섭 실험을 추가로 진행하였다. Mixgraph 벤치마크와 db_stress를 함께 실행했을 때 대부분 db_stress가 CPU를 50% 이상 사용하지 않기 때문에 60% 이상의 CPU 간섭은 제외하였다. CPU 간섭을 제외한 나머지 조건은 [표 4]의 구성과 같다.

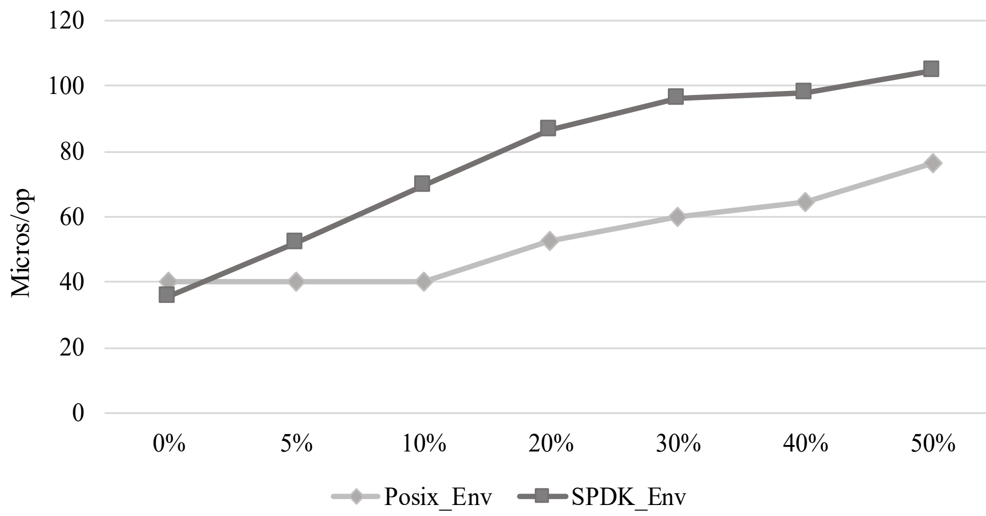
b) 실험 결과

[그림 16]은 Mixgraph 벤치마크 스레드가 받는 db_stress의 CPU 간섭 정도에 따른 영향을 나타낸다. [그림 16]의 (a)는 작업 당 지연시간, (b)는 초당 작업량인 처리량으로 1000단위로 표현된다.

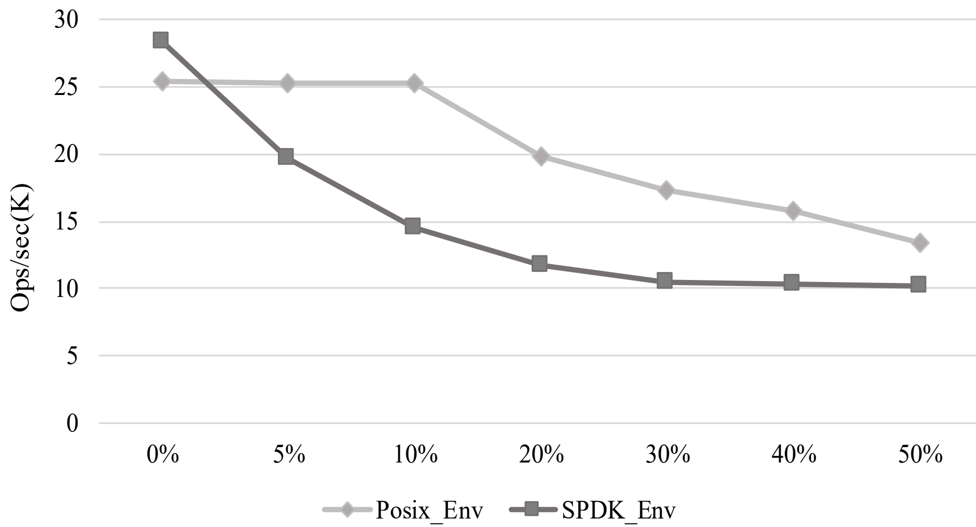
db_stress가 간섭하지 않아 CPU 경쟁이 없는 환경에서는 폴링 방식의 입출력을 사용하는 SPDK 환경이 더 낮은 지연시간과 더 높은 처리량을 달성하며 좋은 성능을 보였다. 그러나 CPU 경쟁이 발생하면서 SPDK 환경의 성능은 Posix 환경의 성능보다 낮아졌다. 지연시간과 처리량 모두 db_stress

의 CPU 사용량을 10%로 제한하였을 때 Posix 환경에서는 1% 미만의 성능 저하가 발생하지만, SPDK 환경에서는 약 50%의 성능 저하가 발생한다. CPU 간섭 정도가 높아지면서 두 환경 간의 성능 차이는 점차 줄어들었다.

CPU 간섭 정도가 5%일 때까지는 Posix 환경과 SPDK 환경의 처리량이 약 20% 차이로 감당할 수 있을 정도의 성능 저하가 나타났지만, 간섭 정도가 5%를 넘어가면서부터 두 환경 간의 처리량 차이가 40% 이상으로 크게 벌어졌다. 따라서 CPU 자원을 95% 이상 활용할 수 있는 환경에서는 폴링 방식의 입출력을, 활용할 수 있는 CPU 자원이 95% 미만인 환경에서는 인터럽트 방식의 입출력을 사용하는 것이 좋을 것으로 예상된다.



(a) 지연시간



(b) 처리량

[그림 16] CPU 간섭 정도에 따른 성능 변화

4) 캐시 크기 대비 데이터셋 크기 비율의 영향

a) 실험 방법

캐시 크기 대비 데이터셋 크기 비율이 CPU 경쟁 상황에서의 성능 변화에 미치는 영향을 파악하고자 10GB의 캐시를 사용할 때 데이터셋의 크기를 20GB, 50GB, 100GB로 조정하여 실험을 진행하였다. 고성능 저장장치에서의 영향을 분석하기 위해 저장장치로는 [표 3]의 PCIe 4.0 기반 NVMe SSD인 B 저장장치를 사용하였다. 워크로드를 실행하기 전 데이터베이스에 삽입하는 key-value 쌍의 개수를 변경을 통해 데이터셋 크기를 조정하였으며 CPU 경쟁 상황을 조성하기 위해 하나의 코어 위에서 Mixgraph 스레드와 db_stress 스레드를 동시에 실행하였다. 데이터베이스 크기를 제외한 나머지 조건은 [표 5]와 같다.

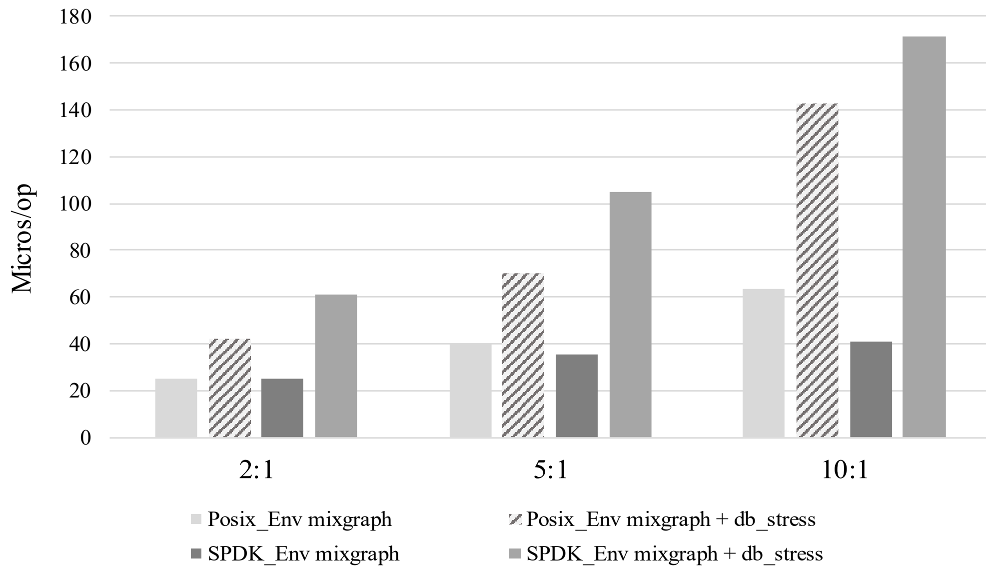
b) 실험 결과

[그림 17]은 데이터베이스 크기와 캐시 크기의 비율에 따른 영향을 나타낸다. [그림 17]의 (a)는 작업 당 지연시간, (b)는 초당 작업량인 처리량으로 1000단위로 표현된다. CPU 자원이 여유로운 상황에서는 Posix 환경보다 SPDK 환경의 성능이 높았으며, 두 환경 모두 캐시 크기에 비해 데이터 크기가 커질수록 성능이 저하되었다. 그러나 db_stress를 동일한 코어에 동시에 실행시켜 CPU 자원 경쟁 상황이 발생하자 SPDK 환경에서의 성능 저하가 심각하게 발생하여 지연시간과 처리량 모두 Posix 환경보다 낮은 성능을 보였다.

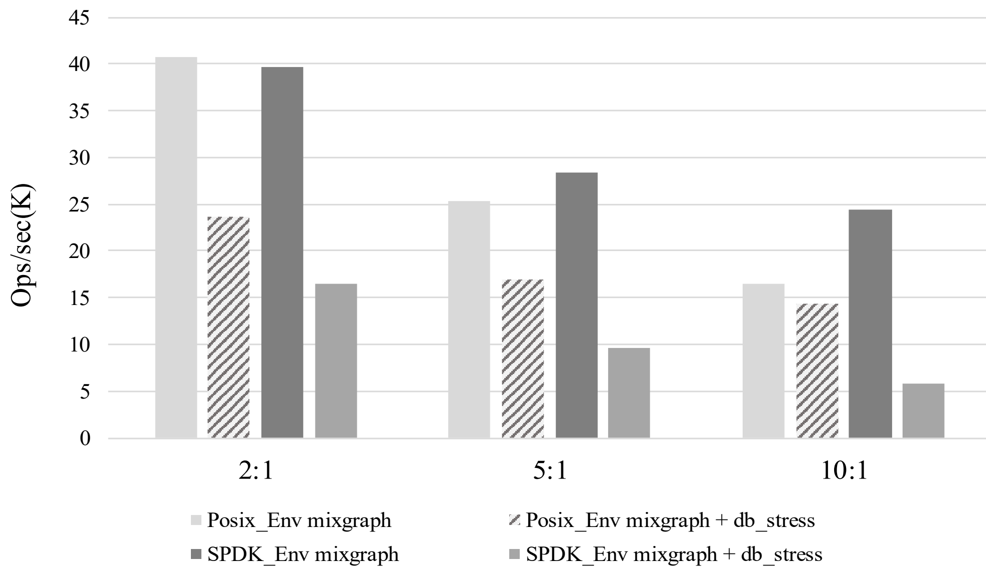
Mixgraph 단독실행 시 데이터베이스 크기와 캐시의 비율이 2:1인 상황에서는 SPDK와 Posix 환경의 성능이 거의 비슷했지만 5:1로 증가했을 때는 SPDK 환경의 성능이 Posix 환경에서의 성능보다 12%, 10:1로 증가했을 때

는 48% 높았다. 데이터베이스 크기가 증가함에 따라 캐시에서 처리할 수 있는 요청이 줄어들기 때문에 폴링 기반 입출력 처리의 성능적인 이점이 나타난다. CPU 자원이 여유로운 상황에서는 확장성이 좋은 폴링 방식을 사용하는 것이 좋지만, CPU 경쟁이 빈번하게 발생하는 환경에서는 인터럽트 방식을 사용하는 것이 좋다.

Mixgrpah와 db_stress를 같은 코어에 실행시킨 CPU 경쟁 상황에서는 캐시 크기 대비 데이터셋 크기가 커지면서 성능 저하율도 함께 증가했다. 특히 SPDK 환경에서 지연시간은 최대 318% 증가했고 처리량은 최대 76% 감소했다. 캐시 크기 대비 데이터셋의 크기의 크기가 클수록 CPU 경쟁으로 인한 성능 저하는 심각해지고 이러한 성능 저하는 인터럽트 기반 입출력을 사용하는 Posix 환경보다 폴링 기반 입출력을 사용하는 SPDK에서 더 심화된다. 고성능 저장장치에서 CPU 자원에 제한이 없는 상황에는 데이터셋의 크기가 클수록 입출력 성능이 좋은 폴링 방식을 사용하는 것이 좋다. 그러나 CPU 자원이 부족한 상황에서는 폴링 방식의 성능이 인터럽트보다 낮아지기 때문에, 인터럽트 방식을 사용하는 것이 좋다.



(a) 지연시간



(b) 처리량

[그림 17] 데이터베이스와 캐시 크기 비율에 따른 성능 변화

Ⅳ. 결론 및 향후연구

고성능 저장장치의 성능을 이용하기 위해서는 폴링 방식의 입출력이 효율적이라고 알려져 있다. 하지만 폴링 방식은 CPU 자원의 독점적 이용을 필요로 하기 때문에, CPU 자원이 부족한 상황에서는 폴링 방식이 인터럽트 방식보다 더 높은 성능을 보장한다고 확신할 수 없다. 하나의 스토리지 노드가 여러 저장장치를 관리하여 CPU 사용률이 높은 분산 시스템 환경에서는 CPU 자원의 효율적인 활용이 매우 중요한 요소이다.

이에 본 논문에서는 CPU 자원 경쟁 상황에서 최적의 입출력 처리 방식을 모색하기 위해 CPU 자원이 부족한 상황을 모사하여 성능 변화를 분석하였다. PCIe 3.0 기반 NVMe SSD에서는 폴링과 인터럽트 간 성능 차이가 거의 발생하지 않기 때문에, CPU 경쟁 상황에서 성능 저하율이 낮은 인터럽트를 사용하는 것이 좋다. 그러나 PCIe 4.0 기반의 고성능 저장장치에서는 CPU 자원이 여유로운 상황에서 폴링의 입출력 성능이 인터럽트보다 좋기 때문에 폴링의 특성을 활용할 수 있도록 CPU 경쟁 상황을 고려한 입출력 메커니즘이 필요하다. 폴링 기반 입출력 처리를 사용하는 SPDK는 CPU 코어에 대한 작은 간섭에도 민감하게 반응하기 때문에 CPU 간섭이 5% 이상 발생할 때는 인터럽트를, 그렇지 않으면 폴링을 사용하는 것이 좋다. CPU 자원이 여유로운 상황에서 폴링 방식은 좋은 확장성을 가지지만, CPU 부하가 높은 상황에서는 성능 저하가 크게 발생한다. 따라서 CPU 부하가 높은 상황에서는 인터럽트 방식을 사용하는 것이 좋다.

이러한 결과를 통해 CPU 코어를 입출력 작업에 100% 사용하지 못하는 경우 폴링 기반 입출력 인터페이스로부터 기대하는 성능을 얻지 못할 수 있으며, 고성능 저장장치에서 폴링 기반의 입출력 처리 방식이 항상 정답은 아닐 수 있다는 것을 확인했다. CPU 자원이 여유로운 상황에서는 고성능

저장장치의 저지연 특성을 활용할 수 있는 폴링 방식의 입출력을 사용하는 것이 효율적이지만, CPU 자원이 부족한 상황에서는 성능 저하로 인해 인터럽트 방식을 사용했을 때보다 성능이 낮아질 수 있다. 따라서 저지연 특성을 활용하는 폴링의 이점을 이용하면서 CPU 자원이 부족한 상황을 고려할 수 있도록 고성능 저장장치를 위한 새로운 입출력 메커니즘이 필요하다.

향후 연구로는 본 연구의 성능 분석 결과를 토대로 현재의 CPU 상황과 다양한 입출력 조건을 고려하여 유저 영역 폴링과 커널 영역 인터럽트를 선택적으로 지원하는 하이브리드 입출력 인터페이스 개발에 관한 연구를 진행할 예정이다.

참고문헌

[1] S. Koh, J. Jang, C. Lee, M. Kwon, J. Zhang, and M. Jung, Faster than flash: An in-depth study of system challenges for emerging ultra-low latency ssds, 2019 IEEE International Symposium on Workload Characterization (IISWC), pp.216 - 227, IEEE, 2019.

[2] L. Paul and V. Vishal, Spdk blobstore: look inside the nvm optim ized allocator, https://www.snia.org/educational-library/spdk-blobs tore-look-inside-nvm-optimized－allocator-2017, accessed Dec 02, 2023.

[3] Ultra-low latency with samsung Z-NAND SSD, https://www.sa msung.com/semiconductor/global.semi.static/Ultra-Low_Latency_wi th_Samsung_Z-NAND_SSD-0.pdf, accessed Dec 02, 2023.

[4] B. Harris and N. Altiparmak, When poll is more energy efficient than interrupt, 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage 22), pp.59 - 64, 2022.

[5] G. Lee, S. Shin, and J. Jeong, Efficient hybrid polling for ul-tra-low latency storage devices, Journal of Systems Architecture, vol.122, p.102338, 2022.

[6] S. Koh, C. Lee, M. Kwon, and M. Jung, Exploring system chal-lenges of ultra-low latency solid state drives, 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18), 2018.

[7] Z. Yang, J.R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L.E. Paul, Spdk: A development kit to build high performance storage applications, 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp.154 - 161, IEEE, 2017.

[8] D. Didona, J. Pfefferle, N. Ioannou, B. Metzler, and A. Trivedi, Understanding modern storage apis: a systematic study of libaio, spdk, and io uring, Proceedings of the 15th ACM International Conference on Systems and Storage, pp.120 - 127, 2022.

[9] M. Oh, J. Park, S.K. Park, A. Choi, J. Lee, J.H. Choi, and H.Y. Yeom, Re-architecting distributed block storage system for improving random write performance, 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), pp.104 - 114, IEEE, 2021.

[10] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, Asynchronous i/o support in linux 2.5, Proceedings of the Linux Symposium, pp.371 - 386, Citeseer, 2003.

[11] Intel, What is spdk, https://spdk.io/doc/about.html, accessed Dec 02, 2023.

[12] J. Axboe, fio - flexible i/o tester rev. 3.29, https://fio.readthedocs.io/en/latest/fio_doc.html, accessed Dec 02, 2023.

[13] J. Bucek, K. Lange, and J. Kistowski, Spec cpu2017: Next-generation compute benchmark, Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, pp.41 - 42, 2018.

[14] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, Ceph: A scalable, high-performance distributed file system, Proceedings of the 7th symposium on Operating systems design and implementation, pp.307 - 320, 2006.

[15] D. Bovet and M. Cesati, Understanding The Linux Kernel, Oreilly & Associates Inc, 2005.

[16] V. Vishal and K. John, Improved storage performance using the new linux kernel i/o interface, https://www.snia.org/educational-library/improved-storage-performance-using-new-linux-kernel-io-interface-2019, accessed Dec 02, 2023.

[17] J. Axboe, Efficient io with io_uring, https://kernel.dk/io_uring.pdf 9, accessed Dec 02, 2023.

[18] D. Moal, I/o latency optimization with polling, Vault Linux Storage and Filesystems Conference, 2017.

[19] Z. Cao, S. Dong, S. Vemuri, and D.H. Du, Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook, 18th USENIX Conference on File and Storage Technologies (FAST 20), pp.209 - 223, 2020.

[20] J. Hennessy and D. Patterson, A new golden age for computer architecture, Communications of the ACM, vol.62, no.2, pp.48 - 60, 2019.

[21] pread, https://man7.org/linux/man-pages/man2/pread.2.html, accessed Dec 02, 2023.

[22] pwrite, https://man7.org/linux/man-pages/man3/pwrite.3p.html, accessed Dec 02, 2023.

[23] K. Kourtis, N. Ioannou, and I. Koltsidas, Reaping the performance of fast NVM storage with uDepot, 17th USENIX Conference on File and Storage Technologies (FAST 19), pp.1-15, 2019.

[24] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A.C. Arpaci-Dusseau, and R.H.Arpaci-Dusseau, Scale and performance in a filesystem semi-microkernel, Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pp.819 - 835, 2021.

[25] S. Xue, S. Zhao, Q. Chen, G. Deng, Z. Liu, J. Zhang, Z. Song, T. Ma, Y. Yang, Y. Zhou, K. Niu, S. Sun, and M. Guo, Spool: Reliable virtualized nvme storage pool in public cloud infrastructure, 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp.97 - 110, 2020.

[26] X. Zhang, X. Zheng, Z. Wang, H. Yang, Y. Shen, and X. Long, High density multi-tenant bare-metal cloud, Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp.483 - 495, 2020.

[27] B. Stephen, Linux optimizations for low latency block devices, https://www.snia.org/educational-library/linux-optimizations-low-latency-block-devices-2017, accessed Dec 02, 2023.

[28] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Yeom, Os i/o path optimizations for flash solid-state drives, 2014 USENIX Annual Technical Conference (USENIX ATC 14), pp.483 - 488, 2014.

[29] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, Reducing dram footprint with nvm in facebook, Proceedings of the Thirteenth EuroSys Conference, pp.1-13, 2018.

[30] Z. Ren and A. Trivedi, Performance characterization of modern storage stacks: Posix i/o, libaio, spdk, and io uring, Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems, pp.35 - 45, 2023.

[31] J. Yang, D. Minturn, and F. Hady, When poll is better than interrupt, 18th USENIX Conference on File and Storage Technologies (FAST 12), p.3, 2012.

[32] G. Lee, S. Shin, W. Song, T. Ham, J. Lee, and J. Jeong, Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds, 2019 USENIX Annual Technical Conference (USENIX ATC 19), pp.603 - 616, 2019.

[33] D. Shin, Y. Yu, H. Kim, J. Choi, H. Yeom, et al., Dynamic interval polling and pipelined post i/o processing for low-latency storage class memory, 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13), 2013.

[34] Facebook .RocksDB. team, A persistent key-value store for fast storage environments, https://rocksdb.org, accessed Dec 03, 2023.

[35] M. Liu, H. Liu, C. Ye, X. Liao, H. Jin, Y. Zhang, R. Zheng, and L. Hu, Towards low-latency i/o services for mixed workloads using ultra-low latency ssds, Proceedings of the 36th ACM International Conference on Supercomputing, pp.1 - 12, 2022.

[36] Y. Son, H. Han, and H. Yeom, Optimizing file systems for fast storage devices, Proceedings of the 8th ACM International Systems and Storage Conference, pp.1 - 6, 2015.

[37] V. Vasudevan, M. Kaminsky, and D. Andersen, Using vector interfaces to deliver millions of iops from a networked key-value storage server, Proceedings of the Third ACM Symposium on Cloud Computing, pp.1 - 13, 2015.

[38] Y. Yu, D. Shin, W. Shin, N. Song, J. Choi, H. Kim, H. Eom, and H. Yeom, Optimizing the block i/o subsystem for fast storage devices, ACM Transactions on Computer Systems (TOCS), vol.32, no.2, pp.1 - 48, 2014.

[39] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, Linux block io: introducing multi-queue ssd access on multi-core systems, Proceedings of the 6th international systems and storage conference, pp.1 - 10, 2013.

[40] A. Papagiannis, G. Saloustros, M. Marazakis, and A. Bilas, Iris: An optimized i/o stack for low-latency storage devices, ACM SIGOPS Operating Systems Review, vol.50, no.2, pp.3 - 11, 2017.

[41] Linux userspace i/o system (uio), https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html, accessed Dec 02, 2023.

[42] Linux virtual function i/o (vfio),https://www.kernel.org/doc/Documentation/vfio.txt, accessed Dec 02, 2023.

[43] H. Kim, Y. Lee, and J. Kim, Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds, 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16), 2016.

[44] Cgroups, https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, accessed Dec 03, 2023

[45] A. Caulfield, A. De, J. Coburn, T. Mollow, R. Gupta, and S. Swanson, Moneta: A high-performance storage array architecture for next-generation, non-volatile memories, 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp.385-395, 2010.

[46] J. Liu and B. Abali, Virtualization polling engine (vpe) using dedicated cpu cores to accelerate i/o virtualization, Proceedings of the 23rd international conference on Supercomputing, pp.225 - 234, 2009.

[47] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. Kandemir, N. Kim, J. Kim, and M. Jung, Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds, 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp.477 - 492, 2018.

ABSTRACT

Performance Analysis of I/O Interfaces on High-Performance Storage Devices in a High CPU Contention Scenario

Lee SeulA
Department of Computer Science
Graduate School of
Sungshin University

To maximize the input/output performance of modern PCIe 4.0-based high-performance, ultra-low latency NVMe storage devices, it is known that using a polling-based input/output interface is preferable to an interrupt-based interface. However, the polling method for input/output processing requires exclusive use of CPU resources, which can negatively impact input/output performance in situations of CPU resource competition. Generally, distributed storage applications utilize significant CPU resources not only for input/output but also for maintaining consistency between storage nodes and for data replication. In environments with high CPU usage, using CPU resources exclusively for polling-based input/output processing may be less efficient than interrupt-based input/output processing.

In this thesis, we aim to identify the most efficient input/output method for high-performance storage devices in situations where CPU resources are limited, such as in distributed storage systems. To this end, we conducted a comparative analysis of performance changes based on the input/output methods under simulated CPU resource competition environments typical in distributed storage systems. Evaluating various factors such as storage device performance characteristics, CPU interference, and the ratio of dataset size to cache size, we found that in scenarios where multiple tasks compete for CPU resources, the performance degradation of the polling-based input/output method is significant, which may even result in lower performance than the interrupt-based method in certain situations. Based on these results, we demonstrate the necessity for a hybrid input/output mechanism that dynamically applies the optimal input/output method considering various factors in distributed storage systems, combining both polling and interrupt approaches to achieve maximum input/output performance.