학사학위 청구논문

멀티큐 SSD의 공정한 대역폭 공유를 위한 입출력 스케줄링

(I/O Scheduling for Fair Bandwidth Sharing of Multi-queue SSDs)

2017 년 12 월 11 일

숭실대학교 IT대학

정보통신전자공학부

강 성 일

학사학위 청구논문

멀티큐 SSD의 공정한 대역폭 공유를 위한 입출력 스케줄링

I/O Scheduling for Fair Bandwith Sharing of Multi-queue SSDs

지도교수 : 김 강 희

이 논문을 학사학위 논문으로 제출함

2017 년 12 월 11 일

숭실대학교 IT대학

정보통신전자공학부

강 성 일

(인준서)

강성일의 학사학위 논문을 인준함

심사위원장 노 동 건 (인)

심 사 위 원 김 강 희 (인)

2017 년 12 월 11 일

숭실대학교 IT대학

목 차

표 및 그림 목차

국문초록

멀티큐 SSD의 공정한 대역폭 공유를 위한 입출력 스케줄링

정보통신전자공학부 강 성 일
지도교수 김 강 희

4차 산업 혁명과 함께 미래의 신기술로 촉망 받고 있는 클라우드 서비스 업체들이 고성능 서비스를 위해 SSD로 서버를 구축해내고 있는 현재 시점에서, 모든 고객에게 공정한 대역폭으로 서비스를 제공해야 하는 것은 매우 중요한 문제이다. 최근에 제안된 NVMe 기반의 멀티큐 SSD는 여러 개의 코어들이 전담 큐들을 통해 병렬적으로 입출력을 수행함으로써 높은 SSD 대역폭을 제공한다. 하지만 이 멀티큐 자료구조에는 공정한 지분 스케줄링이 아직 지원되지 않고 있다. 때문에 위의 멀티큐 방식의 SSD와 같은 블록 장치에서 공정한 큐잉 기능의 도입은 필수적이다. 이 논문에서는 이를 Task level 스케줄러의 도입을 통해서 이루어 내고자 하였다. 제안하는 스케줄러는 최솟값 선택 기법을 사용하며, 각 태스크가 가중치에 비례하는 대역폭을 수신하도록 한다. 실험 결과, 제안하는 스케줄러는 멀티큐잉이 지원되는 블록 장치 환경에서 정확한 대역폭 분할 효과를 보여준다.

I. 서 론

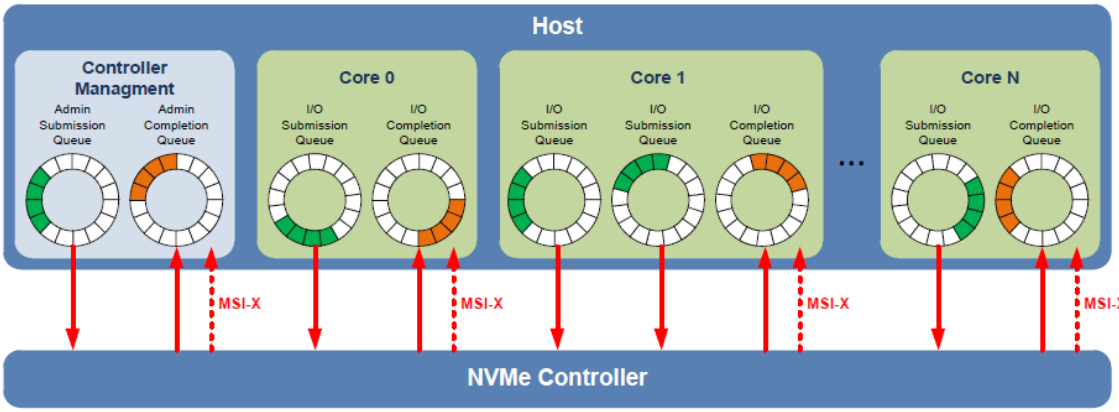
최근에 등장한 NVMe 기반 멀티큐 SSD는 기존의 단일큐 SSD에 비해 높은 대역폭을 제공한다. 단일큐 SSD는 SATA버스를 통해 AHCI 컨트롤러를 이용하는데, 이 경우 여러 개의 CPU 코어가 생성하는 입출력 요청들을 하나의 큐에 삽입하고 이것을 컨트롤러가 읽어 들여 처리한다. [표 1-1]에서 보듯이 AHCI 컨트롤러는 32개의 큐 엔트리를 가지는 명령어 큐 하나를 가지고 있으므로 높은 대역폭을 가지는 SSD를 사용하는 경우 병목 지점이 되기 쉽다.

[표 1-1] AHCI와 NVMe의 비교[1]

	AHCI	NVMe
최대 큐 깊이	하나의 명령 큐 큐 하나 당 32개의 명령	65,535개의 큐 큐 하나 당 65,536개의 명령
캐시 불가능한 레지스터 접근 수 (각각 2000 사이클)	큐에 없는 명령마다 6번 큐가 있는 명령마다 9번	명령 당 2번
MSI-X와 인터럽트 스티어링	단일 인터럽트 스티어링 없음	2048개의 MSI-X 인터럽트
병렬화 및 다중 스레드	명령 발행을 위해 동기화 락이 필요	락 없음
4KB 명령의 효율성	명령 매개변수는 2개의 직렬화된 호스트 DRAM 패치 필요	하나의 64바이트 패치의 명령 매개변수를 가져옴

반면에 멀티큐 SSD는 PCIe 버스를 통해 NVMe 컨트롤러[2]를 사용하는 경우로 CPU 코어가 생성하는 입출력 요청들은 각각의 전담 큐에 병렬적으로 삽입되고, 이것을 컨트롤러가 읽어 들여 처리한다. [표 1-1]에서 보듯이 NVMe인터페이스는 65,535개의 큐를 가지며 큐 하나 당 65,536개의 명령을 삽입할 수 있다. 이러한 큐 구조를 통해 대량의 명령어를 병렬적으로 보내고 처리할 수 있다.

호스트 운영체제에 속하는 NVMe 드라이버는 명령어를 저장하는 SQ(Submission Queue)와 완료된 명령어 정보를 저장하는 CQ(Completion Queue)를 가진다. 입출력 요청을 처리하기 위해 필요한 명령어를 SQ에 삽입하면 NVMe 컨트롤러가 이를 전달받아 처리하고, 요청이 완료된 경우 CQ를 통해 호스트 운영체제에게 요청이 완료되었음을 알린다.



[그림 1-1] NVMe 컨트롤러와 입출력 큐들[3]

[그림 1-1]에서 보듯이 이 큐들은 환형 큐 형태이며, 각 코어에 대해서는 SQ와 CQ의 쌍을 관리한다. 각 코어는 여러 개의 SQ를 둘 수 있으며 이 경우에 SQ들은 하나의 CQ를 공유하여 사용한다. 또한 각 코어는 MSI/MSI-X 인터럽트를 통해 개별적으로 입출력 완료 통지를 받는다. 여러 개의 코어들은 병렬적으로 입출력 요청들을 제출할 수 있고, 완료 통지들은 병렬적으로 통보받을 수 있다. 이러한 특징을 통해 NVMe 인터페이스는 멀티코어 환경에서 입출력 요청을 효율적으로 처리할 수 있다.

높은 대역폭을 제공하는 멀티큐 SSD가 등장함에 따라 스토리지 QoS(Quality of Service)를 제공하는 것이 갈수록 중요해지고 있다. 이에 따라 호스트 운영체제에서의 입출력 스케줄러는 스토리지 QoS를 제공하기 위한 두 가지 도전에 직면하게 되었다. 첫째, 입출력 스케줄러는 여러 개의 코어들이 멀티큐 SSD의 대역폭을 최대한 달성할

수 있도록 코어 간 성능 간섭이 발생하지 않게 스케줄링을 수행해야 한다. 둘째, 코어들이 SSD 전체 대역폭의 일정 지분을 수신할 수 있도록 공정한 대역폭 스케줄링을 수행해야 한다. 이는 각각의 코어들이 실행하는 응용 프로그램 간의 대역폭 분배를 가능하게 함으로써 각 응용 프로그램이 수행하는 입출력 성능을 보장하는 것을 의미한다.

FIOS[4]와 FlashFQ[5]와 같은 기존 연구들은 입출력 응용들 간에 공정한 대역폭을 제공하기 위해 호스트 수준 스케줄러를 제안하였다. 하지만 이러한 스케줄러들은 단일 큐 SSD를 기반으로 하고 있기 때문에 멀티큐 SSD의 대역폭을 충분히 활용하지 못한다.

최근 리눅스 운영체제는 멀티코어 환경에서 멀티큐 SSD를 효율적으로 사용하기 위해 block-mq[6]라 불리는 새로운 블록 계층을 도입하였다. 이 블록 계층은 CPU 코어와 연결된 소프트웨어 큐와 멀티큐 SSD의 큐와 연결된 하드웨어 큐를 포함한 계층으로, 소프트웨어 큐들 간의 입출력 스케줄링을 통해 QoS와 병렬성을 제공할 수 있게 되었다.

본 논문은 멀티큐 SSD를 기반으로 스토리지 QoS를 제공하는 호스트 수준 입출력 스케줄러를 제안한다. 제안하는 스케줄러는 단일큐 SSD를 위한 스케줄링 알고리즘인 FlashFQ를 확장한 스케줄러로서 멀티큐 SSD를 위한 멀티코어 확장성을 제공한다. 또한 공정 큐잉 입출력 스케줄러인 FlashFQ의 속성을 상속받아 각 코어가 공정한 대역폭 지분을 수신하는 것을 가능하게 한다. FlashFQ 알고리즘은 스케줄링 결정을 위해 모든 코어가 유지하고 있는 가상 시간(Virtual time)을 참조하고, 그 중 최소 가상 시간을 가지는 코어의 요청을 서비스한다. 스케줄링 결정을 수행하는 순간의 모든 코어의 가상 시간을 비교하기 위해서는 전역적인 잠금 변수(Global lock variable)가 필요하다. 이러한 잠금 변수는 오버헤드가 되어 성능 하락을 야기할 수 있다.

본 논문에서 제안하는 스케줄러는 Task level 스케줄러이다. 한 CPU 코어 위에 할당된 software queue로 다양한 태스크들이 공정한 대역폭을 부여 받기 위해서 도임된 스케줄러다. 이는 FlashFQ에서 사용한 알고리즘의 스케줄링 결정과 같이 최소 가상 시간을 가지는 태스크를 software queue로 할당하게 된다.

본 논문은 제안하는 스케줄러를 리눅스 운영체제에 구현하며, block-mq 계층을 포함한 커널을 대상으로 성능을 평가한다. 성능 평과 결과 제안하는 스케줄러는 코어 당 쓰레드 개수가 일정 수준 이상 많아지면 성능이 감소하지만 공정한 대역폭 지분을 제공함을 확인하였다.

II. 관련 연구

II-1. FlashFQ 스케줄러

스토리지의 QoS를 보장하기 위해 호스트 수준에서의 입출력 스케줄링을 통해 입출력 응용간의 공정한 대역폭을 제공하는 스케줄러가 제안되었다.

FlashFQ 스케줄링 알고리즘은 공정 큐잉 알고리즘을 통해 공정한 대역폭을 분배할 수 있도록 하는 SFQ(D) (Depth-controlled Start-time Fair Queuing) 알고리즘[7]을 기반으로 한다. SFQ(D) 알고리즘은 한 번에 하나의 요청을 서비스하는 SFQ 알고리즘을 확장하여 한 번에 D개의 요청을 서비스할 수 있도록 한 알고리즘이다. 즉, 매개변수 D를 제외하면 FlashFQ는 SFQ와 동일하게 동작한다.

FlashFQ는 각 입출력 스트림의 가중치에 비례하는 공정한 서비스를 제공하기 위해 모든 스트림들이 균일한 가상 시간을 유지하도록 스케줄링 한다. 이를 위해 FlashFQ는 각 스트림의 초기 가상 시간을 소위 시스템 가상 시간으로 불리는 SVT(System Virtual Time)로 설정하며, 할당된 가중치 w_i에 비례하게 각 입출력 스트림 a_i에 대한 가상 시간 VT_i를 유지한다. 가상 시간 VT_i는 해당 스트림 a_i가 수신한 서비스의 총량을 표현하며, 이 값은 스트림 a_i가 서비스를 받을 때마다 증가한다. 즉, VT_i는 스트림 a_i로부터의 요청 $r_{i,j}$가 전달(dispatch)될 때마다(또는 그 서비스가 완료될 때마다) 증가한다. 이는 수식 (1)로 표현될 수 있다.

$$VT_i = VT_i + s_{i,j}/w_i \tag{1}$$

수식 (1)에서 보듯이 VT_i는 요청 $r_{i,j}$가 수신한 저장장치 서비스 시간 $s_{i,j}$를 가중치 w_i로 나누어 얻어진 정규화된 서비스 시간만큼 증가한다. 저장장치 서비스 시간 $s_{i,j}$는 해당 요청 $r_{i,j}$가 저장장치 자원을 사용한 시간을 의미하여, 흔히 다른 요청들의 간섭이 전혀 없을 때 저장장치에 $r_{i,j}$를 전달한 시간부터 서비스 완료가 통지될 때까지의 응답 시간으로 정의하고 미리 측정되어 있다고 가정한다

[5]. 또한 위 수식에서 각 요청$r_{i,j}$에 대해 시작 태그와 마침 태그가 정의되는데, 시작 태그는 증가 전의 VT_i값이 되고, 마침 태그는 증가 후의 VT_i값이라고 정의한다. 그리고 a_i에서 새로운 요청 $r_{i,j+1}$이 도착할 때, $r_{i,j+1}$의 시작 태그는 이전 요청 $r_{i,j}$가 $r_{i,j+1}$의 도착 순간에 저장장치에서 서비스 중인지에 따라 다르게 결정된다. 즉, 만약 $r_{i,j}$가 서비스 중이라면 $r_{i,j+1}$의 시작 태그는 $r_{i,j}$의 마침 태그와 동일하게 설정된다. 즉, 만약 $r_{i,j}$가 완료되었다면 이것은 a_i가 잠시 동안 유휴(IDLE)상태를 유지하는 것을 의미하며, a_i에 대한 가상 시간 VT_i는 $r_{i,j+1}$가 도착하는 시간 t에서 관찰되는 시스템 가상 시간 $SVT(t)$로 조정된다. 따라서 $r_{i,j+1}$의 시작 태그는 $SVT(t)$로 설정된다. 시스템 가상 시간으로 조정하는 이유는 SFQ가 유휴 기간을 지내고 재가한 스트림 a_i를 시스템 상에서 처음 새롭게 시작된 스트림과 동일하게 취급하고, 따라서 VT_i를 유휴 상태에 있지 않은 다른 바쁜 스트림들의 VT_j들로 조정함으로써 그 a_i를 다른 바쁜 스트림들과 동등하게 취급하기 위함이다. 하지만 시간 t에서 관찰된 다른 모든 바쁜 스트림들의 $VT_j(t)$들이 반드시 동일하지는 않다. 왜냐하면 실제 저장장치에서는 모든 바쁜 스트림들이 매 순간 함께 서비스 중일 수는 없기 때문이다. 이 때문에, 보통 시스템 가상 시간 $SVT(t)$는 시간 t에서 관찰되는 모든 바쁜 스트림들의 $VT_j(t)$들의 최솟값으로 정의된다. 이는 수식 (2)로 나타낼 수 있다.

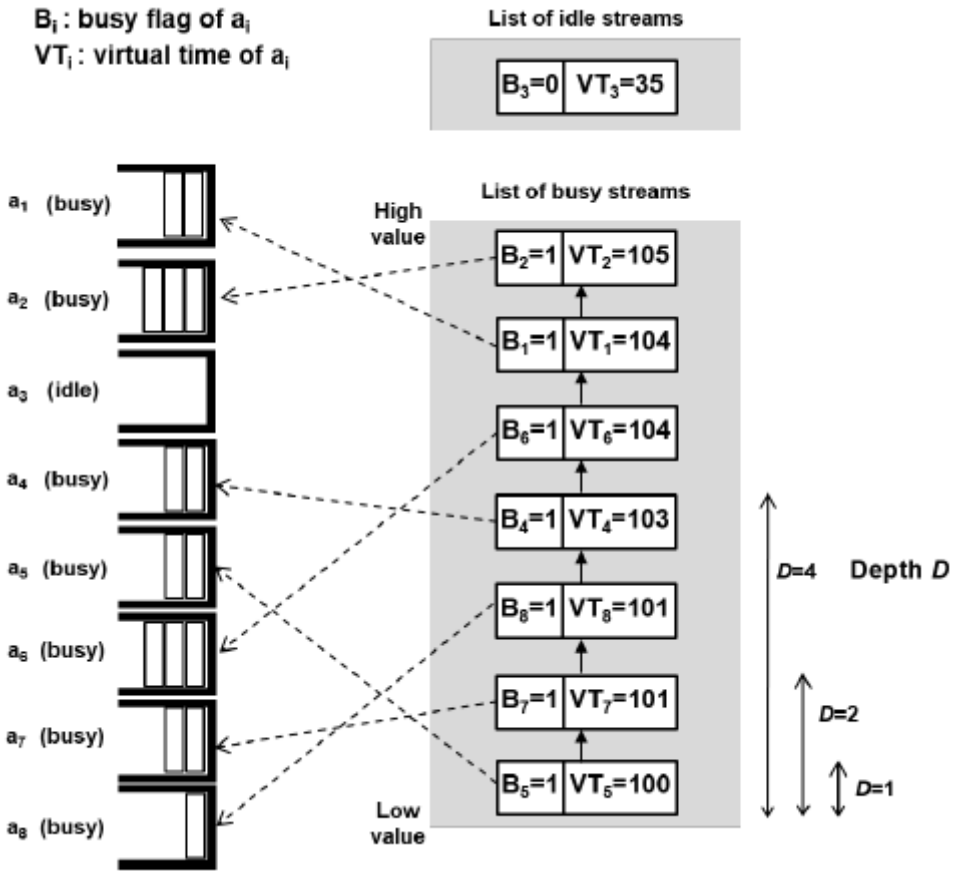
$$SVT(t) = \min\{VT_1(t), VT_2(t), ..., VT_n(t)\} \quad (2)$$

$$VT_i(t) = SVT(t) \quad (3)$$

따라서 유휴 기간을 지내고 재개한 스트림 a_i는 수식 (3)에 의해 설정된 최소 VT를 가지는 또 다른 바쁜 스트림과 동일하게 취급된다.

SFQ(D)는 SFQ가 매 순간 최소 가상 시간을 가지는 하나의 요청만을 저장장치가 서비스한다는 제약 조건을 완화시켜, 최대 D개 까지 서비스하도록 변경한 알고리즘이다. 이것은 저장장치 내부의 병렬성이 증가함에 따라 매 순간 하나의 요청만을 서비스해서는 저장장치 대역폭을 포화상태로 만들 수 없는 저장장치들이 등장했기 때문이다. 따라서 SFQ(D)는 매 순간 서비스되는

요청들의 개수를 최대 D개로 유지하기 위해서 현재 서비스 상태에 있는 D개의 요청들 중에서 종료하는 요청이 생길 때마다 최소 가상 시간을 갖는 대기 요청을 저장장치에 전달한다.



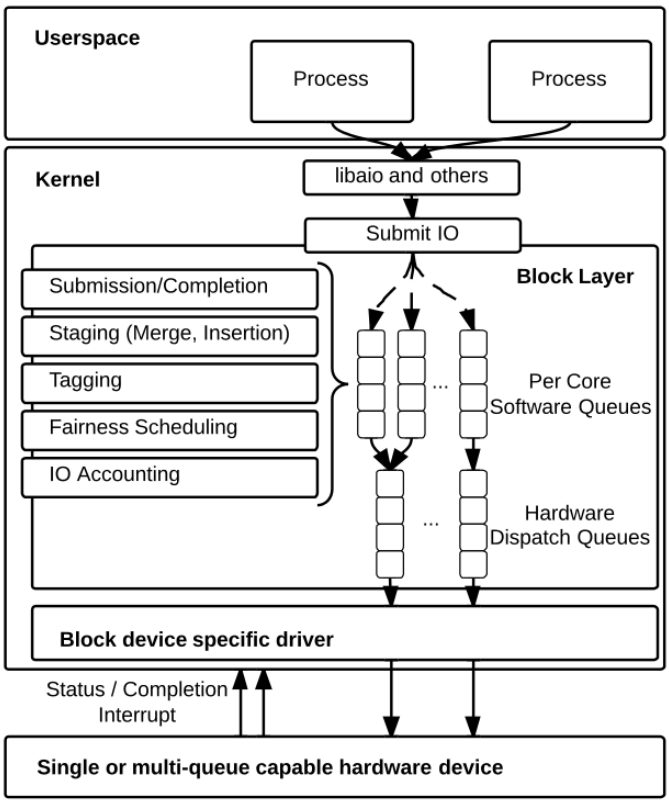
[그림 2-1] FlashFQ 스케줄링의 예[8]

[그림 2-1]은 매 순간 서비스 중인 요청들의 개수를 최대 D개로 유지하는 FlashFQ 스케줄링의 예이다. D = 4인 경우 바쁜 스트림들 중에서 가상 시간이 작은 순서대로 스트림 a_5, a_7, a_8, a_4 의 첫 번째 요청들이 저장장치에 전달되어 서비스되며, 그 중에 하나라도 서비스가 완료되면, 그 다음으로 작은 가상 시간을 가진 스트림의 첫 번째 요청이 저장장치에 전달된다.

FlashFQ는 SFQ(D)의 상기 동작에서 $SVT(t)$를 계산하는 방법에 변화를 준 것이다. 그 변화는 유휴 스트림에 도착한 새로운 요청의 시작 태그를 결정하는 것에 관한 것이다. [5]에 따르면, Min-SFQ(D)라 불리는 FlashFQ의 한 버전은 $SVT(t)$를 시간 t에서 서비스 중인 모든 요청들의 최소 시작 태그로 가정하며, 따라서 유휴 스트림에 도착한 요청들의 시작 태그를 이렇게 결정된 $SVT(t)$를 기준으로 결정한다.

II-2. Block-mq 계층

블록 계층의 단일 큐 구조는 멀티코어 환경에서 입출력 요청을 처리할 때 락 경쟁으로 인한 오버헤드로 충분한 성능을 제공하지 못한다는 문제점이 있다. 이러한 문제점을 해결하기 위해 block-mq라는 새로운 블록 계층이 리눅스 운영체제에서 제안되었다.



[그림 2-2] Block-mq의 멀티큐 구조[9]

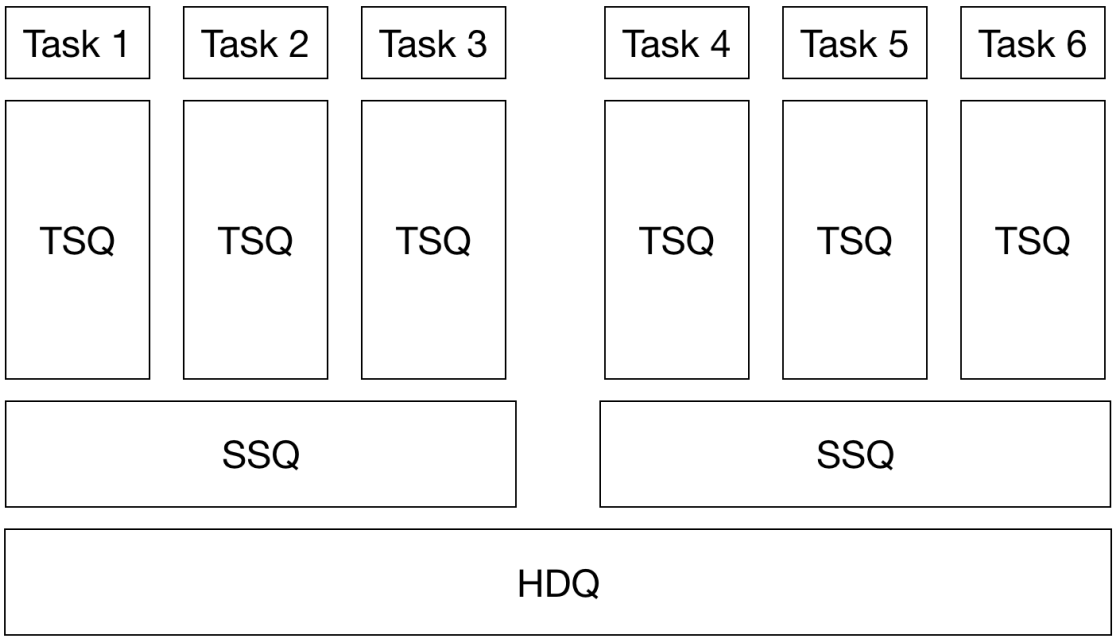
block-mq 계층은 [그림 2-2]에서 보듯이 NVMe 드라이버와 파일 시스템 사이에 존재한다. 이 블록 계층은 SSQ(Software Staging Queue)와 HDQ(Hardware Dispatch Queue)로 구성되어 있으며, 파일 시스템에서 생성된 bio 요청은 READ/WRITE 요청으로 변환된 이후 이 큐들을 차례로 거쳐서 NVMe 드라이버의 SQ에 도달하게 된다. SSQ는 CPU 코어마다 하나씩 생성되고, HDQ는 NVMe 드라이버의 SQ마다 하나씩 생성된다. HDQ는 SQ의 개수에 따라 생성되기 때문에 SSQ의 개수보다 적은 경우가 존재하며, 이 경

우에는 두 개 이상의 SSQ들이 하나의 HDQ를 공유하여 사용할 수 있다. bio 요청들이 블록 계층을 거쳐 NVMe 드라이버의 SQ에 도달하는 과정에서 이 요청들은 우선 block-mq 계층의 HDQ를 우회하여 해당 HDQ와 관련된 SQ에 전달되며, SQ에 빈 공간이 없는 경우 HDQ에 요청들을 임시로 저장한다. HDQ에 임시로 저장된 요청들은 block-mq 계층에서 주기적으로 해당 HDQ와 관련된 SQ를 확인하고 빈 공간이 있는 경우 다시 전달한다. 그러나 현 block-mq 계층에는 기존 단일 큐 구조에서 이루어지던 CPU들 간의 공정 지분 스케줄링 기능이 지원되지 않고 있다. SSD가 데이터베이스나 클라우드 서비스와 같은 광대역폭 서버에 널리 사용되고 있는 현재 시점에서, 모든 고객에게 공정한 대역폭으로 서비스를 제공해야 하는 매우 중요한 이슈이다. 때문에 NVMe기반 멀티큐 SSD의 확장성을 위해서는 공정 큐잉 기능의 도입이 필요하다고 생각하였다. 본 논문은 block-mq 계층을 포함하는 환경에서 멀티큐 SSD를 위한 새로운 Task level 입출력 스케줄러를 제안한다.

III. 제안하는 스케줄러

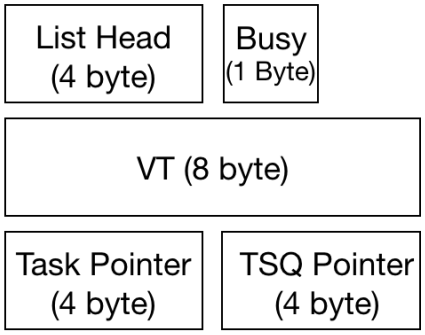
FlashFQ에서 이야기하는 SSQ와 HDQ에서 dispatch되는 과정에서의 스케줄링은 태스크 관점에서 봤을 때 한계를 나타내게된다. Task의 스케줄링으로 인한 한 코어에 작업의 쏠림 현상이 일어날 수 있고, 이런 경우 코어의 공정한 스케줄링 만으로는 Task 관점에서는 공정한 스케줄링을 받을 수 없다. 이런 한계점을 극복하기 위해서 각 코어 별로는 균등하게 request가 처리될 수 있다는 가정위에서 Task들이 동일한 하나의 CPU core위에 request dispatch에 공정한 분배가 가능하다면, task 레벨에서도 I/O 자원이 lock이 없이도 공정한 자원 분배가 가능할 것 이다.

기존 블록장치 구조에서 Task가 SSQ에 request를 넣는 방식은 [그림 2-2]에서 볼 수 있듯 Submit IO 레벨에서 처리된다. 이 부분에서 현재 CPU context를 기반으로 SSQ를 찾고 해당 SSQ에 queue가 된다. 현재 리눅스 시스템 내에는 task 개별의 Queue는 없기 때문에 Task Staging Queue를 제작하고 가상시간을 태스크 별로 계산하여 SSQ에 할당해 준다.



[그림 3-1] 제안하는 스케줄링 모델

[그림 3-1]에서 보듯 SSQ마다 여러개의 TSQ를 두고 이를 HDQ에서 SSQ로 dispatch하듯 linux kernel내 함수인 generic_make_request 함수에서 TSQ에서 SSQ로 dispatch한다. 이를 평가할때에는 SSQ별로 VT의 최솟값을 계산하고 해당 Task만이 dispatch 가능하고 그렇지 않은 task의 경우에는 1ms timer event가 발생할때에 다시 기회를 부여받기를 대기하며 TSQ에 머물게 된다.



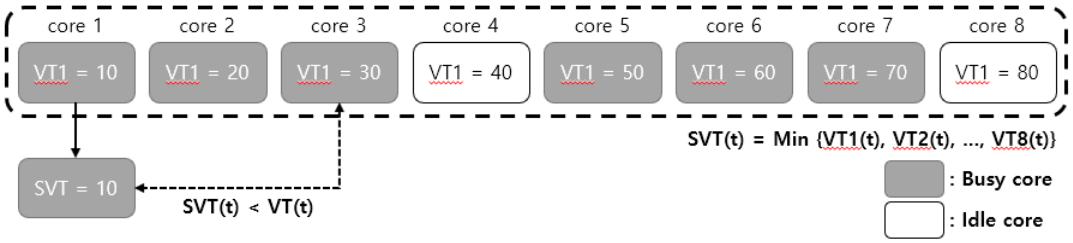
[그림 3-2] TSQ Payload

TSQ의 구조는 CPU 코어별로 TSQ와 VT(Virtual Time) 해당하는 Task가 담긴 Payload를 리스트 형식으로 관리한다. 이 Payload는 태스크가 처음 Block I/O 작업을 시행할 때 생성 되며 busy flag가 내려가고 1ms 동안 I/O 요청이 발생하지 않는다면 삭제된다. generic_make_request에서 내려오는 I/O request의 형태가 bio이고 이를 리눅스에서도 이미 bio_list라는 구조체 형태로 관리하고 있다. TSQ는 리눅스의 기본 struct인 bio_list를 이용한다. bio_list 또한 리눅스 내에서는 linked list 형식으로 동작하고, TSQ를 담고 있는 TSQ Payload또한 linked list형식으로 관리해 각 CPU 코어 마다 리스트를 하나씩 갖고 잇는 형태이다.

도입하는 스케줄러는 각각의 IO request를 bio list 구조로 저장하고, 배열의 각 요소는 지정된 task의 VT(Virtual Time)를 나타낸다. generic_make_request_checks는 리눅스가 해당 io request를 처리할지에 대해 판단하는 부분이다. VT의 최솟값을 구하는 과정은 FlashFQ와 동일하다. FlashFQ에서는 최소 VT를 참조하여 해당 코어의 요청이 서비스되어 VT가 변경되는 경우 VT 리스트에 접근하여 변경된 VT를 업데이트 한다.

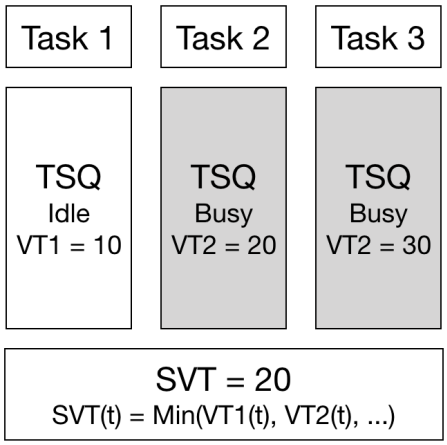
수식 (2)에 따라서 최소 VT를 참조하여 $SVT(t)$를 정하고 나면 $SVT(t)$는 각 코어의 관점에서 다음 두 가지 경우에 참조할 필요가 발생한다. 첫 번째 경

우는 유휴 큐에 새로운 요청이 도착할 때, 요청의 시작 태그가 $SVT(t)$로 결정되어야 하는 경우이다. 두 번째 경우는 코어 가 $VT_i(t) \leq SVT(t)$인지 검사함으로써 SQ_i에 있는 요청들을 SSD로 최종전달(dispatch)할지를 결정하는 경우이다. 제안하는 스케줄러에서 각 코어는 (1) 요청이 SQ_i에 도착할 때마다, (2) SQ_i로부터 전달된 요청이 완료될 대마다, 또는 (3) 각 코어에 관련된 주기적인 타이머가 만료될 때마다 이 테스트를 수행한다.



[그림 3-3] FlashFQ의 스케줄링 결정 [10]

스케줄링 결정은 바쁜 코어 간에 이루어지므로 바쁜 코어들의 VT를 비교하여 $SVT(t)$를 정한다. 예를 들어, 시간 t에서 코어 3이 요청을 서비스 받기를 원하는 경우 [그림 3-2]에서와 같이 VT 리스트 중 최소 VT가 $SVT(t)$로 정해진다. 이렇게 정해진 $SVT(t)$와 서비스를 받고자 하는 코어 3의 $VT(t)$를 비교하는데, 이때 $VT(t)$가 $SVT(t)$보다 크기 때문에 코어 3 의 요청은 서비스를 받지 못하고 다음 차례를 기다린다. 위 방식과 동일한 알고리즘으로 $SVT(t)$를 측정하고 측정된 $SVT(t)$를 갖는 TQ_i가 dispatch의 자격을 얻고 SSQ에 dispatch할 수 있다.



[그림 3-4] 제안하는 스케줄러의 SVT 결정

Ⅳ. 모의실험 결과 및 토의

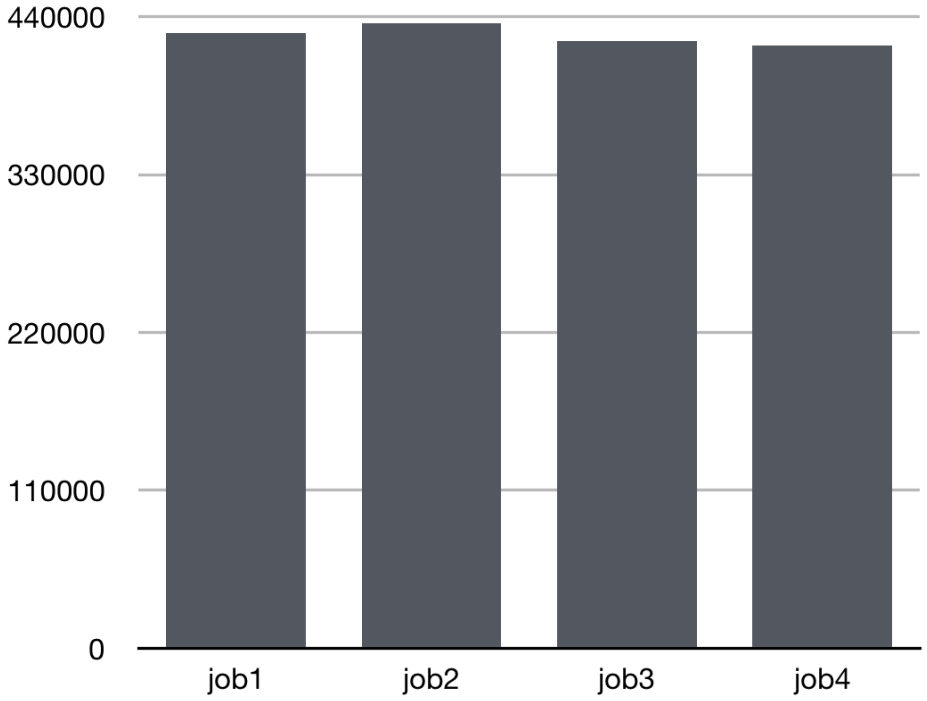
Ⅳ-1. 실험 환경

제안한 스케줄러의 성능 평가를 위해서 8개 코어를 갖는 2.0 GHz Intel Xeon CPU와 128GB DRAM, NVMe SSD인 Intel SSD 750 Series를 장착한 컴퓨터 시스템을 실험에 사용하였다. 제안하는 스케줄러는 block-mq를 사용하는 리눅스 커널 4.8 버전에 구현하고 상기 시스템에서 실행하였다. 입출력 응용 프로그램으로는 SSD 성능 평가에 많이 사용되는 FIO 벤치마크[11]를 사용하였다.

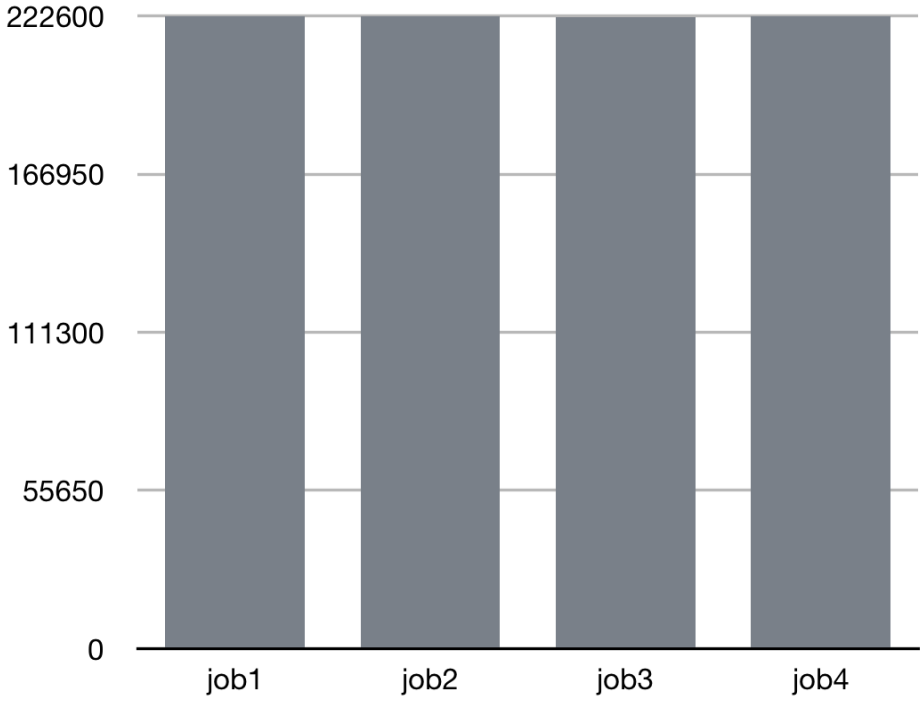
Ⅳ-2. 성능 평가

Ⅳ-2-1. 성능 비교

제안하는 스케줄러는 스케줄러를 작용하지 않은 커널과 비교하였으며, 4개의 태스크를 생성해서 모두 동일한 코어 위에서 동작하도록 구성하였다. 각 태스크 마다 I/O 작업을 임의로 만들어서 부하를 테스트해서, 태스크들이 공정한 대역폭을 부여 받을 수 있는지에 대해서 태스트를 해볼 필요가 있다. 이를 위해 모든 Task가 서로 다른 I/O buffer 사이즈를 갖게 작업한다. 이 실험을 위해 블록 장치의 벤치마킹이 가능한 툴인 FIO를 이용해서 부하 테스트를 진행했다.



[그림 4-1] 제안하는 스케줄러의 읽기 처리량 비교



[그림 4-2] 제안하는 스케줄러의 쓰기 처리량 비교

[그림 4-1]은 동일 코어 위에 4개의 태스크가 64 KB 크기의 읽기 요청들만을 비동기적으로 계속 생성하는 경우이며, [그림 4-2]는 동일 코어 위에 4개의 태스크가 64 KB 크기의 쓰기 요청들만을 비동기적으로 계속 생성하는 경우이다.

NVMe SSD인 Intel SSD 750 Series의 최대 BW를 스케줄링 없는 상태의 커널과 BW를 약간의 오버헤드를 갖지만 거의 동일하게 사용하는 것을 알 수 있다. 읽기부분에서는 약간의 오버헤드를 발견할 수 있었지만 쓰기 부분에서는 크게 오버헤드가 없었다. 이 부분은 스케줄러 추가에 대한 오버헤드이며 새로운 알고리즘의 도입을 통해서 해결해야할 부분이다. 읽기와 쓰기에서 오버헤드의 차이는 SSD의 작업 시간을 고려할 수 있다. SSD의 작업 시간이 스케줄링에 걸리는 시간보다 큰 경우 오버헤드가 무시될 것으로 추측된다. 이런 점을 고려할 때 Read 작업을 위해서는 조금 더 빠른 스케줄링이 가능한 스케줄링 기법을 접근해 볼 필요가 있다.

Ⅴ. 결 론

본 논문은 동일 CPU 코어 위에서 동작하는 태스크 간에 멀티 큐 블록 장치의 대역폭을 공정하게 분배하기 위해서 커널 레벨에서 공정 큐잉 기반 입출력 스케줄러를 제안한다. 제안하는 스케줄러는 FlashFQ의 태스크 레벨에서 발생할 수 있는 문제점에 대해서 일정 수준의 태스크 레벨에서 공정한 스케줄링과 처리가 가능하다. 본 논문에서는 FIO 벤치마크를 이용한 실험을 통해서 스케줄링 기능이 없는 Block-mq와 이 논문에서 작성된 스케줄러를 도입한 Block-mq를 비교하였다. 위와 같은 실험을 통해서 제안하는 스케줄러가 공정한 대역폭 분배를 달성함을 확인하였다.

향후 공정성과 성능을 모두 보장하는 스케줄러를 위해 제안한 스케줄러에 다양한 알고리즘을 적용하여 스케줄링 결정을 수행할 수 있다. VT 리스트에서 샘플링 하는 VT의 개수를 무한히 증가시켜도 오버헤드에 큰 영향을 주지 않는 알고리즘을 적용하고, 태스크가 코어를 이동하면 해당 태스크에 VT의 처리에 대해서 조금 공정한 스케줄링이 가능하다면 공정성과 성능을 함께 보장할 수 있을 것이다.

참고문헌

[1] NVM Express. [Onine]. Available: https://ko.wikipedia.org/wiki/NVM_익스프레스

[2] NVM Express, Inc. NVM Express: Non-Volatile Memory Express Standard, version 1.2. Specification. (2014)

[3] Dell, Inc. Kelvin Marks, "An NVM Express Tutorial", Flash Memory Summit 2013, pp.11, 2013

[4] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Schedler", Proc. of the 10th USENIX Conference on File and Storage Technologies, USENIX FAST'12, pp. 1-13, San Jose, CA, USA, 2012.

[5] Shen, K., & Park, S. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In USENIX Annual Technical Conference (pp. 67-78). (2013, June).

[6] Bjørling, M., Axboe, J., Nellans, D., & Bonnet, P. Linux block IO: introducing multi-queue SSD access on multi-core systems. In Proceedings of the 6th international systems and storage conference. pp. 1-10. (2013, June)

[7] P. Goyal, H. M. Vin, and H. Cheng, "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks", IEEE/ACM Transactions on Networking, Vol. 5, No. 5, pp. 690-704, Oct. 1997.

[8] Kanghee Kim, Fair Bandwidth Sharing of Multi-queue SSDs on Multi-core Processor pp. 4-5

[9] Thomas-Krenn.AG. [Online]. Linux Multi-Queue Block IO Queueing Mechanism. (blk-mq) Available: https://www.thomas-krenn.com/en/wiki/Linux_Multi-Queue_Block_IO_Queueing_Mechanism_(blk-mq)

[10] 조민정, 멀티큐 SSD의 공정한 대역폭 공유를 위한 입출력 스케줄링 = I/O Scheduling for Fair Bandwidth Sharing of Multi-queue SSDs, 숭실대학교 대학원, (2017, 6)

[11] Fio: Flexible IO Tester. [Online]. Available: hppts://github.com/axboe/fio (downloaded 2016, Dec. 1).

부 록

부록 1. 설치 방법 및 개발 환경

이 논문의 작품은 ubuntu 16.04 LTS 버전에서 작업했다. 그리고 사용한 linux kernel로는 debian 버전의 리눅스에서 사용 가능한 최신의 4.8 버전의 kernel 이미지를 사용해서 작업했다.

설치는 기존의 linux kernel source에 수정한 부분을 반영하고 makefile script를 이용해서 kernel build configuration과 build 과정을 거친 뒤 일반적인 kernel 설치 과정을 거치면 된다. 이 설치 과정에서 문제가 발생한 경우에는 올바른 kernel version을 사용하고 있는가, kernel module은 정상적으로 설치되었는가 확인해보고 작업하기를 권장한다.

성능이 실제 이론이나 이 논문에 작성된 수치가 나오는지 확인하기 위한 벤치마킹 작업을 위해서는 fio(Flexible Input/Output Tester) 프로그램을 활용하기를 권장한다. 그리고 fio의 설정 파일을 CPU mask를 설정하고 테스팅하기를 권장합니다.

1) 리눅스 커널 설치

/usr/src/ 디렉토리에서

apt-get install linux-source-버전이름

명령어를 통해서 원하는 linux 커널 소스 파일을 다운받는다. 커널 버전은

uname -r

명령어로 확인이 가능하다. 이 논문의 경우에는 4.8 버전의 커널을 사용했고 우분투 환경에서 테스트 했다. 이 버전과 동일하게 사용할 것을 권장한다.

2) 패키지 설치

커널 컴파일을 위해서 다음과 같은 명령어를 통해 패키지들을 설치해 주어야 한다.

sudo apt-get install build-essential libncurses5 libncurses5-dev kernelpackage -y

3) 커널 소스를 수정한다.

multi-queue 에 대한 소스들의 위치는 다음과 같다.

/block/blk-core.c

/block/blk-mq.c

/block/Makefile

/block/blk-mq-iosched.c (Scheduler 구현을 위해 새로 추가한 소스)

4) 커널을 새로 컴파일, 인스톨 한다.

빌드 이전에 커널의 빌드 설정을 위해서

```
make config
```

명령어를 이용해서 기존 커널의 빌드 설정 파일을 불러온다.
/usr/src/linux-source-해당버전 디렉토리에서 다음과 같은 명령어를 실행한다.

```
make -j 16
make -j 16 modules
make modules_install
make install
```

위 명령어들 중에서 -j 옵션을 사용하는 이유는 가능한 CPU들을 모두 사용하기 위해서이다. 이 옵션을 사용하지 않는 경우에는 1개 CPU만을 사용해서 컴파일을 진행하게 되며, 이로 인해 컴파일 시간이 더 오래 걸린다. -j 옵션 뒤의 숫자는 사용할 최대 CPU의 개수를 나타내며, 일반적으로 현재 사용 가능한 CPU개수의 2배 를 적어준다.

5) 커널을 설치한 버전으로 재부팅한다.

터미널에서 reboot 명령어를 통해 컴퓨터를 재부팅한다. 재부팅 도중에 shift 키를 눌러서 커널 버전을 선택할 수 있다. 새롭게 설치한 버전의 커널을 선택해서 부팅이 완료되면 터미널 창에서 다음과 같은 명령어를 통해 버전이 제대로 바뀌었는지 확인한다.

```
uname -r
```

6) job 파일을 통해서 fio를 테스트 한다.

fio 는 job파일 (ex. job.ini) 파일을 통해서 시나리오를 설정할 수 있다. /home/rubicom/testroom에 들어있는 fio파일을 참고해서 job파일을 작성한 이후 다음과 같은 명령어를 통해 테스트를 진행할 수 있다.

```
fio job.ini
```

위 명령어를 통해 나타난 fio 결과값들 중 io(IO, 처리한 io의 양)과 bw(Bandwidth, 초당 처리한 io의 양) 을 통해서 각 CPU를 통해 처리된 요청의 양을 확인할 수 있다.
단, NVMe 저장장치에 데이터가 차 있는 경우 데이터 저장 상황에 따라서 테스트과에 영향을 미칠 수 있다 따라서 매번 fio 벤치마크를 실행할 때마다 NVMe 저장장치를 포맷해 주어야 한다.

job.ini 파일

```
[global]
filename=/dev/nvme0n1
ioengine=libaio
runtime=20
bs=16k
direct=1
invalidate=1
randrepeat=0
log_avg_msec=1000
#group_reporting=1
time_based
thread=1
iodepth=128
```

```
[job1]
cpumask=128
rw=randread

[job2]
cpumask=128
rw=randread

[job3]
cpumask=128
rw=randread

[job4]
cpumask=128
rw=randwrite
```

설명

filename: 테스트할 disk를 설정할 수 있다.
cpumask: 비트 플래그 방식으로 원하는 CPU 코어에 마스킹이 가능하다.
rw: Read와 Write설정이 가능하다.
runtime: fio툴의 동작 시간 설정이 가능하다.
bs: buffer size를 설정할 수 있다.

부록 2. Source code 및 설명

Task level 스케줄러 (block/blk-mq-iosched.c)

```
bool
blk_mq_iosched_checks(struct request_queue *q, struct bio *bio)
{
    struct blk_mq_iosched_payload *min_payload;
    struct blk_mq_iosched_payload *current_payload;
    int cpu_id = get_cpu();
    bool retval = true;
    unsigned long long min_vruntime = 0;
    if(q->nr_hw_queues < 2)
        goto exit;
    mq = q;
    current_payload = bio->iosched_payload;

    if(current_payload != NULL) {
        blk_mq_apply_vruntime(current_payload, bio);
        goto exit;
    }
#ifdef DEBUG_IOSCHED
    pr_info("IOSCHED: checks cpu_id: %d\n", cpu_id);
#endif
    min_payload = blk_mq_iosched_get_min_payload(cpu_id);
    if(min_payload != NULL)
        min_vruntime = min_payload->vruntime;

#ifdef DEBUG_IOSCHED
    pr_info("IOSCHED: %p minvrun %llu\n", min_payload, min_vruntime);
#endif
    if(current_payload == NULL) {
        current_payload = blk_mq_iosched_list_add(current, bio, min_vruntime);
        bio->iosched_payload = current_payload;
    }
    // else if(min_payload == NULL)
    //     current_payload->vruntime = 0;


    if(current_payload->vruntime <= min_vruntime) {
```

```
            blk_mq_apply_vruntime(current_payload, bio);
            // refresh blk-mq iosched min task
            blk_mq_iosched_refresh_min_payload(cpu_id);
            goto exit;
        }

        bio_list_add(current_payload->queue, bio);
        retval = false;

        if(min_payload != NULL) {
            blk_mq_iosched_flush_payload(min_payload);
            blk_mq_iosched_cleanup_payload(cpu_id);
            blk_mq_iosched_refresh_min_payload(cpu_id);
        }

exit:
        put_cpu();
        return retval;
}
```

변수 설명:

ctx: SSQ의 구조체, ctx->rq_list에 READ/WRITE 요청들이 리스트 형태로 저장되어 있다.

request_queue: disk 장치의 request queue의 추상화 구조체

request_queue: disk 장치의 request queue의 추상화 구조체

bio: User-level에서 넘어온 I/O request

(1) bio 요청을 갖고 task의 payload를 찾는다.

(2) task payload의 VT를 찾고 TSQ를 찾는다.

(3) 해당 task payload의 VT가 SVT인가 확인하고, 최소인 경우 SSQ로 dispatch 될 수 있도록 return true를 한다.

(4) 만일 SVT가 아닌 경우 해당 task payload의 TSQ에 bio를 queuing 한다.