박 지 웅 교수 지도

석사학위 청구논문

ERB:고성능 스토리지 시스템을 위한 eBPF 기반 네트워크 및 스토리지 입출력 스택 통합 최적화 연구

2025

성신여자대학교 대학원

컴퓨터학과

안 설 령

ERB:고성능 스토리지 시스템을 위한 eBPF 기반 네트워크 및 스토리지 입출력 스택 통합 최적화 연구

박 지 웅 교수 지도

이 논문을 석사학위논문으로 제출함

2024년 11월

성신여자대학교 대학원
컴퓨터학과
안 설 령

인 준 서

안설령의 석사학위 논문으로 인준함

2024년 12월

심사위원장 김규영 (서명 또는 인)

심 사 위 원 박자응 (서명 또는 인)

심 사 위 원 임연섭 (서명 또는 인)

성신여자대학교 대학원

논문개요

저장장치 기술의 발전에 따라 메모리와의 성능 격차는 점차 줄어들고 있다. 고성능 저장장치의 등장과 함께 스토리지 스택의 소프트웨어 오버헤드는 새로운 병목 지점으로 작용하고 있다. 특히, 분산 시스템 환경에서는 클라이언트와 서버 간 요청과 응답 과정에서 커널의 네트워크 스택과 스토리지 스택을 여러 차례 왕복해야 하며, 이에 따라 심각한 문맥 교환(Context-switch) 및 데이터 복사 오버헤드가 발생한다. 이러한 오버헤드는 Linux 커널의 네트워크 및 스토리지 스택에서 발생하는 불필요한 처리 비용을 초래해 고성능 스토리지 시스템의 성능을 저하하는 주요 원인으로 작용한다.

Redis[1]는 고성능 스토리지 시스템의 대표적인 예이다. 메모리 기반 데이터 저장소로 설계된 Redis는 메모리 중심의 구조와 Append-Only File(AOF) 등의 영속성 지원을 통해 데이터의 신뢰성과 안정성을 제공하며, 고성능 스토리지 시스템으로 활용될 수 있다. 그러나 Redis 역시 커널 소프트웨어 스택 오버헤드로 인해 높은 처리량이 요구되는 환경에서 성능이 제한될 수 있다.

이러한 문제를 해결하기 위해 최근에는 Linux 커널의 네트워크 스택이나 스토리지 스택을 우회하여 불필요한 오버헤드를 근본적으로 제거하는 커널 우회(Kernel Bypass) 기술이 개발되었다[29, 47]. 하지만 이러한 커널 우회 기술은 CPU 자원 낭비가 매우 심하고, 사용자가 커널이 제공하던 기능을 직접 재구현해야 하는 등의 한계를 가지고 있다.

본 논문에서는 이러한 한계를 극복하고 고성능 스토리지 시스템의 성능을 극대화하기 위해 eBPF(extended Berkeley Packet Filter)[6]를 활용한 네트워크 및 스토리지 통합 입출력 프레임워크인 ERB(eBPF-based Redis Boos

ter)를 제안한다. ERB는 Redis[1]의 성능 최적화에 초점을 맞추어 설계된 프레임워크로, TCP 프로토콜을 기반으로 동작하며 eBPF XDP와 TC 훅 지점을 활용하여 고속의 패킷 처리를 구현한다. 또한 Redis의 데이터 영속성을 지원하기 위해 커널 모듈로 구현된 ERB 입출력 모듈을 사용하여 네트워크 패킷 처리 과정에서 디스크 입출력을 분리하고, 비동기적인 커널 페이지 캐시 작성 및 *fdatasync()*를 호출을 통해 클라이언트의 체감 응답 시간을 효과적으로 감소시킨다. 실험 결과, 멀티스레드 기반으로 성능 최적화를 적용한 Redis 대비 최대 6.5배의 성능 향상을 달성하였으며, 99번째 백분위수 응답 시간도 평균 2.5배 ~ 3배가량 감소시켰다. 이를 통해 고성능 스토리지 시스템의 성능 향상을 위해서는 네트워크와 스토리지 스택의 통합적인 최적화가 필수적임을 확인하였다.

목 차

논문 개요

그림 목차

표 목차

I. 서 론

고성능 메모리 기술의 비약적 발전을 통해 Intel Optane SSD, Samsung Z-SSD, Toshiba XL_FLASH과 같은 NVMe(Non Volatile Memory express) 저장장치는 약 3마이크로초의 저지연과 7GB/s의 대역폭을 달성하였다[2, 46]. 하지만, 저장장치의 성능이 향상됨에 따라 Linux 커널의 스토리지 스택은 입출력 성능의 새로운 병목 지점이 되고 있다[22]. PCIe 3.0 기반의 NVMe 저장장치의 경우 스토리지 스택 소프트웨어 오버헤드로 인해 발생하는 지연 시간은 전체 입출력 지연 시간 중 약 15%만을 차지하지만, PCIe 4.0 기반의 최신 NVMe 저장장치의 경우 읽기 작업 시간의 절반을 스토리지 스택 소프트웨어 지연 시간이 차지한다[21]. 현대 스토리지 시스템은 스토리지 스택의 소프트웨어 오버헤드뿐만 아니라 심각한 네트워크 스택 오버헤드 역시 겪고 있다. 이는 스토리지 시스템의 특성상 네트워크를 통한 패킷 송수신이 필수적이며, 이 과정에서 커널 영역과 사용자 영역 사이의 막대한 문맥 교환 오버헤드와 데이터 복사 오버헤드가 발생하기 때문이다.

현재 네트워크 스택의 불필요한 오버헤드를 줄이거나, 효율적인 스토리지 스택을 구현하기 위해 eBPF[6]를 활용한 연구가 제안되고 있다. eBPF는 커널 소스 코드 수정 또는 추가적인 커널 모듈 생성 없이 사용자 영역에서 작성한 프로그램을 Linux 커널 내부에서 안전하게 실행시킬 수 있다. 특히 eBPF가 제공하는 XDP 훅과 TC 훅을 사용하는 경우 Linux 커널의 네트워크 스택을 거치지 않고 eBPF 프로그램 내부에서 패킷을 조작 후 클라이언트에게 재전송하여 스토리지 시스템 서버와 클라이언트 간 불필요한 문맥 교환 오버헤드를 획기적으로 줄일 수 있다[11]. 이러한 방식은 DPDK[47]와 SPDK[29] 같이 커널을 우회하여 Linux 커널 스택의 오버헤드를 완전히 제거할 수 있지만 구현이 매우 복잡하고 CPU 낭비를 발생시키며, 커널이 제공하

는 유용한 기능들을 사용할 수 없는 커널 우회 방식[47, 29, 33-35]의 대안으로 활용될 수 있다.

최근 eBPF를 활용하여 네트워크 스택과 스토리지 스택의 오버헤드를 줄이고 효과적인 성능 향상을 달성한 연구가 제안되고 있다[14, 21-25, 38-40]. 하지만 해당 논문들은 eBPF를 사용하여 네트워크 스택과 스토리지 스택 각 각을 최적화할 뿐, 두 스택 모두를 최적화하는 연구는 아직 제안되지 않았다. 하지만 고성능 스토리지 시스템은 패킷 처리를 위한 네트워크 최적화와 데이터 영속성 보장을 위한 스토리지 최적화 모두가 필요하다.

본 논문은 네트워크를 통한 패킷 송수신과 강력한 데이터 영속성을 제공하여 고성능 스토리지 시스템으로 활용할 수 있는 Redis[1]를 활용한다. 따라서 고성능 스토리지 시스템으로서의 Redis 성능 향상을 위해 eBPF를 활용하여 네트워크 스택과 스토리지 스택을 통합 최적화한 새로운 형태의 입출력 프레임워크인 ERB(eBPF-based Redis Booster)를 제안한다. ERB는 eBPF XDP 훅과 TC 훅을 사용한 고성능 패킷 처리를 달성하며 입출력 모듈을 통한 디스크 입출력을 가능하게 한다.

ERB의 주요 기여 내용은 다음과 같다.

첫째, 네트워크 스택과 스토리지 스택의 통합 최적화를 통해 고성능 스토리지 시스템에서의 성능 향상에 기여한다.

둘째, XDP 훅에서의 디스크 입출력 제한을 해결하기 위해 커널 스레드를 사용하여 비동기적인 커널 쓰기 작업을 가능하게 한다.

셋째, eBPF Verifier의 한계를 극복하고 스토리지 스택 최적화를 위해 다양한 eBPF kfuncs[16] 함수를 제공한다.

II. 배 경

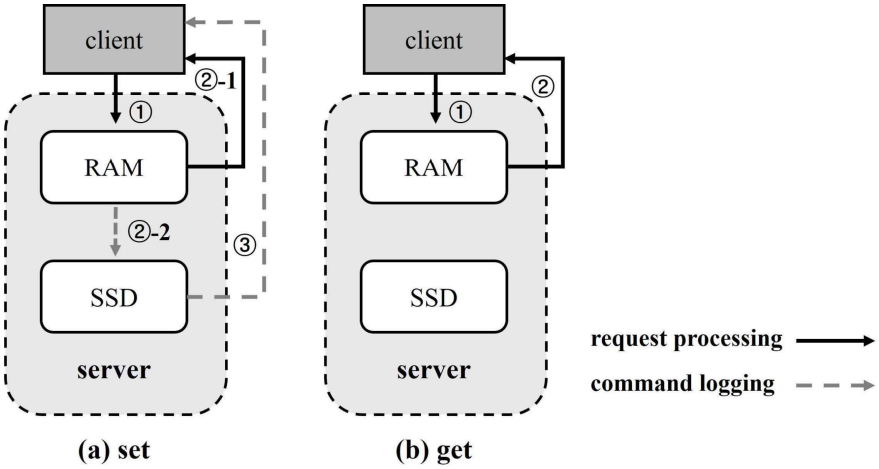
1. Redis

Redis[1]는 오픈소스 기반의 인-메모리 키-밸류 스토어로 메인 메모리를 활용한 고속의 데이터 처리를 통해 높은 성능을 제공한다. 따라서 캐싱, 실시간 트렌젝션 분석, 세션 관리 등 빠른 응답이 요구되는 다양한 서비스에서 사용되고 있다. 인-메모리 데이터베이스의 경우 휘발성 메인 메모리에 의존하기 때문에 시스템 장애 발생 시 메모리에 저장된 데이터가 소실될 가능성이 높다. Redis는 이러한 인-메모리 데이터베이스의 한계를 극복하기 위해 다양한 데이터 영속성 보장 방법을 제공한다. Redis의 데이터 영속성 보장 방법은 명령어 실행 내용을 로그 형태로 저장하는 것이다. 본 논문에서는 이러한 데이터 영속성 보장 방법을 Redis 로깅 방식으로 정의한다. Redis는 이러한 로깅 방식을 통해 Redis 서버에서 실행 된 명령어 내용을 SSD에 기록하여 시스템 장애, 전원 장애 이후 데이터를 복구할 수 있는 신뢰성을 제공한다. 따라서 Redis는 데이터 영속성을 제공하여 데이터 손실 가능성을 현저히 낮추고, 복잡하고 다양한 데이터 타입을 제공하며[3], 클러스터링과 레플리카[41, 42]를 지원하여 확장성이 높다는 점에서 고성능 스토리지 시스템으로 활용할 수 있다.

1) 데이터 영속성을 고려한 Redis의 명령어 처리 과정

Redis 로깅 방식은 Redis 데이터베이스의 상태가 변화하는 경우 동작한다. 따라서 Redis 명령어 처리 과정은 Redis 로깅 여부에 따라 데이터를 저장하는 명령어와 데이터에 단순 접근하는 명령어로 나눌 수 있다. 그림 1은

이러한 특징을 반영한 Redis의 대표적 명령어 set, get의 동작 과정이다.



[그림 1] Redis의 명령어 처리 과정

그림 1-(a)는 set 명령어 처리 과정으로 ① Redis 클라이언트가 set 명령어를 요청하는 경우 Redis 서버는 명령어 패킷을 분석하여 키와 밸류 값을 파싱한다. 이후 RAM에 위치한 Redis 데이터베이스에 해당 키, 밸류 값을 저장한다. Redis 로깅 방식은 필수가 아닌 옵션으로 Redis 서버가 선택하여 사용할 수 있다. Redis 로깅 방식을 사용하지 않는 경우, ②-1 Redis 서버는 set 명령어에 대한 응답 값을 Redis 클라이언트에게 즉시 전송한다. Redis 로깅 방식을 사용하는 경우 ②-2 명령어 내용을 저장하기 위해 *write()* 시스템 콜을 호출한다. 마지막으로 ③ set 명령어에 대한 응답을 Redis 클라이언트에게 전송한다. 다음으로 그림 1-(b)는 get 명령어 처리 과정이다. get 명령어는 데이터베이스에 접근해 기존에 저장된 데이터를 사용하는 것뿐, 키, 밸류 값에 변화를 발생시키지 않아 Redis 로깅 과정이 실행되지 않는다. 따라서 ① Redis 클라이언트가 get 명령어를 요청하는 경우 Redis 서버는 명령어 패킷을 분석하여 키 값을 파싱한다. 이후 데이터베이스에서 해당 키

값을 탐색하고 ② 키에 대한 밸류 값을 Redis 클라이언트에게 전송하는 것으로 완료된다.

2) Redis의 데이터 영속성 보장 방식

앞서 Redis는 데이터 영속성을 제공하기 위해 데이터베이스 상태에 변화가 생기는 경우 Redis 로깅 방식을 사용한다고 설명했다. Redis는 사용 환경에 따라 더욱 효율적인 로깅 방식을 지원하기 위해 RDB(Redis Database) 방식과 AOF(Append Only File) 방식 2가지를 제공한다[1]. 각 방식은 단일 로그 파일로 유지되며 “.rdb”, “.aof” 접미사를 사용해 구분된다.

(1) RDB (Redis DataBase)

RDB는 특정 시간 간격마다 데이터베이스의 데이터 세트를 스냅샷 형태로 생성하여 디스크에 저장하는 방식이다. 스냅샷은 생성 시점의 데이터베이스 전체 상태를 나타내기 때문에 데이터 복원 시 특정 시점의 테이터 상태로 빠르게 복원할 수 있는 장점이 있다. 또한, 스냅샷 생성은 Redis 서버의 명령어 처리 과정과 비동기적으로 진행되기 때문에 명령어 처리 과정을 담당하는 Redis 메인 프로세서의 성능에 큰 영향을 끼치지 않는다. 하지만, 스냅샷 저장 간격이 긴 경우 마지막 스냅샷 이후의 데이터는 소실 될 가능성이 크며, 명령어 단위의 즉각적인 로깅 방식이 아니기 때문에 실시간 데이터 복구가 어려운 단점이 있다.

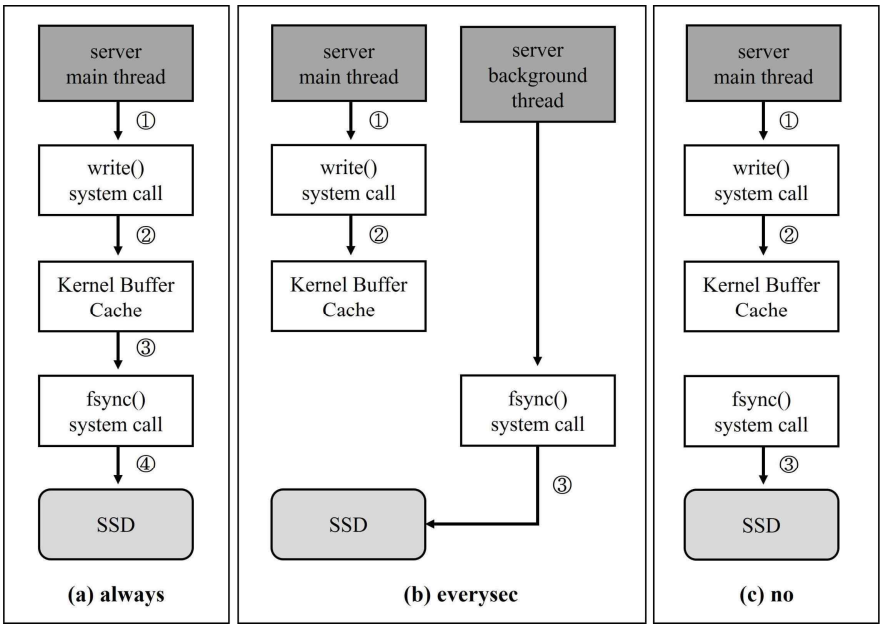
(2) AOF (Append Only File)

AOF는 데이터베이스 상태를 변화시키는 명령어에 대해 즉각적인 로깅을 실행하는 방식이다. 따라서 특정 시간 간격 단위로 동작하는 RDB와 달리 AOF는 명령어 단위로 실행되어 데이터 영속성이 강하게 보장된다. 이러한 방식으로 인해 AOF는 데이터 소실 가능성이 매우 낮고, 실시간 데이터 복구가 쉬운 장점이 있다. 하지만 RDB에 비해 디스크 입출력 작업이 빈번하

게 호출되어 시스템 전체 성능에 악영향을 끼칠 가능성이 존재한다. 또한, 데이터 복원 시 로그 파일에 저장된 명령어 내용 전체를 재실행하는 방식으로 동작하기 때문에 재시작 시 많은 시간이 소요되는 단점이 있다.

3) Redis AOF

Redis를 고성능 스토리지 시스템으로 활용하기 위해선 데이터 영속성을 강하게 보장하여 데이터 손실 가능성을 현저히 낮춰야 한다. 따라서 AOF 방식은 고성능 스토리지 시스템으로의 활용을 위해 필수적이다. AOF 과정은 크게 2가지 동작으로 구분된다. Redis 서버가 명령어를 처리한 후 커널 버퍼에 로그 내용을 기록하는 과정, 그리고 디스크 입출력을 호출하여 커널 버퍼의 내용을 SSD에 내려 적는 과정이다. AOF 방식은 디스크 입출력 호출 빈도에 따라 3가지 방식으로 나누어진다. 그림 2는 디스크 입출력 호출 빈도에 따른 AOF 방식별 실행 과정이다.



[그림 2] Redis AOF 방식

(1) Redis AOF always

그림 2-(a)는 AOF always 방식의 동작 과정을 나타낸다. AOF always는 데이터베이스의 상태를 변화시키는 모든 명령어에 대해 *fdatasync()* 시스템 콜을 호출하여 디스크 입출력을 발생시킨다. 자세한 동작 과정으로는 ① Redis 서버의 메인 스레드가 명령어 처리를 완료한 후 *write()* 시스템 콜을 호출하여 ② 커널 버퍼에 명령어 내용을 작성한다. ③ 이후 즉시 *fdatasync()* 시스템 콜을 호출하여 ④ 커널 버퍼에 작성된 로깅 내용을 SSD로 내려 적는다. AOF always 방식의 경우 명령어 단위로 플러시(flush) 작업이 실행되기 때문에 데이터 소실 가능성이 매우 낮아 영속성이 가장 강하게 보장된다. 하지만 빈번한 디스크 입출력 호출은 SSD의 높은 쓰기 지연 시간과 입출력 병목 현상을 악화시켜 시스템의 전반적인 성능 하락을 초래할 가능성이 크다.

(2) Redis AOF everysec

그림 2-(b)는 AOF everysec 방식의 동작 과정을 나타낸다. AOF everysec는 ① Redis 서버의 메인 스레드가 명령어를 처리한 뒤 *write()* 시스템 콜을 호출하여 ② 커널 버퍼에 명령어 내용을 즉각적으로 저장한다. 이후 AOF always 방식과 다르게 커널 버퍼 작성 후 동기적인 디스크 입출력을 호출하지 않고 ③ Redis 서버의 백그라운드 입출력 스레드(bio-threads)를 사용하여 매초 *fdatasync()* 시스템 콜을 호출한다. 즉, 명령어 단위가 아닌 1초 단위로 플러시 작업을 실행하는 것이다. 따라서 마지막 *fdatasync()* 호출 후 1초 이내에 서버 장애가 발생하는 경우 해당 *fdatasync()* 호출과 서버 장애 사이에 생성된 데이터는 소실 될 가능성이 매우 크다. 따라서 AOF everysec 방식의 경우 AOF always 방식에 비해 낮은 데이터 영속성을 보장하지만, 더 적은 디스크 입출력 호출과 백그라운드 입출력 스레드를 사용한 플러시 처리를 통해 시스템 성능 하락을 피하며 영

속성을 보장할 수 있다는 장점이 있다.

(3) Redis AOF no

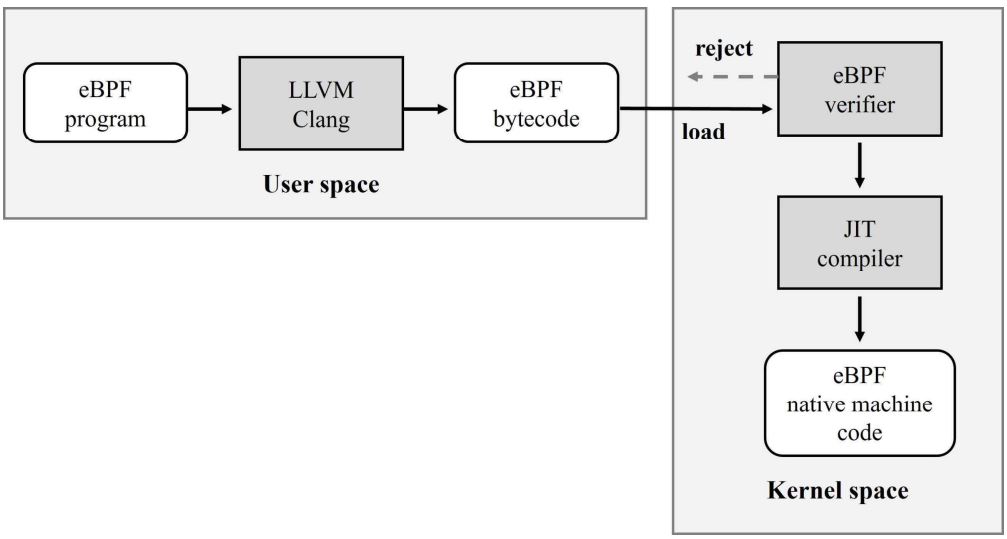
그림 2-(C)는 AOF no 방식의 동작 구조를 나타낸다. AOF no 방식은 ① Reids 서버의 메인 스레드가 명령어 처리 후 *write()* 시스템 콜을 호출하고 ② 커널 버퍼에 해당 내용을 로깅 하는 점에서 앞선 AOF 방식과 같다. 하지만 *fdatasync()* 호출의 경우 ③ Redis 서버는 기본 운영 체제의 플러시 정책에 의해 주기적으로 실행되며, 일반적인 Linux 커널의 경우 30초마다 *fdatasync()*가 호출된다. 해당 방식은 앞선 AOF always와 AOF everysec 방식과 비교해 가장 적은 디스크 입출력을 호출한다는 점에서 최상의 성능을 달성할 수 있지만 데이터 소실 가능성이 가장 크다.

Redis는 시스템 성능과 데이터 영속성 간의 균형을 고려하여 약간의 데이터 소실이 허용되는 경우 AOF everysec을 추천한다[1]. 이는 성능이 중요한 환경에서 AOF always로 인해 발생하는 심각한 성능 저하를 고려한 결과이다.

2. eBPF

eBPF(extended Berkeley Packet Filter)[6]는 BPF(Berkeley Packet Filter)[7]의 확장 기술로 커널 코드 수정 또는 추가적인 커널 모듈 생성 없이 개발자가 작성한 사용자 영역의 코드를 Linux 커널 내부에서 안전하게 실행시키는 기술이다. 사용자 영역의 코드를 Linux 커널 내부에서 안전하게 실행하기 위해선 엄격한 프로그램 안전성 검사와 특수한 컴파일 과정이 필수적이다[48]. 그림 3은 eBPF 프로그램의 컴파일 과정이다. 우선 개발자는 C언어 또는 Python과 같은 고급어(High-level language)로 eBPF 프로그램을 작성한 뒤, LLVM/Clang 컴파일러를 통해 eBPF 프로그램을 eBPF 바이

트코드로 변환한다. 변환된 eBPF 바이트코드는 eBPF 훅 지점이라고 불리는 미리 지정된 eBPF 실행 위치에 옮겨져 실행 준비를 시작한다. 이때, eBPF 프로그램이 Linux 커널 내부에서 실행될 수 있는 안전한 프로그램인지 확인하는 과정이 필수적이다. 따라서 eBPF Verifier[8]는 eBPF 프로그램의 실행 가능한 모든 경우의 수를 파악하여 메모리 할당 문제, 무한 루프 여부, 미리 지정된 명령어 개수 초과 등 다양한 검증 기준을 확인한다. eBPF Verifier를 통과 하는 경우 eBPF 바이트코드는 JIT(just-in-runtime) 컴파일러에 의해 네이티브 머신 코드로 변환되고 eBPF 가상 머신[9]을 통해 실행된다. eBPF Verifier를 통과하지 못한 경우 해당 프로그램은 커널 내부에서 실행될 수 없는 부적절한 프로그램으로 판단되어 Linux 커널은 해당 프로그램의 실행을 거부한다.

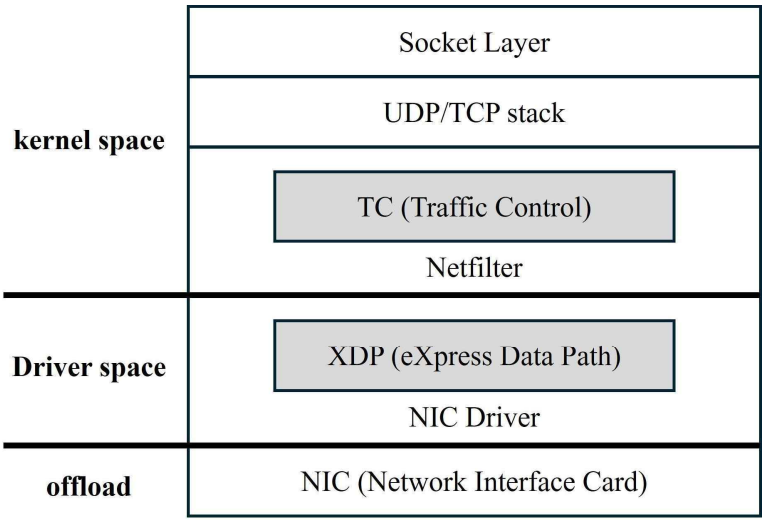


[그림 3] eBPF 컴파일 과정

1) eBPF 훅 지점

eBPF 프로그램은 이벤트 드리븐 형태로 커널 또는 애플리케이션이 미리

지정한 훅 지점에 도달하는 경우 해당 지점에 연결된 eBPF 프로그램이 자동으로 호출된다[13]. eBPF의 대표적인 훅 지점으로는 XDP(eXpress Data Path)[10,11], TC(Traffic Control)[12]가 있다.



[그림 4] Linux 커널 네트워크 스택 및 XDP, TC 훅 지점

[표 1]

XDP, TC, NIC offload 별 패킷 처리량

	NIC	XDP	TC
MRPS (Million Requests per Second)	74	20	5

그림 4는 NIC(Network Interface Controller) 드라이버와 Linux 커널의 네트워크 스택에서 XDP와 TC 훅 지점의 위치를 나타낸 그림이며, 표1은 NIC과 XDP, TC에서 측정한 초당 패킷 처리량을 나타낸 표이다[49]. XDP

는 NIC 드라이버에 위치하여 NIC에 패킷이 수신되는 즉시 호출된다. 따라서 수신된 패킷이 Linux 커널의 네트워크 스택으로 전달되어 sk_buff[17]에 할당되기 전 패킷을 가로채 임의 조작이 가능하다. 즉, 네트워크 스택 전체를 우회하는 것과 유사한 형태로 동작하며, 따라서 DPDK[18]와 비교할 만한 고속의 패킷 처리 성능을 보인다[10]. NIC으로 수신되는 패킷을 통해서만 호출이 가능한 XDP와 달리 TC는 수신되는 패킷과 송신하는 패킷 모두에서 호출될 수 있다. TC는 네트워크 7계층 중 2계층인 데이터 링크 계층과 3계층인 네트워크 계층 사이에 존재한다. 따라서 TC는 NIC 드라이버 수준에서 실행되는 XDP와 달리 네트워크 스택에 의해 패킷 내용이 sk_buff로 할당된 후 동작한다. 즉, 커널 네트워크 스택의 추가적인 데이터 복사 오버헤드로 인해 XDP에 비해 낮은 성능이 측정된다. 하지만 커널 네트워크 스택의 일부를 우회할 수 있다는 점에서 여전히 일반적인 패킷 처리 속도보다 빠르다는 장점이 있다. 표 1에 따르면 하드웨어 지원을 통해 NIC에서 패킷을 직접 처리하는 오프로딩 방식은 초당 약 74 MRPS의 패킷 처리가 가능하다. 반면 XDP와 TC는 각각 초당 20 MRPS, 5 MRPS의 패킷 처리가 가능하다. 고속의 패킷 처리를 고려한다면 NIC으로의 오프로딩 방식이 가장 우선되어야 하지만 이러한 방식은 하드웨어 의존성이 매우 높고, 사용자 요구에 대한 유연성이 제한된다는 점에서 한계가 발생한다. 따라서 하드웨어 독립적이며 커널과의 통합이 가능하고, 높은 유연성을 제공하는 XDP 훅 지점은 최근 고속의 패킷 처리를 위한 혁신적인 기술로 주목받으며, 핵심적인 네트워크 처리 지점으로 간주 되고 있다.

2) eBPF 맵(Map)

eBPF는 보안 및 데이터 안정성의 이유로 인해 외부 사용자 공간의 프로세스 또는 커널 영역과의 직접적인 데이터 교환이 제한된다. 이러한 제약을

극복하기 위해 eBPF는 키-밸류 형태의 맵 데이터 구조를 사용하여 커널과 사용자 공간, eBPF 프로그램 또는 함수 간 데이터 공유 매커니즘을 제공한다. eBPF 맵[50]은 고정된 크기로 정적 선언 및 생성되어야 하며 각 맵은 최대 $2^{32}-1$ 바이트의 요소를 $2^{32}-1$개 저장할 수 있다[14]. eBPF 맵은 여러 유형을 제공하며 사용 목적에 따라 선택할 수 있다. 대표적인 유형으로는 밸류를 배열 형태로 저장하는 BPF_MAP_TYPE_ARRAY, 입력된 키를 해싱하여 특정 버킷에 저장하는 BPF_MAP_TYPE_HASH, CPU 별로 배열 형태의 맵을 유지하는 BPF_MAP_TYPE_PERCPU_ARRAY 등이 있다. 일반적인 맵의 경우 CPU 별 독립적인 맵 구조를 유지하는 것이 아닌 Linux 커널 내의 모든 eBPF 프로그램이 공유할 수 있는 전역 객체로 사용된다. 따라서 각 CPU 코어에서 병렬적으로 실행되는 eBPF 프로그램들이 동일한 맵에 접근하여 데이터를 읽고 수정할 수 있다. 이 경우 모든 코어가 공동으로 데이터 상태를 관리할 수 있는 장점이 있지만, 데이터 동시성 문제가 발생할 가능성이 있다. 반면, CPU 별로 독립적인 맵을 유지하는 경우 코어별로 동작하는 eBPF 프로그램만이 해당 코어에서 유지 중인 맵에 접근할 수 있다. 이 경우 멀티 코어 환경에서 메모리 접근 경합을 줄여 성능 향상을 가능하게 한다.

3) eBPF Kfuncs

eBPF Verifier[9]는 eBPF 프로그램의 안전한 실행을 위해 까다로운 검증 과정을 실행하며, 이 과정에서 대부분의 커널 영역 함수와 사용자 영역 함수는 eBPF Verifier의 검증 기준을 통과하지 못한다. Linux 커널은 이와 같은 eBPF Verifier의 까다로운 함수 사용 제약을 극복하기 위해 eBPF Helper functions[51]을 지원한다. eBPF Helper function은 Linux 커널에 구현되어 eBPF 프로그램 작성에 필요한 다양한 기능을 제공한다. 대표적인

함수로는 *bpf_trace_printk(), bpf_xdp_adjust_head()*[15] 등이 있다.

eBPF kfuncs[16]는 Linux 커널 5.13부터 도입된 것으로 eBPF Helper function의 대안으로 제시된다. eBPF Helper function은 eBPF 프로그램만을 위해 생성된 함수로 eBPF 프로그램 작성 용이성을 높이지만 주로 패킷 처리에 관련된 함수만을 제공하고, Linux 커널에 정의된 기존 커널 영역 함수는 사용할 수 없다는 한계가 있다. eBPF Kfuncs는 이러한 문제를 해결하기 위해 eBPF 프로그램에서 Linux 커널 함수 호출을 가능하게 하여 다양한 목적의 eBPF Helper function을 사용할 수 있게 한다. eBPF Kfuncs의 동작 과정은 (1) eBPF Kfuncs를 커널 영역에 정의한 후 (2) eBPF 프로그램에서 호출하는 과정으로 구분할 수 있다.

(1) eBPF Kfuncs 정의

eBPF Kfuncs를 정의하는 경우 *__bpf_kfunc* 메크로를 사용해야 한다. *__bpf_kfunc* 메크로는 eBPF Kfuncs가 정적 커널 함수일 경우 컴파일러가 해당 함수를 인라인(inlining) 하는 문제, 해당 함수가 커널 내부에서 실질적으로 호출되지 않아 LTO(Link Time Optimazation) 빌드 과정에서 제거되는 문제를 방지하기 위해 필수적이다. 이후 *register_btf_kfunc_id_set()* 함수를 사용해 정의한 eBPF Kfuncs를 BPF 서브 시스템에 추가한다. 이 과정에서 eBPF Kfuncs 프로그램 역시 eBPF Verifier의 함수 안전성 검증 과정을 거치며 Linux 커널 내부에서의 실행 안전성을 보장한다.

(2) eBPF 프로그램에서의 eBPF Kfuncs 호출

eBPF Kfunc는 Linux 커널 외부 공개 심볼(kernel export symbol)과 유사한 형태로 사용자 영역의 eBPF 프로그램에서 *extern* 키워드를 사용해 호출할 수 있다. 이때, 커널 영역의 심볼 테이블에서 해당 함수를 찾아 로드(load) 해야 하기 때문에 *__ksym* 메크로 사용이 필수적이다.

4) eBPF 제약 사항

eBPF 프로그램은 Linux 커널 내부에서의 안정적인 실행을 위해 eBPF Verifier[8]의 엄격한 검증 과정이 필수적이다. 이 과정에서 다음과 같은 한계로 인해 eBPF 함수 사용에 많은 제약이 발생한다.

(1) 동적 메모리 할당 제한

eBPF 프로그램은 기본적으로 정적으로 정의된 데이터만을 사용할 수 있다. 즉, 프로그램이 로드되는 시점에 모든 데이터 구조의 크기와 형태가 고정된 경우에만 메모리를 할당받을 수 있다. 이렇게 정적으로 메모리를 할당하는 경우 메모리 공간 낭비 등의 문제가 발생할 가능성이 크다. 하지만 eBPF는 대부분의 환경에서 동적 메모리 할당이 불가능하며 매우 제한적인 환경에서만 프로그램 실행 중 동적 메모리 크기 조정이 가능하다는 한계가 있다. 최근 Linux 커널은 eBPF Kfuncs에 dynptr(dynamic pointer) 함수를 추가했다[52]. dynptr의 경우 eBPF Kfunc를 사용하여 링 버퍼 등을 동적으로 할당하고 메모리 크기를 조정 할 수 있다. 하지만 eBPF Map과 같은 eBPF 주요 구조에 대한 동적인 메모리 조정과는 거리가 멀다.

(2) 명령어 수의 제한

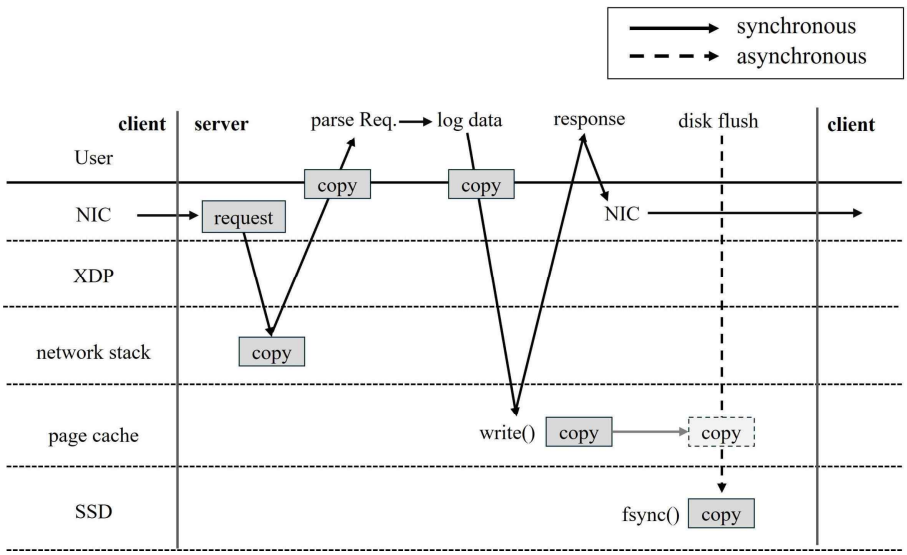
eBPF는 무한 루프 가능성을 방지하기 위해 하나의 eBPF 함수에서 실행할 수 있는 명령어 수를 백만 개로 제한한다[19]. 해당 제약으로 인해 하나의 eBPF 함수에서 복잡한 패킷 처리 과정을 모두 구현하기 어려운 한계가 발생한다. 이러한 문제를 해결하기 위해 Linux 커널은 eBPF Tail calls[20]을 지원한다. eBPF 명령어 수 제약은 eBPF 함수마다 개별적으로 적용되기 때문에 대규모의 eBPF 로직을 여러 개의 작은 eBPF 함수로 쪼개어 eBPF Tail calls로 구현하는 경우 명령어 수 제약을 통과할 가능성이 커진다. 하지만 해당 방식의 경우 코드 복잡성이 증가하며 최신 Linux 커널의 경우 최대 33개의 eBPF Tail calls만을 지원하는 한계가 있다.

III. 동기

1. 기존 Redis의 문제점

1) 지연 시간 측면의 문제점

Redis는 네트워크를 통해 데이터를 송수신하는 과정에서 잦은 문맥 교환 오버헤드와 높은 데이터 복사 오버헤드로 인해 시스템 성능 저하를 겪고 있다. 특히 성능과 데이터 영속성 보장 사이의 균형을 위한 AOF everysec 로깅 방식은 네트워크 패킷 처리와 더불어 주기적으로 *write()* 시스템 콜을 호출한다. 따라서 이로 인해 문맥 교환 오버헤드와 데이터 복사 오버헤드는 더욱 심각해진다. 그림 5는 Redis AOF everysec 로깅 방식에서 발생하는 문맥 교환 오버헤드와 데이터 복사 오버헤드를 나타낸 그림이다.



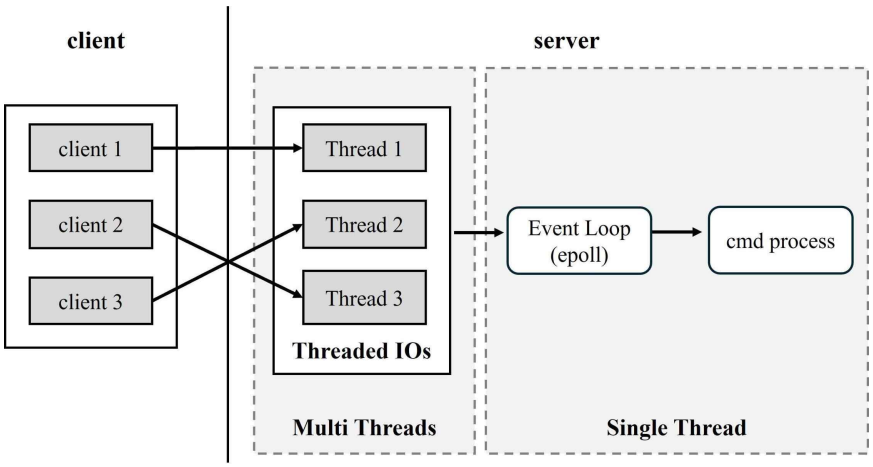
[그림 5] Redis AOF everysec 로깅 방식의 문맥 교환 오버헤드와 데이터 복사 오버헤드

Redis 서버 측 NIC에 요청 패킷이 전송된 경우 NIC은 인터럽트를 통해 커널에게 새로운 패킷이 도착했음을 알린다. 커널은 수신 큐(Rx queue)에 저장된 요청 패킷 내용을 사용하기 위해 Linux 커널 네트워크 스택을 지나며 sk_buff에 해당 내용을 복사한다. 이 과정에서 데이터 복사 오버헤드가 발생한다. 이후 Redis 서버에게 해당 패킷 내용을 전달하는데 이 과정에서 커널 영역과 사용자 영역 사이의 문맥 교환 오버헤드와 데이터 복사 오버헤드가 함께 발생한다. Redis 서버는 패킷 내용을 확인한 후 데이터 처리를 마친다. 이때 Redis 로깅을 위해 *write()* 시스템 콜을 호출하여 명령어 내용을 기록한다. 따라서 사용자 영역에서 커널 영역으로의 문맥 교환 오버헤드가 다시 발생하며 사용자 영역의 데이터인 로그 내용을 커널 영역에서 사용하기 위한 데이터 복사 오버헤드 역시 발생한다. 이후 디스크 입출력을 위해 *fdatasync()*를 호출하는 경우 이 역시 사용자 공간에서 커널 영역으로의 문맥 전환 오버헤드가 발생하며 커널 페이지 캐시에 작성된 로그 내용을 SSD로 내려 적는 데이터 복사 오버헤드가 발생한다. 문맥 전환 오버헤드가 발생하는 경우 인터럽트 확인, ISR(Interrupt Service Routine) 처리, CPU 스케쥴링 등이 순차적으로 수행되며 이 과정에서 최소 2 마이크로초에서 수십 마이크로초의 시간이 소요된다. 따라서 문맥 전환 오버헤드가 빈번히 발생하는 경우 이로 인한 지연 시간은 무시할 수 없는 수준이 되어 전체 성능에 악영향을 미칠 가능성이 존재한다. 데이터 복사 오버헤드 역시 CPU 사이클을 소비하여 시스템 메모리 대역폭을 소모하는 주된 원인이다. 따라서 반복적인 메모리 복사 역시 시스템 성능에 악영향을 미칠 가능성이 크다.

최근 이러한 문제를 해결하기 위해 제로 카피(Zero-Copy)[43] 기술과 커널 우회(kernel-Bypass)[18, 29]와 같은 새로운 입출력 최적화 방법이 연구되고 있다. 하지만 해당 기술을 사용하기 위해선 소프트웨어 스택을 재설계하거나, 클라이언트 로드(load)가 낮은 경우에도 높은 CPU 자원 사용이 필요한 단점이 존재한다.

2) 확장성 측면의 문제점

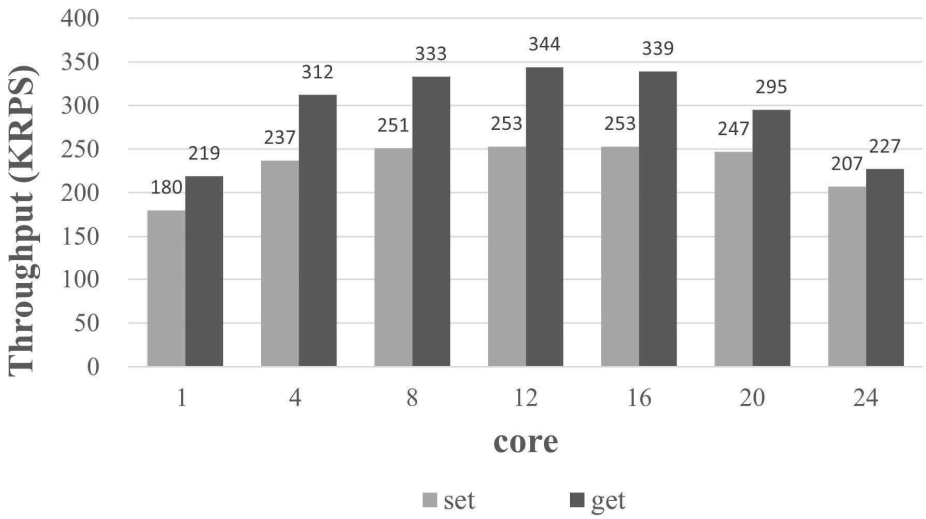
Redis는 단일 스레드 구조로, 모든 클라이언트의 요청을 하나의 서버 스레드에서 순차적으로 처리한다[4]. Redis의 단일 스레드 구조는 데이터 동기화 문제를 최소화하지만, 클라이언트의 요청이 급격히 증가하는 상황에서는 병목 지점이 되어 시스템 전체의 성능을 악화시킬 가능성이 있다. Redis는 단일 스레드로 인한 병목 문제를 해결하기 위해 Redis-6.0부터 ThreadedIO 방식을 도입한다. ThreadedIO는 부분적인 멀티 스레드를 지원하여 Redis의 주요 특징인 원자적 연산을 보장하면서 더욱 빠른 성능을 제공한다. 그림 6은 Redis-6.0 이후의 ThreadedIO 구조이다.



[그림 6] Redis ThreadedIO 구조

Redis는 소켓 읽기, 쓰기와 같은 네트워크 입출력 과정에 멀티 스레드 방식을 지원하여 병렬적인 패킷 처리를 가능하게 하고 전체적인 성능 향상을 가능하게 한다. 하지만 Redis 동작 과정의 기본 원칙인 원자적 연산을 보장하기 위해 네트워크 입출력 외의 실질적인 명령어 처리 과정은 여전히 단일 스레드로 진행된다. 따라서 클라이언트 요청이 급격히 증가하는 경우 단일

스레드로 진행되는 명령어 처리 과정은 여전히 병목 지점으로 작용 될 가능성이 크다. 그림 7은 Redis 단일 서버에서 ThreadedIO로 인한 성능 향상을 평가하기 위한 실험이다. 해당 실험은 Redis 네트워크 입출력 처리에 서로 다른 스레드 수를 할당하며 네트워크 패킷 처리의 병렬성 확장에 따른 성능 향상을 측정하였다. 실험 조건은 6장 실험 파트와 같다.



[그림 7] Redis ThreadedIO 성능 분석

실험 결과 Redis의 네트워크 입출력 병목을 해소하기 위해 최소 1개에서 최대 24개까지 스레드를 할당하여 분산적인 패킷 처리를 실행했지만, set 명령어의 성능은 최대 1.4배, get 명령어의 성능은 1.6배밖에 증가하지 못했음을 알 수 있다. 이러한 결과는 단일 스레드로 진행되는 Redis의 명령어 처리 과정이 병목 지점으로 작용함을 증명한다. 또한 ThraededIO는 네트워크 입출력 처리에 할당된 스레드들이 각각 하나의 CPU를 점유하여 패킷 처리를 위한 바쁜 루프 대기(busy-polling)을 실행한다. 따라서 사용 스레드 수

만큼의 CPU가 100%의 활용률을 달성해 CPU 자원 낭비가 매우 심하다. 해당 실험은 Redis 서버가 사용하는 CPU의 수를 총 24개로 제한하였다. 따라서 그림 7에서 ThreadedIO에 20개의 이상의 스레드를 할당하는 경우 Redis 서버의 메인 스레드와 네트워크 입출력 스레드 사이의 심각한 CPU 경쟁이 발생해 전체적인 성능 하락이 나타난다. 따라서 Redis 단일 서버 구조에서는 ThreadedIO를 사용함에도 효율적인 성능 확장이 이루어지지 않음을 알 수 있다.

eBPF XDP 훅의 경우 패킷이 Linux 커널 네트워크 스택으로 전달되기 전 NIC 드라이버 영역에서 먼저 처리되기 때문에 네트워크 지연을 효과적으로 감소시킬 수 있다. 또한 XDP 훅에 연결된 eBPF 프로그램의 경우 Redis ThreadedIO와 달리 바쁜 루프 대기 방식이 아닌 인터럽트 방식으로 실행되며, RSS(Receive Side Scaling)와 수신(Rx queue), 송신 큐(Tx queue) 코어 맵핑을 통해 효율적인 CPU 자원 활용이 가능하다. 따라서 Linux 커널 네트워크 스택의 각종 지연 시간을 감소시키고 높은 성능 확장성을 달성하기 위해 eBPF XDP 훅은 효율적인 패킷 처리 위치로 고려되어질 수 있다.

2. XDP에서의 디스크 입출력 제한

앞선 Redis의 지연 시간과 확장성 문제는 네트워크 측면에서 발생하는 문제로 eBPF XDP에서의 패킷 처리를 통해 효율적으로 개선될 수 있다고 설명하였다. 하지만 고성능 스토리지 시스템을 위해선 Linux 커널 네트워크 스택 최적화뿐만 아니라 스토리지 스택 최적화도 함께 이루어져야 한다. 따라서 XDP에서의 디스크 입출력 가능성 역시 함께 고려되어야 한다. XDP는 NIC 드라이버에 위치하여 초저지연 패킷 처리를 목표로 한다. 즉, XDP는

Linux 커널의 일반 프로세스 컨텍스트가 아닌 softIRQ 컨텍스트에서 실행된다. XDP가 softIRQ 컨텍스트에서 동작하는 경우 softIRQ 규칙에 따라 실행되어야 하므로 컨텍스트 실행 시간이 짧고, 논-블로킹(non-blocking) 형태의 작업만이 동작할 수 있다[44]. 즉, 긴 시간을 소모하며 블로킹 작업을 발생시키는 커널 프로세스 컨텍스트에서의 입출력 스케쥴링 작업은 실행될 수 없다. 따라서 XDP에서는 디스크 입출력을 호출하는 것이 적합하지 않다. 또한, XDP에서의 효과적인 패킷 처리를 가능하게 하기 위해선 XDP가 패킷을 처리한 후 해당 패킷을 Linux 커널 네트워크 스택으로 전달하는 것이 아닌 XDP_TX와 같은 명령어를 사용해 클라이언트에게 재전송 해야 한다[11]. 즉, XDP에서만 처리되는 패킷은 Linux 커널의 파일 시스템 영역을 지나지 못해 스토리지 작업을 가능하게 할 하위 서브 시스템과의 상호 작용이 불가능하다.

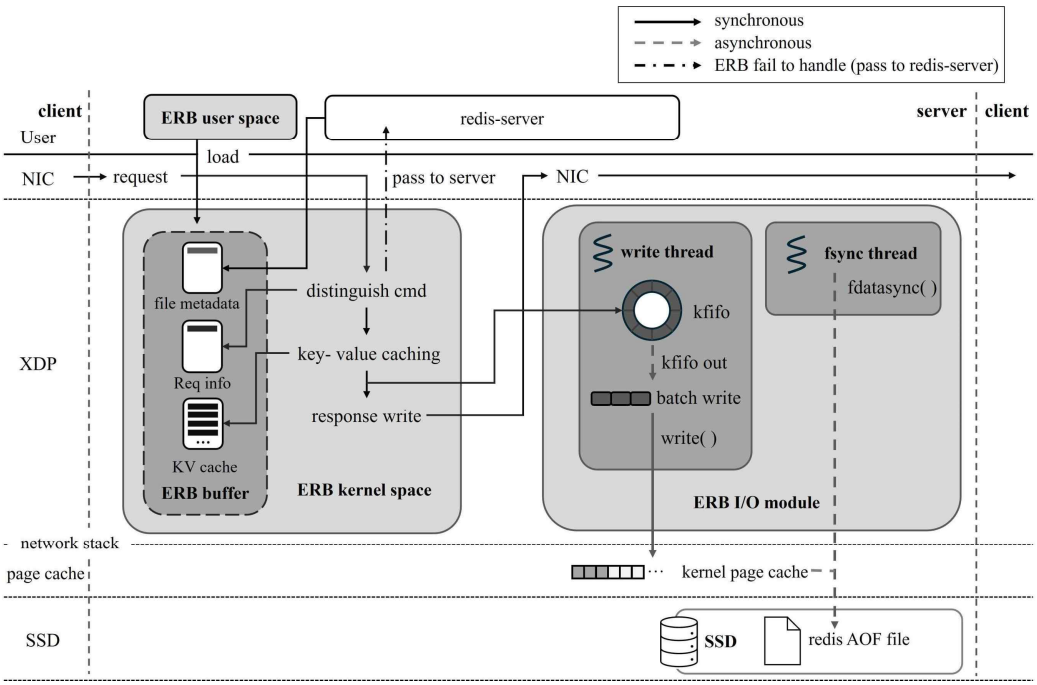
따라서, 본 논문은 XDP의 디스크 입출력 제한 문제를 해결하기 위해 XDP에서 직접적인 디스크 입출력을 요청하지 않고, kfifo 큐를 사용하여 커널 모듈에서의 비동기적인 디스크 입출력을 발생시킨다. 자세한 구현은 5.2 eBPF 커널 쓰기 작업에서 설명한다.

IV. 설계

본 장에서는 eBPF XDP에서의 고성능 패킷 처리와 커널 모듈을 사용한 비동기적 디스크 입출력을 통해 Linux 커널 네트워크 스택과 스토리지 스택의 오버헤드 모두를 효과적으로 감소시키는 ERB(eBPF-based Redis Booster)를 제안한다. ERB의 설계 원칙은 다음과 같다.

1. ERB 동작 과정은 저 수준(low-level)에서 진행된다. (XDP, 커널 모듈)
2. 사용자 영역의 어플리케이션과 Linux 커널 코드의 수정 없이 동작한다.

1. 전체 구조



[그림 8] ERB 전체 구조

그림 8은 ERB의 전체 구조로 네트워크 스택과 스토리지 스택을 통합한 입출력 프레임워크를 제공한다. ERB는 XDP에서 패킷을 빠르게 처리하여 불필요한 문맥 교환 오버헤드와 데이터 복사 오버헤드를 효과적으로 제거할 수 있다. 또한 커널 모듈에서의 비동기적인 디스크 입출력을 통해 사용자 체감 성능에서 스토리지 입출력 시간을 제외할 수 있다. ERB는 크게 ERB 사용자 영역(ERB user space)과 ERB 커널 영역(ERB kernel space), ERB 입출력 모듈(ERB I/O module)로 구성된다.

1) ERB 사용자 영역

개발자는 ERB를 사용하기 위해 XDP 기능 세팅, ERB 커널 영역 로직 검증 및 등록 등을 진행해야 한다. 해당 과정은 ERB 사용자 영역에 작성한 eBPF 사용자 코드를 통해 실행된다.

2) ERB 커널 영역

ERB 커널 영역은 NIC을 통해 전달받은 패킷의 키-밸류 값을 분석하고, 이를 ERB 캐시에 저장, Redis 클라이언트로 전송할 응답 패킷을 조작하며, 디스크 입출력을 위해 ERB 입출력 모듈로의 쓰기 버퍼(write buffer) 전달을 수행한다. 또한, ERB 커널 영역에는 eBPF 맵을 사용해 키-밸류 캐시 공간(KV cache), 로깅 파일 메타데이터 저장 공간 등을 유지하며, 이를 통해 효율적인 키-밸류 캐싱과 디스크 입출력을 위한 파일 정보 저장이 가능하다.

3) ERB 입출력 모듈

ERB 입출력 모듈은 XDP에서의 디스크 입출력 제한을 극복하기 위한 커널 모듈로 커널 페이지 캐시 작성과 디스크 입출력 작업을 수행한다.

2. 명령어 처리 과정과 ERB 입출력 모듈 동작 과정

ERB는 set, get 명령어를 지원하며 각 명령어의 처리 과정은 다음과 같다.

1) set 명령어 처리 과정

ERB 커널 영역은 Redis 서버 측 NIC에 수신된 패킷이 Redis 클라이언트로부터 전송된 것인지, ERB가 처리할 수 있는 명령어인지 분석한다. set 명령어의 경우 패킷에 저장된 키-밸류 값을 ERB 키-밸류 캐시인 KV cache에 저장해야 한다. 이를 위해 명령어 패킷의 키 값을 해싱 후 인덱싱하여 KV cache의 엔트리로 사용한다. set 명령어 내용을 저장할 KV cache 엔트리를 발견했다면 ERB 커널 영역은 키와 밸류 값을 KV cache 내부에 저장한다. 앞서 Redis everysec 로깅 방식은 명령어 단위의 커널 페이지 캐시 작성을 실행한다고 설명했다. 따라서 ERB 커널 영역은 KV cache에 저장한 set 명령어의 내용을 커널 페이지 캐시에 저장해야 한다. 이때 XDP의 구조적 문제로 인해 ERB 커널 영역은 직접적인 커널 페이지 캐시 작성에 제약을 받는다. 따라서 이를 해결하기 위해 페이지 캐시 작성을 ERB 커널 영역의 동작에서 ERB 입출력 모듈로 분리한다. 따라서 ERB 커널 영역에서 생성된 입출력 버퍼를 ERB 입출력 모듈로 전달한 뒤 즉시 Redis 클라이언트에게 전송할 응답 패킷을 조작한 후 송신한다.

2) get 명령어 처리 과정

Redis는 get 명령어 처리를 위해 패킷에 저장된 키 값을 기반으로 키-밸류 저장소를 탐색한 후 일치하는 키에 대한 밸류 값을 Redis 클라이언트에게 전송한다. 따라서 ERB 커널 영역은 명령어 패킷의 키 값을 해싱 후 인덱싱하여 KV cache의 엔트리로 사용한다. 이때, 응답 정확성을 위해 키 해싱 값만으로 KV cache 엔트리 일치 여부를 판단하지 않고 패킷에 저장된

키의 문자열과 KV cache 엔트리에 저장된 키의 문자열을 직접 비교해 정확히 일치하는 키에 대하여 밸류 값을 리턴한다. 이러한 방식으로 요청 키에 대한 밸류 값을 찾았다면 ERB 커널 영역은 즉시 응답 패킷을 조작하여 해당 밸류 값을 Redis 클라이언트로 전송한다.

3) 커널 페이지 캐시 작성과 디스크 입출력

커널 페이지 캐시 작성을 위한 *kernel_write()* 함수와 디스크 입출력을 위한 *fdatasync()* 함수는 ERB 커널 영역의 명령어 처리 과정과 비동기적으로 동작한다. 이를 위해 ERB 커널 영역의 set 명령어 처리 과정에서 *erb_kfifo_in()* kfuncs[16]를 사용해 ERB 커널 영역에서 생성된 쓰기 버퍼(write buffer)를 ERB 입출력 모듈 내부의 kfifo 자료구조로 전달한다. 이후 ERB 입출력 모듈은 해당 kfifo에 비동기적으로 접근하여 kfifo에 저장된 쓰기 버퍼를 순차적으로 읽고 *kernel_write()* 함수를 통해 페이지 캐시 작성을 실행한다. 이후 Redis AOF everysec 로깅 방식과 동일하게 동작하기 위해 매초 비동기적인 *fdatasync()*를 호출하여 디스크 입출력을 발생시킨다.

V. 구현

1. 스토리지 시스템을 위한 eBPF kfunc

eBPF는 eBPF verifier[8]의 엄격한 안전성 확인 과정을 만족시키며 효율적인 코드 작성을 위해 다양한 eBPF helper 함수를 제공한다[15]. XDP 역시 eBPF helper 함수를 사용한 효율적인 패킷 처리가 가능하다. 하지만 XDP에서 사용할 수 있는 eBPF helper 함수는 주로 네트워크 조작을 위해 제공되는 함수이며 스토리지 작업을 위한 함수는 제공되지 않는다. 따라서 ERB는 eBPF kfuncs를 사용해 스토리지를 위한 확장된 eBPF helper 함수를 제공하며 Linux 커널 소스 코드 수정 없이 eBPF kfuncs를 구현하기 위해 커널 모듈을 사용한다(ERB I/O module). 본 논문은 Redis 로깅 방식을 위해 Redis AOF 파일의 파일 디스크립터를 찾는 함수, ERB 입출력 모듈의 kfifo에 쓰기 버퍼를 저장하는 함수를 지원한다.

1) Redis 로깅을 위한 타켓 파일 탐색

Redis 로깅 방식은 Redis AOF 파일에 명령어 내용 즉, 쓰기 버퍼 내용을 계속해서 기록해 나가는 방식이다. 해당 방식을 Linux 커널 내부 동작 과정으로 살펴보자면, 커널 페이지 캐시에 디스크 입출력을 위한 쓰기 버퍼를 작성한 뒤 *fdatasync()*를 호출하여 페이지 캐시의 내용을 SSD로 내려 적는 과정으로 볼 수 있다. 이때 ERB 입출력 모듈에서 Redis 서버가 생성한 AOF 로깅 파일에 접근하여 커널 페이지 캐시 작성을 실행하기 위해선 해당 파일의 파일 디스크립터 값이 필수적이다. 이 과정에서 발생하는 문제는 크게 2가지가 있다. 우선 ERB 커널 영역과 ERB 입출력 모듈이 Redis 서버와 독립적으로 동작하는 문제가 있다. AOF 파일의 경우 Redis 서버에서 생

성되며 그 값은 Redis 서버의 전역 구조체 내부에 저장된다. 하지만 ERB 커널 영역과 ERB 입출력 모듈은 Redis 서버 외부에서 동작하기 때문에, Redis 서버 내부에 저장된 변수에 쉽게 접근할 수 없다는 한계가 발생한다. 다음으로 ERB 입출력 모듈과 Redis 서버가 서로 다른 프로세스에서 동작하는 문제가 있다. ERB 입출력 모듈이 인식한 AOF 파일의 파일 디스크립터는 Redis 서버 프로세스의 파일 디스크립터 테이블에 저장된 인덱스를 기반으로 하는 것이다. 즉, ERB 입출력 모듈이 동작하는 프로세스에서는 AOF 파일을 직접 *open()* 하지 않았기 때문에 해당 프로세스의 파일 디스크립터 테이블에는 AOF 파일의 파일 디스크립터가 존재하지 않을 가능성이 매우 크다. 이러한 문제를 해결하기 위해 ERB 입출력 모듈은 Redis 서버 초기화 시 eBPF kfuncs를 활용하여 AOF 파일의 파일 디스크립터를 ERB 커널 영역으로 전달하며, ERB 입출력 모듈에서 Redis 서버의 프로세스 아이디(PID)와 AOF 파일의 파일 디스크립터 기반으로 Redis 서버 프로세스에 저장된 AOF 파일의 파일 구조체를 참조하는 로직을 제공한다.

(1) 해결 방법

① Redis uretprobe

AOF 파일은 Redis 서버 초기화 시 *aofOpenIfNeededOnServerStart()* 함수 내부에서 생성된다. 이때 ERB 커널 영역은 eBPF uretprobe[53] 훅 방식을 사용하여 Redis AOF 파일의 파일 디스크립터를 참조한다. uretprobe 훅은 eBPF의 다양한 훅 방식 중 하나로 사용자 영역 어플리케이션에서 실행되는 함수의 종료 지점에 eBPF 함수를 연결하는 방식이다. 해당 지점에 연결된 ERB 함수는 내부적으로 *erb_find_fd()* kfuncs를 호출한다. 해당 kfuncs는 AOF 파일의 파일 디스크립터를 참조하기 위한 함수로 내부적으로 Redis 서버 프로세스의 파일 디스크립터 테이블을 순회하여 AOF 파일의 파일 디스크립터를 ERB 커널 영역 내부로 전달한다.

② AOF 파일 디스크립터 탐색 방법

Redis 서버 프로세스의 파일 디스크립터 테이블 내부를 순회하여 AOF 파일 디스크립터를 찾기 위해 *erb_find_fd()* kfuncs는 linux 커널의 /proc/PID/fd를 탐색한다. Redis 서버의 프로세스 아이디(PID)를 기준으로 한 /proc/PID/fd 내부에는 해당 프로세스에서 *open()* 한 파일들의 파일명과 파일 디스크립터 등이 저장되어 있다. 따라서 *erb_find_fd()* 함수는 /proc/PID/fd 내부를 순회하며 AOF 파일의 접미사와 일치하는 파일을 찾고 해당 파일의 인덱스인 파일 디스크립터를 리턴한다. Redis 서버는 각각 1개의 RDB 파일, AOF 파일, manifest 파일을 유지하며, 각 파일의 접미사는 “.rdb”, “.aof”, “.manifest” 이다. 따라서 Redids 서버 프로세스의 파일 디스크립터 테이블 내부에서 접미사가 “.aof“ 인 파일은 AOF 로깅 파일이라고 할 수 있다.

2) kfifo 자료구조를 위한 kfunc

XDP는 softIRQ 컨텍스트에서 실행되기 때문에 디스크 입출력을 호출할 수 없다. 따라서 ERB가 Redis의 AOF everysec 로깅 과정을 정확히 모방하기 위해선 softIRQ 컨텍스트를 사용하는 XDP 영역이 아닌 커널 컨텍스트를 사용하는 외부 영역에서 디스크 입출력을 호출해야 한다. 이를 위해 ERB는 페이지 캐시 작성과 디스크 입출력만을 담당하는 ERB 입출력 모듈을 사용한다. 하지만 AOF 파일에 작성될 쓰기 버퍼(write buffer)는 패킷이 처리되는 과정 즉, ERB 커널 영역의 명령어 처리 과정 중에 생성된다. 따라서 ERB 커널 영역은 AOF 파일에 작성될 쓰기 버퍼를 ERB 커널 영역에서 ERB 입출력 모듈 내부로 전달해야 한다.

(1) 문제점

ERB 커널 영역의 정보를 ERB 입출력 모듈로 전송하는 과정에서 발생하

는 문제는 다음과 같다. ERB 커널 영역과 ERB 입출력 모듈은 서로 다른 영역에 위치하기 때문에 ERB 입출력 모듈은 ERB 커널 영역이 생성한 정보에 직접 접근하고 해당 정보를 조작할 수 없다. 이러한 문제를 해결하기 위해 ERB 입출력 모듈은 kfifo 자료구조를 사용하며, ERB 커널 영역에서 *erb_kfifo_in()* kfuncs를 호출하여 쓰기 버퍼를 kfifo 자료구조에 순차적으로 저장한다.

(2) 해결 방법

kfifo는 Linux 커널이 제공하는 순환 버퍼로 FIFO(First In First Out) 구조를 따른다. *erb_kfifo_in()* 함수는 쓰기 버퍼, 쓰기 버퍼 길이, AOF 파일 디스크립터, Redis 서버 프로세스 아이디 등을 하나의 구조체로 변환하여 ERB 입출려 모듈의 kfifo 내부에 저장한다. 따라서 ERB 커널 영역은 *erb_kfifo_in()* 함수를 통해 ERB 입출력 모듈의 kfifo에 페이지 캐시 작성과 디스크 입출력을 위한 각종 정보만을 저장한 뒤 실질적인 디스크 입출력 과정에는 참여하지 않는다. 이후 ERB 입출력 모듈이 디스크 입출력를 실행하는 경우 ERB 커널 영역과의 추가적인 상호작용 없이 kfifo 자료구조에 저장된 구조체에 비동기적으로 접근하여 *kernel_write()*과 *fdatasync()*를 호출한다. ERB 입출력 모듈은 Redis 클라이언트로부터의 패킷을 처리하고 응답을 담당하는 ERB 커널 영역과 분리되어 비동기적으로 동작하기 때문에 Redis 클라이언트의 체감 지연 시간 중 스토리지 작업과 관련된 지연 시간을 효과적으로 제거한다. 만약 kfifo에 쓰기 버퍼가 가득 차는 경우 *erb_kfifo_in()* 함수를 통해 kfifo 내부에 쓰기 버퍼를 저장하려는 작업은 거부되며, 해당 패킷은 Redis 서버로 보내져 ERB가 아닌 Redis 서버에서 정상 처리된다.

2. ERB 입출력 모듈의 디스크 입출력 방식

ERB 입출력 모듈은 ERB 커널 영역과 독립적으로 동작하며 kfifo에 저장된 요소에 비동기적으로 접근하여 *kernel_write()*과 *fdatasync()*를 실행한다. 이때, 효율적인 동작을 위해 배치 방식을 사용하여 쓰기 작업을 실행하며, 커널 페이지 캐시 쓰기를 담당하는 write kthread와 SSD로의 플러시(flush)를 담당하는 fsync kthread로 분리되어 동작한다.

1) 배치 작업 방식

Redis 서버는 명령어 단위로 AOF 로깅을 실행한다. 따라서 ERB 커널 영역 역시 명령어 단위로 ERB 입출력 모듈의 kfifo 자료구조에 쓰기 버퍼 내용을 저장한다. 따라서 kfifo 자료구조 내부에는 쓰기 작업을 기다리는 쓰기 버퍼 들이 쌓이게 된다. 이때, kfifo에 저장된 요소를 1개씩 순차적으로 처리하는 것은 매우 비효율적인 작업이다. 따라서 ERB 입출력 모듈은 더욱 효율적인 쓰기 방식을 위해 여러 개의 쓰기 버퍼를 묶어 한 번에 처리하는 형태의 배치 쓰기 방식을 실행한다.

배치 쓰기 방식을 사용하기 위해선 kfifo에서 꺼낸 쓰기 버퍼들을 한 번에 묶을 임시 버퍼가 필요하다. 이러한 임시 버퍼는 ERB 입출력 모듈에 정적으로 선언 후 재활용하여 사용한다. 커널 페이지 캐시에 쓰기 버퍼 작성을 담당하는 write kthread는 kfifo에 접근하여 일정 개수의 쓰기 버퍼를 가져온다. 이때 가져올 쓰기 버퍼의 수는 정적으로 고정된 쓰기 버퍼 최대 개수와 kfifo에 남아있는 쓰기 버퍼 수를 비교하여 더 큰 값으로 결정된다. kfifo에서 쓰기 버퍼들을 꺼낸 후 write kthread는 임시 버퍼에 쓰기 버퍼 내용을 순차적으로 복사한다. 마지막으로 *kernel_write()* 호출 시 kfifo에서 가져온 쓰기 버퍼가 아닌 해당 내용을 복사해 둔 임시 버퍼를 넘김으로써 여러 개의 쓰기 버퍼를 한 번에 커널 페이지 캐시에 작성한다.

2) kernel_write()

커널 페이지 캐시 작성을 위해 *kernel_write()*을 호출하는 경우 쓰기 버퍼 외에도 버퍼 내용이 작성될 파일의 파일 구조체가 필수적이다. 이전에 설명한바 같이 ERB 입출력 모듈은 Redis 서버와 서로 다른 프로세스에서 실행되기 때문에 Redis 서버가 *open()* 한 AOF 파일의 파일 디스크립터 값을 참조하지 못하는 문제가 있었다. 이를 해결하기 위해 ERB 커널 영역은 *erb_find_fd()* kfuncs를 호출하여 AOF 파일의 파일 디스크립터를 리턴 받았다. 이렇게 리턴 받은 AOF 파일의 파일 디스크립터는 *kernel_write()*을 위한 AOF 파일 구조체 탐색에 사용된다. AOF 파일의 파일 구조체 탐색 역시 ERB 입출력 모듈이 실행되는 프로세스가 아닌 Redis 서버 프로세스를 참고해야 하는 점에서 AOF 파일의 파일 디스크립터를 찾는 과정과 유사하다. 해당 과정은 *kernel_write()*이 동작하기 직전에 실행되므로 ERB 입출력 모듈의 write kthread에서 동작한다. 파일 디스크립터 테이블은 각 프로세스의 테스크 관리 구조체(task struct)에 저장되어 있다. 따라서 Redis 서버 프로세스의 테스크 관리 구조체에 접근하기 위해 ERB 커널 영역으로부터 전달받은 Redis 서버의 프로세스 아이디(PID)를 사용하여 *pid_task()* 함수를 실행한다. 다음으로 해당 테스크 관리 구조체의 파일 디스크립터 테이블에서 AOF 파일 디스크립터 값을 기반으로 AOF 파일 구조체를 찾는다. 파일 디스크립터 테이블은 일반적으로 RCU(Read-Copy-Update) 보호를 받는다. 따라서 *__fget_files_rcu()* 함수를 통해 파일 디스크립터 테이블 내부의 파일 구조체를 찾는 경우 해당 함수 앞뒤로 *rcu_read_lock()*, *rcu_read_unlock()* 함수를 사용하여 RCU 보호를 받는 데이터에 안전하게 접근해야 한다. 이러한 방법을 사용하는 경우 다른 프로세스에서 *open()* 된 파일의 파일 구조체에 안정적으로 접근할 수 있다. AOF 파일의 파일 구조체를 찾았다면 파일 오프셋 동시성 문제를 해결하기 위해 해당 파일 구조체의의 *f_pos_lock*에 *m*

*utex_lock()*을 사용한다. 이후 *kernel_write()*을 호출하여 파일 무결성을 보장하면서 동시에 XDP 외부에서의 안정적인 커널 페이지 캐시 작성을 수행한다.

3) fdatasync()

Redis 서버는 3가지 종류의 AOF 방식을 지원한다. 이 중 성능과 효율 측면에서 가장 선호되는 everysec 방식의 경우 초당 1번의 *fdatasync()*가 호출된다. 이 경우 매초 한 번의 디스크 입출력이 발생하여 *kernel_write()*을 통해 커널 페이지 캐시에 저장되었던 쓰기 버퍼의 내용이 SSD에 내려 적히게 된다. Redis는 everysec 방식을 사용하는 경우 성능 향상을 위해 *fdatasync()* 작업을 Redis 메인 스레드가 아닌 백그라운드 입출력 스레드(bio-threads)에서 실행한다. ERB 입출력 모듈 역시 해당 방식과 동일한 형태로 구현하기 위해 *fdatasync()*를 실행하는 fsync kthread를 생성하여 커널 페이지 캐시 작성을 담당하는 write kthread와 분리한다. *fdatasync()*를 실행하기 위해 사용되는 *vfs_fsync()* 함수는 SSD로의 플러시(flush)를 진행할 파일의 파일 구조체가 필수적이다. 이때 ERB 입출력 모듈이 Redis 서버의 AOF 파일 구조체를 정확히 참조하지 못하는 문제가 발생한다. 해당 문제는 앞선 5.2.2 kernel_write() 파트에서 발생하는 문제와 동일하며 해결 방법 역시 같다. 따라서 Redis 서버의 테스크 관리 구조체를 기반으로 파일 구조체를 탐색하는 방법을 동일하게 사용하여 AOF 파일 구조체를 찾고 *vfs_fsync()*를 호출하여 *fdatasync()*를 실행한다. 이때 추가 조건 없이 매초 *fdatasync()*를 실행한다면 커널 페이지 캐시에 작성된 내용이 없음에도 불구하고 디스크 입출력을 시도하여 불필요한 오버헤드가 추가되는 문제가 발생한다. 이러한 문제를 해결하기 위해 ERB 입출력 모듈은 플러시 플래그(fsync flag)를 사용한다. *kernel_write()*를 실행하는 경우 커널 페이지 캐시에는 AOF 파일에 대한 새로운 데이터가 작성된다. 이는 SSD로 플러시 할 데이터가 생성된 것이다. 따라서 write kthread에서 *kernel_write()*이 발생하

는 경우 플러시 플래그를 1로 변경하여 fsync kthread에게 *fdatasync()* 호출이 가능함을 알린다. fsync kthread는 매초 깨어나 플러시 플래그를 인지하고 *fdatasync()*를 호출하기 직전 해당 플래그 값을 0으로 변경한다. *fdatasync()* 호출 전 플러시 플래그 값을 초기화하는 경우 *fdatasync()* 과정 중 발생한 *kernel_write()*을 인지할 수 있다. 이렇게 플러시 플래그를 사용하는 경우 불필요한 디스크 입출력을 줄이고 효율적인 쓰기 메커니즘을 사용할 수 있다.

3. eBPF 전송 제어 프로토콜과 해시 충돌

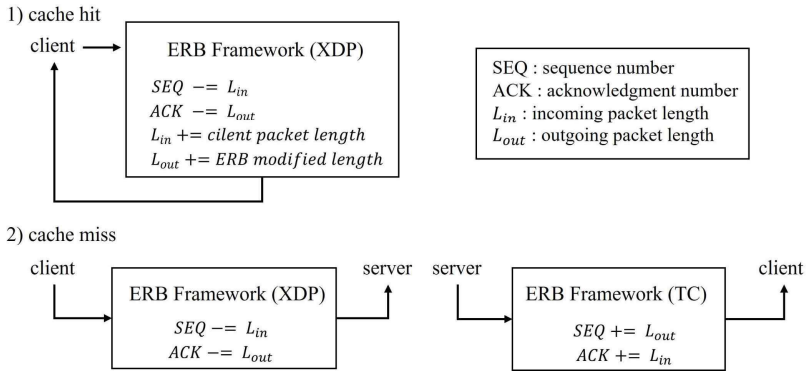
1) eBPF 전송 제어 프로토콜(TCP protocol)

XDP에서 패킷을 처리하여 효율적인 성능 향상을 달성한 기존 연구들은 사용자 데이터그램 프로토콜(UDP protocol)을 사용한 패킷 처리만을 다룬다 [14, 38, 39]. 하지만 Redis는 기본적으로 전송 제어 프로토콜(TCP protocol)만을 제공하며, 해당 프로토콜은 신뢰성 있는 데이터 전송을 기반으로 한 보편적인 프로토콜이다. 따라서 본 논문은 eBPF에서의 전송 제어 프로토콜 지원을 위해 eBPF 맵을 사용해 패킷 순서 번호(Sequence number)와 확인 응답 번호(Acknowledgment number)를 수정하는 Middlecache[45]를 참고하여 XDP에서의 패킷 처리에 전송 제어 프로토콜을 지원한다. 그림 9는 ERB에서 전송 제어 프로토콜 지원을 위해 XDP와 TC에서 패킷의 순서 번호와 확인 응답 번호를 수정하는 방식이다.

2) 해시 충돌

FNV-1A 해시는 고른 해시 분포를 위해 자주 사용된다. 하지만 FNV-1A 해시를 사용하더라도 캐시 엔트리 크기 제한으로 인해 32 비트의 해시 값

전체를 사용하지 못하고 일부 비트 값을 인덱싱하여 사용하는 경우 심각한 해시 충돌 문제가 발생할 가능성이 있다. 따라서 본 논문은 효율적인 캐시 공간 활용을 위해 이중해싱을 사용하여 해시 충돌을 최대한 예방한다. 이중 해싱은 기본적으로 2개의 해시 함수를 사용한다. 우선 첫 번째 해시의 경우 FNV-1A 해싱을 사용하여 32 비트 해시 값을 계산한다. 다음으로 해당 해시 값을 KV cache의 엔트리 수로 나머지 연산하여 KV cache에서 사용될 해시 인덱스를 추출한다. 만약, 이때 해시 충돌이 발생한다면 두 번째 해시를 사용한다. 두 번째 해시는 FNV-1A 해시를 사용해 추출된 값을 미리 지정한 HASH_MOD 값으로 나머지 연산하여 계산한다. HASH_MOD 값은 주로 소수와 2의 제곱수로 사용한다. 이렇게 구해진 두 번째 해시값은 개방 주소법 해싱 방식을 위한 인덱스 값으로 사용된다. 따라서 첫 번째로 구한 FNV-1A의 해시 값에 두 번째로 구한 인덱스 값을 연산하여 비어있는 KV cache 엔트리 위치를 계산한다. 단, eBPF verifier[8]의 명령어 수 제약으로 인해 무한 루프를 사용한 개방 주소법 해싱은 불가능하다. 따라서 정해진 수의 루프 카운트 이내에 비어있는 엔트리를 찾지 못한다면 해당 패킷은 Redis 서버로 전송되어 Redis 서버의 키-밸류 캐시에 저장된다.



[그림 9] ERB의 전송 제어 프로토콜 지원 방식

VI. 실험

본 장에서는 ERB를 사용해 타겟 워크로드의 전송 제어 프로토콜 요청을 처리할 때 발생하는 성능 향상과 지연 시간 감소, 코어 수에 따른 성능 확장성 등에 대해 평가한다. 실험은 마이크로 벤치마크인 redis-benchmark와 매크로 벤치마크인 YCSB로 나누어 진행한다.

1. 실험 환경

해당 실험은 총 2개의 머신을 사용하며 하나는 Redis 서버와 ERB가 실행될 서버 머신, 하나는 클라이언트 머신이다. 표 2는 서버 머신의 스펙, 표3은 클라이언트 머신의 스펙을 나타낸다.

[표 2]

서버 머신 환경

OS	Ubuntu 22.04
Kernel	Linux 6.6.0
CPU	2 X Intel(R) Xeon(R) Gold 6342 2.80GHz
Memory	DDR4 128G
NIC	Mellanox ConnectX-5
SSD	Intel Optane P5800X

서버 머신의 경우 2개의 NUMA 노드를 가진 Intel Xeon Gold 6342 CPU를 사용하며 각 NUMA 노드는 24개의 코어를 가진다. 따라서 해당 실험은 NUMA 지역성을 고려하여 최대 24개의 코어만을 사용한다. 그리고 128G의 메모리를 사용하며 100 Gbps의 Mellanox ConnectX-5 NIC을 사용한다. AOF 로깅 파일을 저장할 SSD의 경우 Intel Optane P5800X를 사용한다. 운

영체제와 Linux 커널은 각각 Ubuntu 22.04와 Linux-6.6.0을 사용한다.

[표 3]

클라이언트 머신 환경

OS	Ubuntu 22.04
Kernel	Linux 6.6.0
CPU	2 X Intel Xeon Gold 6338 CPU 2.00GHz
Memory	DDR4 128G
NIC	Mellanox ConnectX-5

클라이언트 머신의 경우 Intel Xeon Gold 6338 CPU를 사용하며 나머지 머신 환경은 서버 머신과 동일하다. 서버와 클라이언트의 NIC은 직접 연결하여 사용한다. 모든 실험은 하이퍼스레딩을 사용하지 않으며 cpu 거버너(governor) 설정은 performance로 고정 후 진행한다. 또한 클라이언트의 요청을 서버 머신의 NIC 패킷 처리 코어에 효율적으로 분산시키고 고속의 패킷 처리를 위해 NIC의 멀티 큐(multi-queue)와 RSS(Receive Side Scaling)를 조정하였다. NIC의 멀티 큐 기능을 사용하는 경우 수신 큐(Rx queue)와 송신 큐(Tx queue)의 각 쌍별로 동일한 코어를 맵핑하지 않는다면 하나의 패킷이 송수신될 때 서로 다른 코어를 통해 처리될 가능성이 발생하며 이는 캐시 지역성을 고려하지 못한 것으로 성능 하락의 여지가 있다. 앞서 NUMA 지역성을 위해 서버 머신은 최대 24개의 코어를 사용한다고 하였다. 따라서 멀티 큐 역시 최대 24개 코어를 사용한다.

2. 마이크로 벤치마크 실험

본 논문은 마이크로 벤치마크 실험을 위해 redis-benchmark를 사용한다. 표 3은 redis-benchmark 실험 세팅을 나타낸다.

[표 4]

redis-benchmark 실험 세팅

Key size	16 bytes
Value size	100 bytes
Key population	10 million
Client	100
Redis-benchmark threads	100
Max number of redis application threads	24
Max number of RX cores	24

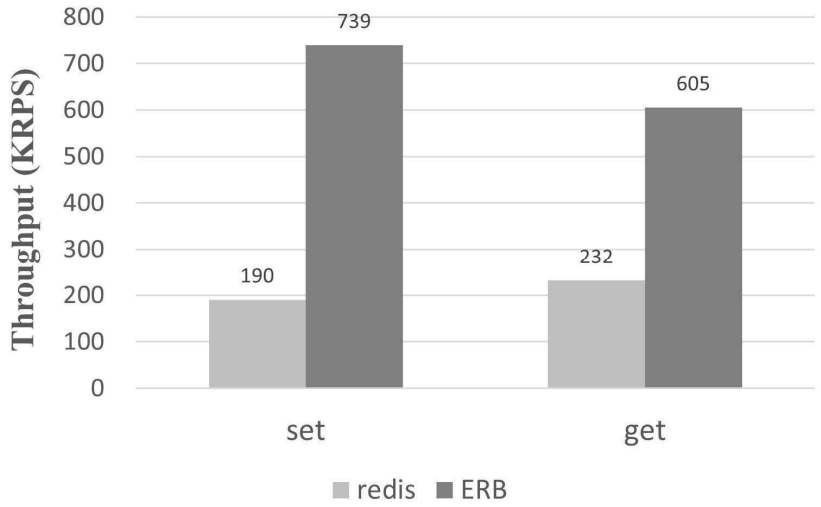
redis-benchmark 실험 시 키 크기는 16 바이트, 밸류 크기는 100 bytes를 사용하며 총 10 만개의 요청을 전송한다. redis-benchmark의 클라이언트 수는 총 100명이며 redis-benchmark가 성능 병목 지점이 되는 것을 막기 위해 redis-benchmark 스레드를 100개로 할당하였다. ERB의 캐시 크기는 실험에서 생성되는 전체 요청을 모두 저장할 수 있는 10 만개로 고정하며 Redis 서버를 위해 할당하는 코어 수와 ThreadedIO를 위한 네트워크 스레드 수는 최대 24개, ERB를 위해 할당되는 코어 수 역시 최대 24개를 사용한다.

1) 지연 시간 평가

Redis는 빈번한 문맥 교환 오버헤드와 데이터 복사 오버헤드로 인해 성능 하락을 겪고 있다. 따라서 ERB를 사용하는 경우 Linux 커널 네트워크 스택의 비효율적인 오버헤드를 줄이고 높은 성능 향상을 달성할 수 있는지 확인하기 위해 지연 시간 측면에서의 성능 평가 실험을 진행한다. 본 실험의 경

우 Redis 서버와 ERB 모두 단일 코어로 세팅하여 코어 수 확장으로 인한 성능 향상 가능성을 배제하였다.

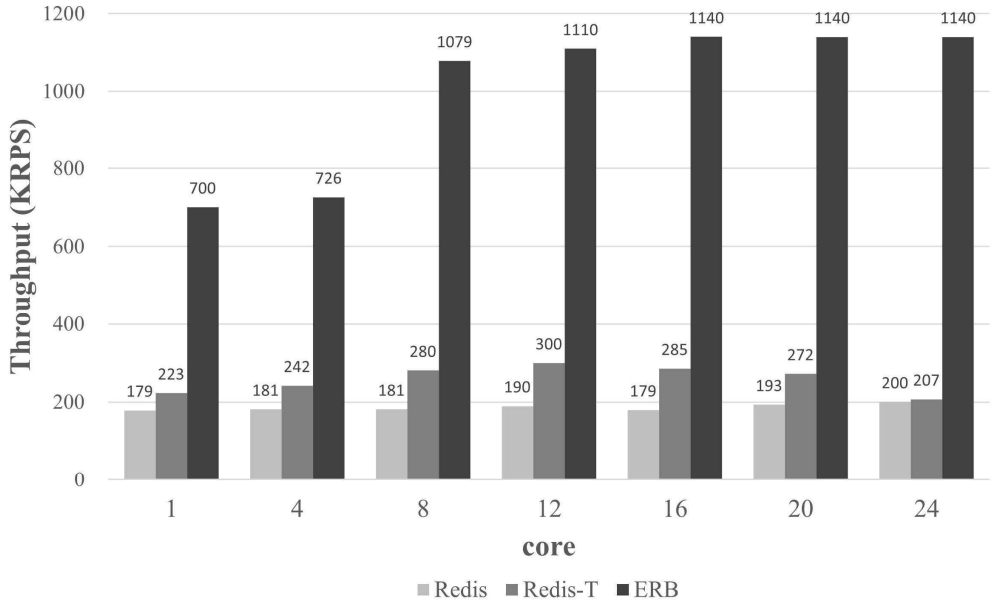
그림 10은 지연 시간 실험의 측정 결과이다. 1개의 코어만을 사용하는 경우 기존 Redis의 set 성능은 약 190 KRPS, get 성능은 약 232 KRPS를 기록하였다. 반면 1개의 코어만을 사용했을 때 ERB의 set 성능은 약 739 KRPS, get 성능은 약 605 KRPS를 기록하였다. 이는 기존 Redis 성능과 비교하여 set 성능은 약 3.9배, get 성능은 2.6배 향상된 수치이다. 이를 통해 ERB를 사용하는 경우 Linux 커널의 네트워크 스택에 진입하기 전 XDP에서의 패킷 처리가 가능해 불필요한 문맥 교환 오버헤드와 데이터 복사 오버헤드로 인한 성능 하락을 효과적으로 해결할 수 있음을 확인하였다. 또한 Redis 클라이언트로부터 수신된 패킷을 처리 후 응답 패킷을 재전송하기까지의 과정에서 페이지 캐시 작성과 디스크 입출력 동작을 분리하고 비동기적으로 실행해 Redis 클라이언트의 체감 응답 시간을 효과적으로 줄일 수 있음을 확인하였다.



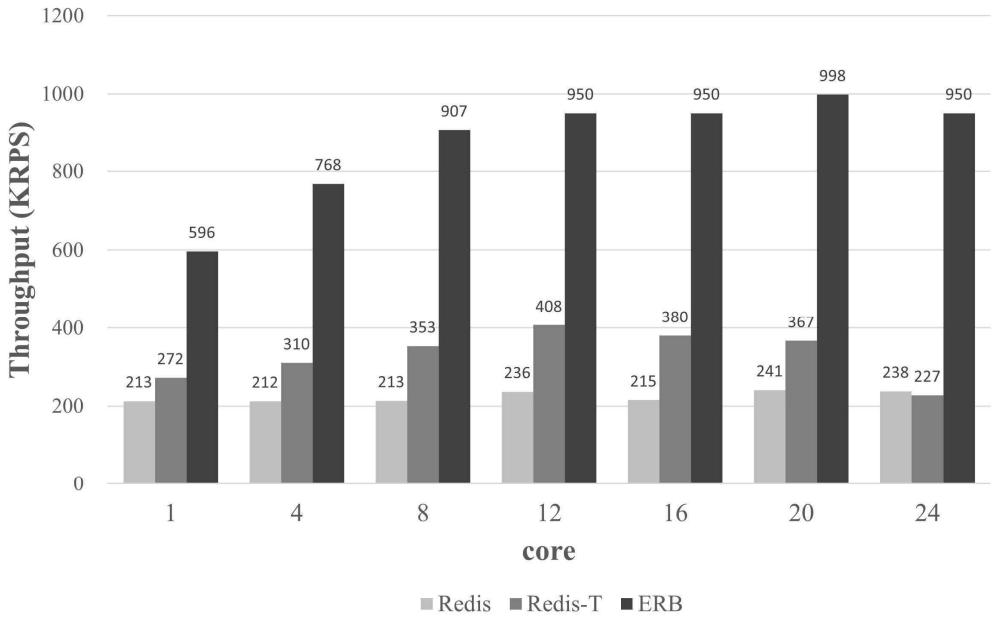
[그림 10] ERB redis-benchmark 지연 시간

2) 확장성 평가

해당 실험은 ERB와 redis에 1개부터 24개까지의 코어 수를 할당하며 코어 수 증가에 따른 ERB의 성능 확장성을 평가하였다. 그림 11은 redis-benchmark에서 set 명령어를 실행했을 때의 결과이며, 그림 12는 get 명령어를 실행했을 때의 결과이다. Redis는 Redis 서버의 server_cpulist 값을 조정하여 서버 사용에 할당된 코어 수를 조정하였으며, Redis-T는 Redis 성능을 최적화 한 것으로 Redis 서버에 할당된 코어 수와 ThreadedIO를 사용해 네트워크 입출력에 할당한 스레드 수를 동일하게 맞춘 것이며, ERB는 ERB에 할당한 코어 수를 조정한 것이다. 해당 실험의 베이스라인은 Redis이며 Redis-T와 ERB의 성능 확장성 결과는 베이스라인의 성능과 비교했을 때의 성능 향상 정도로 평가한다.



[그림 11] redis-benchmark set 명령어 성능 확장성



[그림 12] redis-benchmark get 명령어 성능 확장성

그림 11의 set 명령어 실험 결과 Redis-T는 코어 수가 확장되는 경우 베이스라인에 비해 최소 1.2에서 최대 1.6배의 성능 향상을 보이며, ERB는 코어 수가 확장됨에 따라 최소 4배에서 최대 6.4배의 성능 향상을 보인다. 그림 12의 get 명령어 실험 결과 Redis-T는 베이스라인에 비해 최소 1.2에서 최대 1.7배의 성능 향상을 보이며, ERB는 최소 2.8에서 최대 4.1배의 성능 향상을 보인다. 모든 경우 베이스라인인 Redis의 성능은 단일 스레드로 진행되는 명령어 처리 과정이 병목 지점으로 작용 되어 큰 성능 향상을 이루지 못했음을 알 수 있다. 이를 통해 ERB를 사용하는 경우 코어 수가 확장됨에 따라 분산적인 패킷 처리가 가능해 효율적으로 지연 시간을 감소시키고 따라서 높은 성능 확장성을 달성함을 확인할 수 있다.

3. 매크로 벤치마크 실험

본 논문은 매크로 벤치마크 실험을 위해 YCSB를 사용한다. 표 3은 YCSB 실험을 위한 세팅을 나타낸다.

[표 5]

YCSB 실험 세팅

Key size	20 bytes
Value size	100 bytes
Key population	10 million
Distribution	uniform
Client	100
Max number of redis application threads	20
Max number of RX cores	20

YCSB 실험 시 키 크기는 20 바이트, 밸류 크기는 100 바이트를 사용했으며 총 10 만개의 요청을 전송한다. 그리고 키 분산은 uniform으로 사용한다. 앞선 실험에서 ERB와 Redis 모두 20개 이상의 코어를 할당하는 경우 성능이 포화 됨을 확인하였다. 따라서 매크로 벤치마크의 실험은 최대 할당 코어를 20개로 제한한다. 현재 ERB의 KV cache는 자주 접근되는 데이터(hot data)와 자주 접근되지 않는 데이터(cold data)를 구분하고 불필요한 데이터를 방출시키는 데이터 방출 정책(eviction policy)을 제공하지 않는다. 따라서 이 경우 자주 접근되는 데이터에 주된 요청을 보내는 zifian 분포는 실험 목적에 적합하지 않기 때문에 uniform 분포를 사용해 실험을 진행하며 앞선 마이크로 벤치마크 실험과 동일하게 캐시 크기의 경우 YCSB가 요청하는

명령어 수를 전부 저장할 수 있는 10 만개로 고정한다. 또한 앞선 실험을 통해 ThreadedIO를 실행하지 않은 Redis의 경우 코어 수 확장에 따른 성능 향상이 거의 발생하지 않음을 확인하였다. 따라서 본 실험에서는 ThreadedIO를 실행한 Redis-T와 ERB 사이의 성능 비교만을 진행한다. 표 6는 YCSB 실험에서 사용할 워크로드 별 특성을 나타낸 표이다.

[표 6]

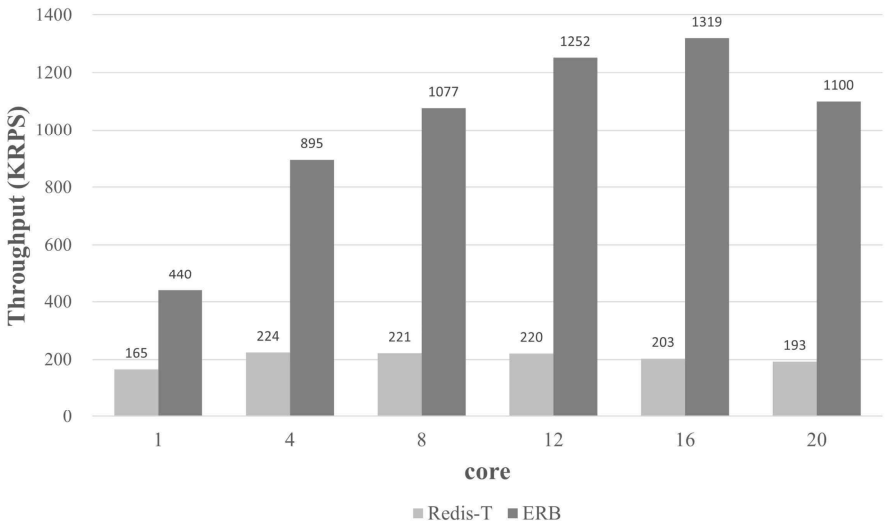
YCSB 워크로드

workload A	update	50%
	read	50%
workload B	update	5%
	read	95%
workload C	update	0%
	read	100%

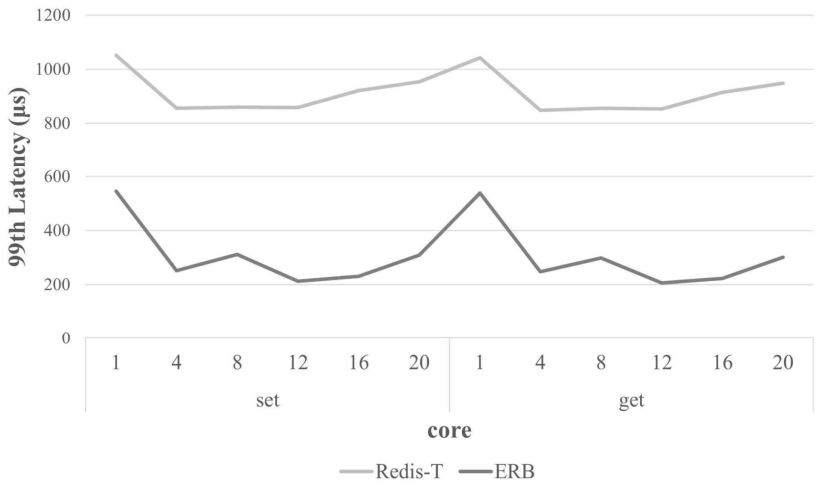
1) 워크로드 A (update 50%, read 50%)

그림 13과 그림 14는 각각 타켓 워크로드로 워크로드 A(set 50%, get 50%)를 실행했을 때 Redis-T와 ERB의 성능 및 99번째 백분위수(꼬리 지연 시간) 응답 시간 측정 결과이다. 실험 결과 ERB는 Redis-T에 비해 최소 2.7배에서 최대 5.9배 나은 성능을 보였다. 또한 set 명령어에 대한 99번째 백분위수 응답 시간을 최소 1.9배에서 최대 2.8배까지 감소시키며, get 명령어에 대한 99번째 백분위수 응답 시간을 최소 2배에서 최대 4.1배까지 감소시켰다. update를 자주 호출하는 워크로드 A의 특성상 잦은 *write()* 명령어가 호출된다. 이때 Redis-T는 주기적인 *write()* 시스템 콜과 *fdatasync()* 호출이 지연 시간에 큰 영향을 끼쳐 비교적 낮은 성능이 측정되었다. 반면 ERB

의 경우 스토리지 처리와 네트워크 명령어 처리가 분리되어 동작하기 때문에 잦은 *write()* 시스템 콜 호출로 인한 성능 악화가 클라이언트 체감 지연 시간에 영향을 끼치지 않는다.



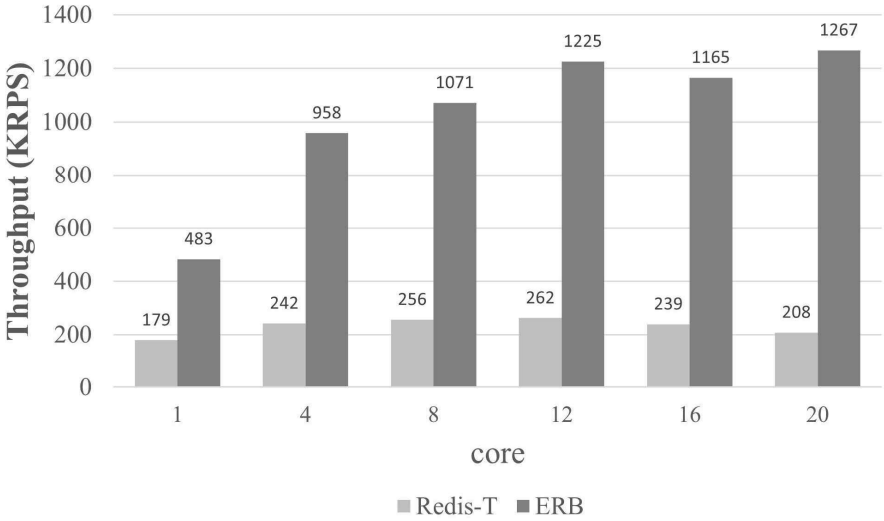
[그림 13] YCSB 워크로드 A 성능



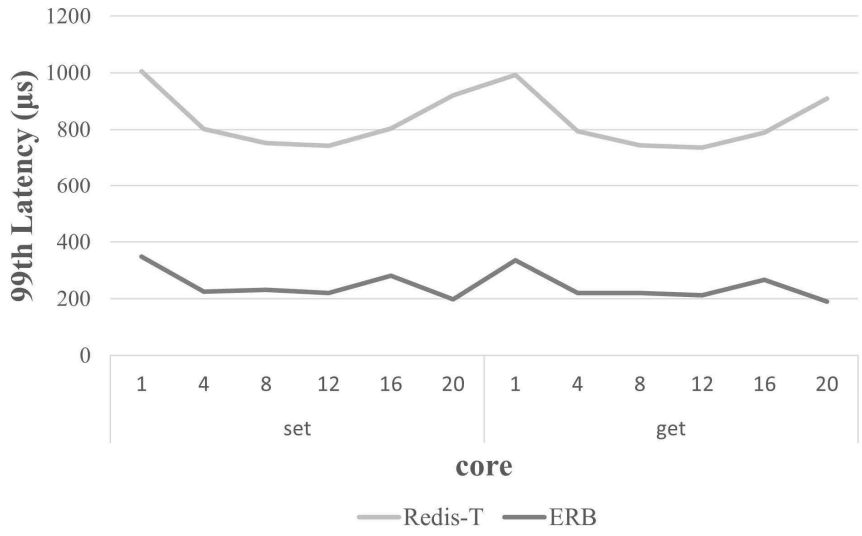
[그림 14] YCSB 워크로드 A 99번째 백분위수 응답 시간

2) 워크로드 B (update 5%, read 95%)

그림 15와 그림 16는 각각 타겟 워크로드로 워크로드 B(set 5%, get 95%)를 실행했을 때 Redis-T와 ERB의 성능 및 99번째 백분위수 응답 시간 측정 결과이다. 실험 결과 ERB는 Redis-T에 비해 최소 2.7배에서 최대 6.1배 나은 성능을 보였다. 또한 set 명령어에 대한 99번째 백분위수 응답 시간을 최소 2.8배에서 최대 4.7배까지, get 명령어에 대한 99번째 백분위수 응답 시간을 최소 3배에서 최대 4.8배 감소시켰다. 주로 읽기 명령어가 실행되는 워크로드 B의 특성상 대부분의 명령어는 키-밸류 저장소에 저장된 밸류 값을 탐색하는 hgetall 명령어가 호출된다. 이때 Redis-T의 경우 코어 수 확장에 따른 효율적인 패킷 병렬 처리와 적은 횟수의 *write()* 시스템 콜 호출을 통해 워크로드 A에서 측정 된 성능보다는 더욱 개선된 성능을 보인다. 하지만 Linux 커널 네트워크 스택에서 발생하는 막대한 문맥 교환 오버헤드와 데이터 복사는 제거되지 않아 성능에 심각한 악영향을 미친다. 반면 ERB의 경우 대부분의 패킷이 XDP에서 처리되어 Redis-T가 겪는 심각한 문맥 교환 오버헤드를 획기적으로 줄일 수 있다. 또한 5%의 hmset 쓰기 명령어에 대해서도 앞선 워크로드 A의 결과 분석과 동일하게 클라이언트 체감 응답 시간에 영향을 주는 네트워크 패킷 처리 작업과 비동기적인 쓰기 작업을 분리 실행하여 워크로드 B 실험 결과에 큰 영향을 미치지 않는다.



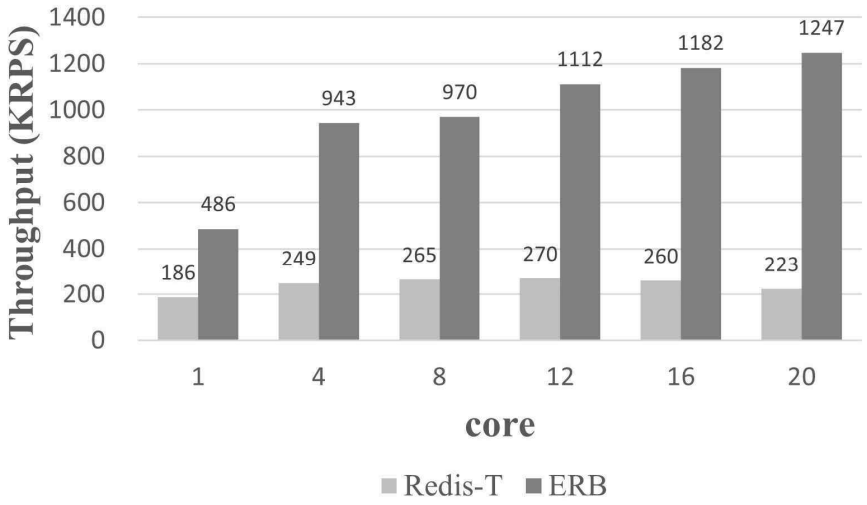
[그림 15] YCSB 워크로드 B 성능



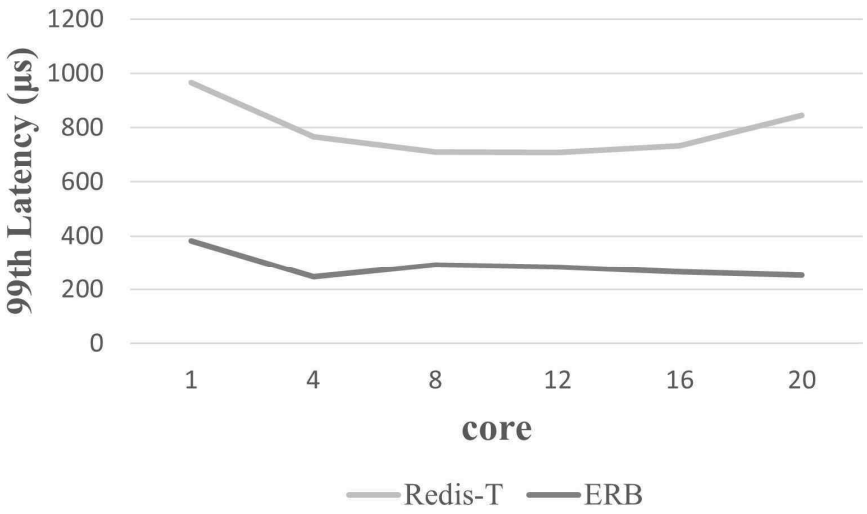
[그림 16] YCSB 워크로드 B 99번째 백분위수 응답 시간

3) 워크로드 C (update 0%, read 100%)

그림 17와 그림 18는 각각 타겟 워크로드로 워크로드 C(set 0%, get 100%)를 시행했을 때 Redis-T와 ERB의 성능 및 99번째 백분위수 응답 시간 측정 결과이다. 성능 측정 결과 ERB는 Redis-T에 비해 최소 2.6배에서 최대 5.6배 나은 성능을 보였다. 또한 get 명령어에 대한 99번째 백분위수 응답 시간을 최소 2.5배에서 최대 2.7배 감소시켰다. 읽기 작업만 호출하는 워크로드인 워크로드 C의 특성상 모든 명령어로 hgetall 명령어가 호출된다. 따라서 해당 실험은 명령어 처리 과정에서 발생하는 스토리지 스택의 오버헤드는 고려하지 않고 Linux 커널 네트워크 스택 오버헤드의 효율적 제거 측면에 초점을 맞춘다. Redis-T의 경우 네트워크 스택 오버헤드와 더불어 스토리지 스택 오버헤드까지 더해진 워크로드 A와 워크로드 B에 비해서 비교적 더욱 향상된 성능을 보인다. ERB 역시 경우 대부분의 hgetall 명령어를 XDP에서 처리하기 때문에 Redis-T의 문맥 교환 오버헤드를 획기적으로 줄여 최상의 성능을 달성한다.



[그림 17] YCSB 워크로드 C 성능



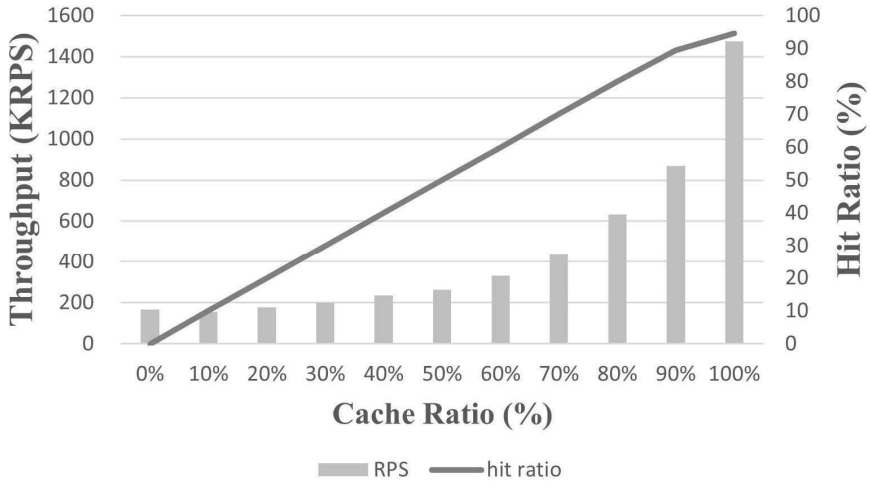
[그림 18] YCSB 워크로드 C 99번째 백분위수 응답 시간

4) ERB 캐시 크기 별 캐시 적중률(hit ratio)

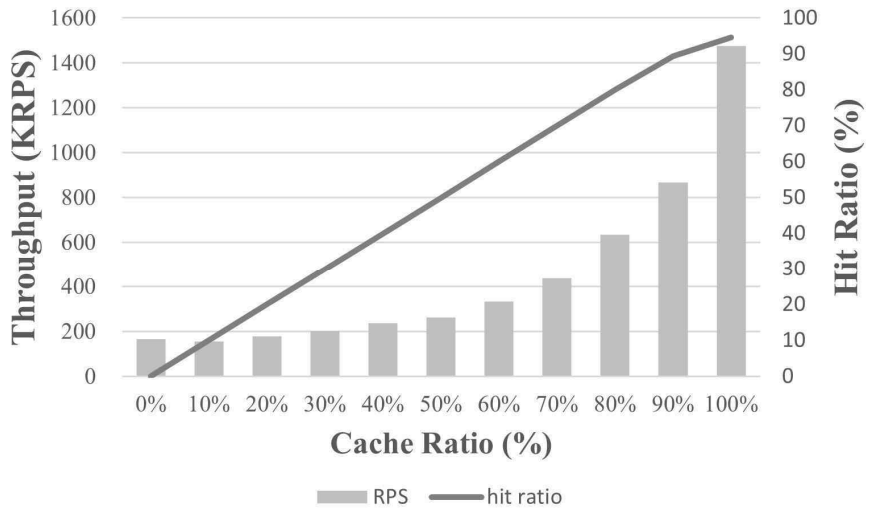
해당 실험은 ERB 캐시 크기에 따른 캐시 적중률과 성능 비교 실험이다.

해당 실험을 위해 YCSB 워크로드 A를 사용하였으며, ERB 캐시의 크기는 YCSB 요청 명령어 수의 10%에서 100% 비율로 확장하였다. 베이스라인인 ERB 캐시 크기 0%는 코어 24개를 사용한 Redis의 성능이다. 그림 19는 ERB set 명령어에서의 캐시 크기 증가에 따른 캐시 적중률 및 성능이며, 그림 20은 get 명령어에서의 캐시 크기 증가에 따른 캐시 적중률 및 성능 결과이다. 두 실험의 성능 결과는 동일하며 명령어에 따른 캐시 적중률에만 차이가 발생한다.

해당 실험 결과를 통해 ERB 캐시 크기가 증가함에 따라 캐시 적중률이 선형적으로 증가함을 알 수 있다. 이는 이중해싱을 사용한 해시 충돌 방지가 효율적으로 작용하여 주어진 ERB 캐시 공간을 낭비 없이 사용하기 때문이다. 성능의 경우 로그 형태로 증가함을 알 수 있다. 이는 개방 주소법을 사용하는 경우 캐시 공간이 다 찼음에도 비어있는 공간을 찾기 위해 이중해싱을 사용하는 과정에서 발생하는 오버헤드가 ERB 성능에 영향을 끼치기 때문이다. 따라서 캐시 크기가 작은 경우 이러한 오버헤드가 더욱 크게 영향을 끼쳐 비교적 성능이 낮게 나타나고, 캐시 크기가 충분한 경우 이러한 오버헤드가 적게 발생해 큰 성능 향상을 달성한 것이다. 나아가 캐시 공간이 0%인 경우 즉, 베이스라인인 Redis의 성능이 10%의 캐시 공간을 사용한 ERB의 성능 보다 약 4% 더 높음을 알 수 있다. 이 역시 앞선 개방 주소법 사용 시 캐시 공간을 파악하지 못해 발생하는 추가적인 오버헤드가 ERB 성능에 악영향을 미쳤기 때문이다. 추후 자주 접근되는 데이터(hot data)와 그렇지 않은 데이터(cold data)를 구분하고 불필요한 데이터를 방출시키는 효율적인 데이터 방출 정책(eviction policy)을 지원하며, 캐시 공간 활용률을 효율적으로 파악한다면 zifian 분포의 실험을 진행하는 경우 더 작은 캐시 공간에서도 높은 성능과 캐시 적중률을 달성할 수 있을 것이다.



[그림 19] ERB 캐시 크기 변화에 따른 set 명령어 성능



[그림 20] ERB 캐시 크기 변화에 따른 get 명령어 성능

VII. 관련 연구

1. 커널 우회 방식(Kernel-Bypass)

다양한 라이브러리와 운영체제들은 입출력 처리 과정에서 발생하는 커널의 각종 오버헤드를 줄이기 위해 사용자가 직접 입출력 장치에 접근할 수 있도록 지원한다[18, 26-32]. 스토리지 입출력을 위한 커널 우회 기술(Kernel-Bypass)의 대표적인 기술에는 Intel의 SPDK[29]가 있다. SPDK는 커널 우회를 통한 고성능 스토리지 기법으로 기존 스토리지 구조의 입출력 병목 현상을 해결하기 위해 사용자 공간에 스토리지 드라이버를 구축하여 커널 공간을 거치지 않고도 빠른 스토리지 입출력을 가능하게 한다. 다만, SPDK를 사용하기 위해선 사용자가 자체적인 파일 시스템을 구축하고 스토리지 입출력 완료에 대해 폴링(polling) 요청을 구현해야 하는 등 개발 복잡성이 매우 높은 문제가 있다. 다음으로 네트워크 입출력을 위한 커널 우회 방식에는 Demikernel[33], TAS[34], mTCP[35] 등이 있다. 해당 방식은 기존 입출력 경로에서 커널을 제외하여 커널의 네트워크 스택에서 발생하는 오버헤드를 효과적으로 제거한다. 이러한 구조는 보통 바쁜 대기 폴링(busy polling)을 사용하며 높은 성능을 제공하는 대신 보안 위험성과 유지 보수에 어려움이 따르는 문제가 있다.

2. 네트워크를 위한 eBPF

과거 eBPF는 패킷 필터링[7], 로드 밸런싱[36], 모니터링[37]에 사용되었다. 하지만 최신 연구들은 eBPF를 단순 모니터링 도구가 아닌 다양한 연산

을 오프로드(offload) 시키는 강력한 도구로 사용하고 있다. BMC[38]는 Memcached의 get 연산 최적화하기 위해 XDP와 TC 훅에서 eBPF를 호출한다. 이 경우 수신된 패킷이 커널 네트워크 스택으로 전달되지 않고 NIC 드라이버에서 처리되기 때문에 매우 높은 성능을 달성할 수 있다. Electrode[39]는 Multi-Paxos[40] 프로토콜의 메시지 전달, 상태 업데이트 등의 작업을 커널 공간에서 직접 처리하여 사용자 영역으로의 불필요한 메시지 전달과 문맥 전환 오버헤드를 제거한다. DINT[14]는 분산 트랜젝션 환경에서 빈번히 발생하는 트랜젝션 작업을 커널로 오프로드하여 직접 처리한다. 이처럼 네트워크 스택에서 eBPF를 활용한 기존 연구는 eBPF XDP와 TC 훅을 사용하여 네트워크 최적화에만 초점을 맞추고 있다.

3. 스토리지를 위한 eBPF

네트워크 시스템에서 eBPF를 활용한 연구가 많아짐에 따라 스토리지 시스템에서도 eBPF를 활용한 연구가 등장하고 있다. XRP[21, 22]는 NVMe 요청 처리 과정을 가속하기 위해 eBPF를 활용하여 NVMe 요청 제출 과정을 체이닝(chaning)하는 즉, 연속적인 NVMe 요청 재제출을 가능하게 한다. Kourtis[23] 등은 분산 NVM 스토리지 환경에서 효율적인 데이터 처리를 위해 eBPF를 활용하여 연산 스토리지(computational storage)로 데이터 처리 과정을 오프로드한다. Delilah[24] 역시 eBPF를 활용하여 데이터 처리 연산을 호스트 시스템에서 연산 스토리지로 오프로드한다. λ-IO[25]의 경우 연산 스토리지의 성능 최적화를 위해 새로운 통합 입출력 스택을 제안한다. 이처럼 현재 스토리지 시스템에서 eBPF를 활용하는 연구는 대부분 연산 스토리지로의 데이터 연산 처리 오프로드와 NVMe 드라이버 내부에서의 효율적인 연산 처리에 집중되어 있다.

현재 eBPF는 커널 네트워크 스택과 스토리지 스택 최적화를 위한 도구로 제안되고 있다. 다만, 현재 제안된 연구들은 네트워크 스택과 스토리지 스택 각 각을 따로 최적화 것으로, 이 둘을 통합하여 최적화한 연구는 존재하지 않는다. 이러한 점에서 ERB는 고성능 스토리지 시스템을 위해 커널 네트워크 스택과 스토리지 스택 모두 최적화하여 기존 연구들과 차별성을 가진다.

VIII. 논의 및 향후 연구

1. 효율적인 데이터 방출 방식(eviction mechanism)

데이터 방출 정책(eviction policy)은 메모리 제한 상황에서 저장 공간을 효율적으로 활용하기 위해 자주 접근되는 데이터를 우선하여 유지하는 방식이다. 이러한 방출 정책이 없이 요청받는 모든 데이터를 유지한다면 데이터 부하가 발생할 때 시스템 메모리 부족 문제가 발생할 가능성이 커진다. 현재 본 논문은 데이터 방출 정책을 지원하지 않기 때문에 ERB의 키-밸류 캐시 크기를 실험 벤치마크의 요청 수 전체를 저장할 수 있는 크기로 설정하였다. 따라서 이 경우 실험 벤치마크의 요청 수가 과도하게 증가하는 경우 메모리 부족 문제를 발생시킬 가능성이 크다. 따라서 본 논문은 이러한 메모리 부족 문제를 해결하고 작은 캐시 공간을 사용해 효율적인 데이터 유지를 가능하게 할 데이터 방출 정책을 향후 개발할 계획이다. 향후 개발 내용은 다음과 같다.

1. 자주 접근되는 데이터 구분
2. 데이터 방출 메커니즘 개발

eBPF는 새로운 패킷을 생성할 수 없으며 XDP 또는 TC를 통해 송수신된 패킷의 헤더와 페이로드(payload) 조작만이 가능하다. 따라서 방출시킬 키-밸류 쌍이 구별되더라도 데이터 방출을 위한 새로운 패킷을 생성하여 Redis 서버에게 전송할 수 없다. 또한 방출될 키-밸류 쌍이 생성될 때마다 이를 Redis 서버로 전송한다면 추가적인 네트워크 스택 오버헤드의 문제가 발생할 가능성이 크다. 따라서 ERB는 방출될 키-밸류 쌍을 비 활용 공간에 저장시킨 후 주기적으로 해당 공간의 데이터를 Redis 서버에게 전송하

고자 한다. 이때 XDP에 수신된 패킷의 페이로드를 변경하여 방출시킬 데이터를 함께 담아 보내는 방식을 사용한다면 최소한의 네트워크 오버헤드로 효과적인 데이터 방출이 가능할 것으로 예상된다.

2. 전송 제어 프로토콜(TCP protocol)의 확장

현재 ERB는 middlecache[45]를 기반으로 하여 eBPF 전송 제어 프로토콜(TCP protocol)을 지원한다. 하지만 해당 방식은 조각화 된 패킷에 대한 재정렬 기능을 제공하지 않아 MTU 크기 이내의 패킷에만 동작한다는 한계가 있다. 따라서 ERB는 더욱 확장적인 전송 제어 프로토콜 지원을 위해 다음과 같은 향후 연구를 진행할 계획이다.

1. eBPF 패킷 재정렬 메커니즘 개발

대부분의 광범위한 네트워크 환경에서는 다중 경로 라우팅과 패킷 혼잡 상황이 빈번히 발생한다. 따라서 전송된 패킷들은 도착 시 그 순서를 보장받지 못하기 때문에 이로 인한 패킷 재정렬은 필수적이다. 패킷 재정렬이 이루어지지 않는다면 도착 데이터의 순서가 어긋나 송신자가 잘못된 형태의 데이터를 읽게 되고 이는 특히 파일 전송, 스트리밍 서비스 등에서 심각한 오류를 발생시킨다. ERB는 전송 제어 프로토콜의 광범위한 활용을 지원하기 위해 조각화 된 패킷의 재정렬을 지원해야 한다. 이를 위해 ERB는 패킷 헤더의 패킷 순서 번호(Sequence number)를 확인하여 연속적인 패킷 순서 번호가 아닌 경우 조각화 된 패킷의 재정렬이 필요함을 인식하고 각 패킷의 순서 번호를 기반으로 패킷 재정렬 후 데이터를 처리하는 메커니즘을 개발할 계획이다.

IX. 결론

고성능 스토리지 시스템은 커널 네트워크 스택을 통한 패킷 처리와 스토리지 스택을 통한 데이터 영속성 보장이 필수적이다. Redis는 네트워크를 통한 패킷 처리와 데이터 영속성을 강하게 보장한다는 점에서 고성능 스토리지 시스템으로 활용될 수 있다.

본 논문은 eBPF를 활용하여 고성능 스토리지 시스템으로 활용될 수 있는 Redis의 성능 최적화를 진행한다. 이를 위해 스토리지 시스템에서 가장 큰 병목 지점으로 작용하는 커널 네트워크 스택 오버헤드와 스토리지 스택 오버헤드를 효율적으로 제거하는 네트워크와 스토리지 통합 입출력 프레임워크인 ERB를 제안한다. ERB는 eBPF XDP와 TC 훅을 사용하여 NIC으로 송수신되는 패킷을 직접 제어하고 페이지 캐시 작성과 디스크 입출력 경로를 네트워크 패킷 처리 경로와 분리해 클라이언트의 체감 응답 시간을 효과적으로 감소시킨다. ERB 실험 결과 ThreadedIO를 사용해 성능 최적화를 달성한 Redis보다 최대 6.5배의 성능 향상을 보였으며 99번째 백분위수 응답 시간 역시 평균 2.5배에서 3배가량 감소시켰다.

참고문헌

[1] Redis, Redis persistence, https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/, accessed Oct 21, 2024.

[2] Ultra-Low Latency with Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf, accessed Oct 21, 2024.

[3] Redis, Data Structure, https://redis.io/redis-enterprise/data-structures/, accessed Oct 21, 2024.

[4] Redis, Single Threads nature of Redis, https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/latency/#single-threaded-nature-of-redis, accessed Oct 22, 2024.

[5] Redis, Diving Into Redis 6.0, https://redis.io/blog/diving-into-redis-6/, accessed Oct 22, 2024.

[6] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux eBPF. https://ebpf.io, , accessed Oct 22, 2024.

[7] S. McCanne and V. Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture”. In Proceedings of the Usenix Winter 1993 Technical Conference, pages 259 - 270. 1993

[8] The Linux Kernel documentation, eBPF Verifier. https://docs.kernel.org/bpf/verifier.html, accessed Oct 22, 2024.

[9] The Linux Kernel documentation, Linux socket filtering aka berkeley packet filter(bpf), https://www.kernel.org/doc/html/latest/networking/filter.html#ebpf-verifier, accessed Oct 22, 2024.

[10] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. “The eXpress data path: fast programmable packet processing in the operating system kernel. In Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, pages 54 - 66. 2018.

[11] Vieira, Marcos AM, Matheus S. Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira, “Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications”. ACM Computing Surveys (CSUR) 53, pages 1-36, 2020

[12] Linux Programmer’s Manual. tc-bpf(8). https://man7.org/linux/man-pages/man8/tc-bpf.8.html,.accessed Oct 22, 2024.

[13] eBPF.io, what is eBPF, https://ebpf.io/what-is-ebpf/#hook-overview, accessed Oct 22, 2024.

[14] Zhou, Yang, Xingyu Xiang, Matthew Kiley, Sowmya Dharaniprag ada, and Minlan Yu. “{DINT}: Fast {In-Kernel} Distributed Transactions with {eBPF}”, In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 401-417. 2024.

[15] Linux Programmer's Manual. bpf-helpers(7), https://man7.org/linux/man-pages/man7/bpf-helpers.7.html, accessed Oct 22, 2024.

[16] The Linux kernel development community, BPF Kernel Functions (kfuncs), https://docs.kernel.org/bpf/kfuncs.html, accessed Oct 22, 2024.

[17] The Linux kernel development community. struct sk_buff. https://docs.kernel.org/networking/skbuff.html, accessed Oct 22, 2024.

[18] DPDK, Home - DPDK, https://www.dpdk.org/, accessed Oct 22, 2024.

[19] cilium.io, bpf architecture, https://docs.cilium.io/en/latest/bpf/architecture/, accessed Oct 22, 2024.

[20] eBPF.io, tail calls, https://docs.ebpf.io/linux/concepts/tail-calls/, accessed Oct 22, 2024.

[21] Zhong, Yuhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris et al, "{XRP}:{In-Kernel} Storage Functions with {EBPF}", In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 375-393. 2022.

[22] Zhong, Yuhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang, "BPF for storage: an exokernel-inspired approach", In Proceedings of the Workshop on Hot Topics in Operating Systems, pages 128-135. 2021.

[23] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. “Safe a nd efficient remote application code execution on disaggregated N VM storage with eBPF”. arXiv preprint arXiv:2002.11528, 2020.

[24] Hedam, Niclas, Morten Tychsen Clausen, Philippe Bonnet, Sangji n Lee, and Ken Friis Larsen, “Delilah: eBPF-offload on Computat ional Storage”, In Proceedings of the 19th International Workshop on Data Management on New Hardware, pages 70-76. 2023.

[25] Yang, Zhe, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Si yu He, and Jiwu Shu. “{λ-IO}: A Unified {IO} Stack for Comput ational Storage.″ In 21st USENIX Conference on File and Storage Technologies (FAST 23), pages 347-362. 2023.

[26] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Ko gias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. W hen idling is ideal: Optimizing tail-latency for heavy-tailed datace nter workloads with perséphone. In Proceedings of the ACM SIG OPS 28th Symposium on Operating Systems Principles, SOSP ’21, page 621 - 637, 2021.

[27] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In 14t h USENIX Symposium on Operating Systems Design and Imple mentation (OSDI 20), pages 281 - 297, 2020.

[28] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 1 – 16, 2014

[29] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A development kit to build high performance storage applications. In 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pages 154 – 161, 2017.

[30] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, page 325 – 341, 2017

[31] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 345 – 360, 2019.

[32] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In Proceedings of the Workshop on Hot Topics in Operating Systems, pages 73 - 80, 2019.

[33] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In Proceedings of ACM SOSP, pages 195 - 211, 2021.

[34] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In Proceedings of EuroSys, pages 1 - 16, 2019

[35] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In Proceedings of USENIX NSDI, pages 489 - 502, 2014.

[36] GitHub - facebookincubator/katran: A high performance layer 4 load balancer, 2020. URL: https://github.com/facebookincubator/katran.

[37] The IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc, accessed Oct 22, 2024.

[38] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In Proceedings of USENIX NSDI, pages 487 - 501, 2021.

[39] Zhou, Yang, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. "Electrode: Accelerating Distributed Protocols with {eBPF}." In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages. 1391-1407. 2023.

[40] Leslie Lamport. Paxos Made Simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), pages 51 - 58, 2001.

[41] Redis, Scaling, https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/, accessed Oct 21, 2024.

[42] Redis, Replication, https://redis.io/docs/latest/operate/oss_and_stack/management/replication/, accessed Oct 21, 2024.

[43] Thadani, Moti N., and Yousef A. Khalidi. An efficient zero-copy I/O framework for UNIX. Sun Microsystems Laboratories, 1995.

[44] Rothberg, Valentin. Interrupt handling in Linux., 2015.

[45] Pang, Yiren, Sheng Chen, Wenxin Li, Hao Liu, Yulong Li, Xin He, Song Zhang, Zewei Guan, Lide Suo, and Yuan Liu. MiddleCache: Accelerating TCP based In-memory Key-value Stores using eBPF. In 2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS), pages 2428-2435. IEEE, 2023.

[46] Intel® Optane™ SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html. accessed Oct 21, 2024.

[47] Home, DPDK, https://www.dpdk.org/, accessed Oct 21, 2024.

[48] eBPF.io, Loader&Verification Architecture, https://ebpf.io/what-is-ebpf/#loader--verification-architecture, accessed Oct 21, 2024.

[49] Khushi_chhillar, Unlocking Network Performance with XDP and eBPF, https://medium.com/@kcl17/unlocking-network-performance-with-xdp-and-ebpf-67c712128025, accessed Oct 21, 2024.

[50] eBPF.io, Maps, https://docs.ebpf.io/linux/concepts/maps/, accessed Oct 21, 2024.

[51] eBPF.io, Helper functions, https://docs.ebpf.io/linux/helper-function/, accessed Oct 21, 2024.

[52] eBPF.io, dynptrs, https://docs.ebpf.io/linux/concepts/dynptrs/, accessed Oct 21, 2024.

[53] eBPF.io, Program type BPF_PROG_TYPE_KPROBE, https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_KPROBE/, accessed Oct 21, 2024.

ABSTRACT

ERB: Unified Network and Storage I/O Optimization Using eBPF for Redis

An Seolryeong
Department of Computer Science
Graduate School of
Sungshin University

With advancements in storage technology, the performance gap between memory and storage has progressively narrowed. However, the emergence of high-performance storage devices has introduced software overhead in the storage stack as a new bottleneck. In distributed systems, client-server interactions often necessitate multiple round trips through the kernel's network and storage stacks, leading to significant context-switching and data copying overheads. These inefficiencies introduce unnecessary processing costs within the Linux kernel's network and storage stacks, becoming a critical factor that degrades the performance of high-performance storage systems.

Redis[1], a prominent example of a high-performance storage system, is designed as an in-memory data store. By leveraging its memory-centric structure and features such as the Append-Only File (AOF) for persist

ence, Redis ensures data reliability and stability, making it suitable for hi gh-performance use cases. Nevertheless, Redis also faces performance li mitations in environments requiring high throughput, primarily due to ker nel software stack overheads. To address these challenges, recent efforts have introduced kernel bypass techniques that circumvent the Linux kern el's network and storage stacks to eliminate unnecessary overheads fund amentally[29, 47]. While effective, these techniques come with significant drawbacks, including inefficient CPU resource usage and the requirement for users to reimplement functionalities provided by the kernel.

This paper proposes ERB (eBPF-based Redis Booster), a novel networ k and storage integrated I/O framework leveraging eBPF (extended Berk eley Packet Filter)[6] to overcome these limitations and maximize the per formance of high-performance storage systems. ERB is specifically desig ned to optimize Redis[1] and operates based on the TCP protocol. It achi eves high-speed packet processing by utilizing eBPF XDP and TC hook points. To support Redis's data persistence, ERB employs a kernel modul e-based I/O mechanism that separates disk I/O from network packet pro cessing, thereby reducing client-perceived response times through asynch ronous kernel page cache writes and fdatasync() calls.

Experimental results demonstrate that ERB achieves up to a 6.5x perfo rmance improvement compared to a multi-threaded optimized Redis imple mentation, along with a 2.5x to 3x reduction in the 99th-percentile respo nse times. These findings underscore the necessity of integrated optimiza tion across the network and storage stacks to effectively enhance the pe rformance of high-performance storage systems.