김 규 영 교수 지도

석사학위 청구논문

eBPF/XDP 기반 인-메모리 키-밸류 스토어 성능 분석

2025

성신여자대학교 대학원

컴퓨터학과

이 지 현

eBPF/XDP 기반 인-메모리 키-밸류 스토어 성능 분석

김 규 영 교수 지도

이 논문을 석사학위논문으로 제출함

2024년 11월

성신여자대학교 대학원
컴퓨터학과
이 지 현

인 준 서

이지현의 석사학위 논문으로 인준함

2025년 1월

심사위원장 박 지 웅 (서명 또는 인)

심 사 위 원 김 규 영 (서명 또는 인)

심 사 위 원 임 연 섭 (서명 또는 인)

성신여자대학교 대학원

논문개요

인-메모리 키-밸류 스토어는 오늘날 온라인 서비스의 데이터를 저장하는 핵심 구성 요소이다. 그러나 커널 네트워크 스택을 경유하면서 발생하는 오버헤드는 데이터 액세스 지연을 지나치게 증가시키는 요인이 된다.

본 논문에서는 프로그래머블 호스트 패킷 처리 기술인 XDP(eXpress Data Path)와 eBPF를 활용한 인-메모리 키-밸류 스토어를 설계하고 성능을 분석한다. XDP는 커널의 네트워크 인터페이스 카드 드라이버 수준에서 패킷을 처리하는 것을 가능케 하여 커널 네트워크 스택의 오버헤드를 회피할 수 있는 기회를 제공한다. 우리는 eBPF/XDP 기반 인-메모리 키-밸류 스토어의 성능을 널리 사용되는 인-메모리 키-밸류 스토어인 Redis와 비교하는 실험을 진행하였다. 실험 결과, eBPF/XDP 기반 인-메모리 키-밸류 스토어는 Redis에 비해 최대 24배 높은 처리량을 보였으며, 더 낮은 지연 시간을 유지하는 것을 확인할 수 있었다.

목 차

논문 개요

Ⅰ. 서 론 ····· 1

Ⅱ. 배 경 ····· 3

1. 키-밸류 스토어 ····· 3

2. 네트워크 스택 오버헤드 ····· 5

3. eBPF ····· 7

1) BPF 검증기 ····· 8

2) BPF 맵 ····· 8

3) eBPF 훅 ····· 8

4. XDP ····· 10

Ⅲ. eBPF/XDP 기반 인-메모리 키-밸류 스토어 ····· 12

1. 해시 테이블 맵 ····· 12

2. 요청 처리 방법 ····· 14

1) 읽기 요청 ····· 14

2) 쓰기 요청 ····· 15

3) 요청 반환 ····· 15

Ⅳ. 성능 분석 ····· 16
1. 실험 환경 ····· 16
1) 테스트베드 ····· 16
2) 비교군 ····· 17
3) 워크로드 ····· 17
2. 실험 결과 ····· 18
1) 처리량 ····· 18
2) 지연 시간 ····· 20
3) 쓰기 비율의 영향 ····· 22
4) 데이터 크기의 영향 ····· 24
5) 스캔 비율의 영향 ····· 26

Ⅴ. 논의 및 향후 연구 ····· 29
1. eBPF 제약사항 ····· 29
1) 정적 메모리 할당 ····· 29
2) 제한된 프로그래밍 ····· 30
2. 데이터 백업 ····· 32

Ⅵ. 관련 연구 ····· 33
Ⅶ. 결론 ····· 36

참고문헌
ABSTRACT

그림 목차

[그림 1] Redis 중앙 지연 시간 네트워크 스택 오버헤드 ····· 5
[그림 2] Redis 꼬리 지연 시간 네트워크 스택 오버헤드 ····· 5
[그림 3] eBPF 작업 흐름 ····· 9
[그림 4] XDP 기반 네트워크 패킷 처리 구조 ····· 10
[그림 5] eBPF/XDP 기반 인-메모리 키-밸류 스토어 요청 처리 방법 ····· 13
[그림 6] 키 접근 패턴별 최대 처리량 ····· 18
[그림 7] 처리량 대비 중앙 지연 시간 ····· 20
[그림 8] 처리량 대비 꼬리 지연 시간 ····· 20
[그림 9] 쓰기 요청 비율별 처리량 ····· 22
[그림 10] 키 데이터 크기별 처리량 ····· 24
[그림 11] 밸류 데이터 크기별 처리량 ····· 24
[그림 12] 스캔 요청 비율별 처리량 ····· 26
[그림 13] 처리량 대비 꼬리 지연 시간 (99%-읽기, 1%-스캔) ····· 28
[그림 14] 처리량 대비 꼬리 지연 시간 (90%-읽기, 10%-스캔) ····· 28

표 목차

<표 1> 서버 사양 ·· 17

I. 서 론

검색, 소셜 네트워킹 서비스, 전자 상거래 등과 같은 오늘날의 온라인 서비스들은 저지연 데이터 접근을 위해 키-밸류 스토어를 널리 활용하고 있다. 키-밸류 스토어는 관계형 데이터베이스와 달리, 키-밸류 쌍을 저장하는 단순한 구조로 테이블 조인을 필요로 하지 않아 데이터 검색과 저장이 매우 효율적이다. 이러한 성능을 극대화하기 위해 하드디스크 기반 저장소 대신 메모리에 데이터를 저장하는 인메모리 키-밸류 스토어가 등장하였다. 대표적인 인-메모리 키-밸류 스토어로는 Redis [1]와 Memcached [2]가 있으며, 메모리에 데이터를 저장함으로써 데이터 접근 속도가 비약적으로 향상된다. 이러한 특성 덕분에 실시간 데이터 접근이 요구되는 애플리케이션에 특히 적합하다. 실제로 X(구 Twitter)와 Facebook에서는 Redis와 Memcached를 통해 서비스 성능을 최적화하고 있다 [3, 4, 5].

한편, 데이터 요청 패킷이 비대해진 오늘날의 커널 네트워크 스택을 통과하는 과정에서 발생하는 오버헤드는 키-밸류 스토어가 저지연을 달성하는 데 있어서의 가장 큰 걸림돌 중 하나이다. 표준 소켓 I/O를 사용하는 시스템은 커널 및 사용자 수준에서 높은 네트워크 스택 오버헤드를 초래하기 때문에, 이를 해결하기 위해 사용자 계층에서 네트워크 스택 기능을 구현하여 커널 네트워크 스택을 경유하지 않는 키-밸류 스토어 기술이 제안되기도 하였다 [6]. 그러나 이러한 커널 우회 기술은 복잡한 버전 관리와 디버깅 이슈 등으로 인한 배포 상의 어려움으로 인해 널리 사용되고 있지 못하며 여전히 커널 네트워크 스택 기반의 키-밸류 스토어가 광범위하게 사용되고 있다.

이에 본 논문에서는 eBPF/XDP 기반 인-메모리 키-밸류 스토어를 제안한다. XDP(eXpress Data Path)는 Linux 커널의 eBPF (extended Berkeley Packet Filter) 구성 요소 중 하나로, 네트워크 인터페이스 카드(Network

Interface Card, NIC) 드라이버 영역에서 커스텀 로직을 통해 패킷을 처리할 수 있는 고성능 프로그래머블 호스트 패킷 처리 프레임워크이다 [7]. eBPF와 XDP를 활용하여 패킷이 커널 네트워크 스택을 방문하지 않고도 클라이언트가 요청한 읽기 또는 쓰기 작업을 처리한 후 즉시 응답을 반환케하는 eBPF/XDP 기반 인-메모리 키-밸류 스토어를 설계 및 구현함으로써 커널 네트워크 스택에서 발생하는 오버헤드를 완전히 회피하고 높은 처리량과 낮은 지연 시간을 달성한다.

eBPF/XDP 기반 인-메모리 키-밸류 스토어의 성능을 평가하기 위해 널리 사용되는 인-메모리 키-밸류 스토어인 Redis와 비교 실험을 수행하였다. 실험 결과, eBPF/XDP 기반 인-메모리 키-밸류 스토어는 Redis 대비 최대 24배 높은 처리량을 달성하며, 낮은 꼬리 지연 시간을 유지하는 것을 확인하였다. 이러한 결과는 XDP 기반의 인-메모리 키-밸류 스토어가 고성능과 배포 용이성을 모두 달성할 수 있는 차세대 키-밸류 스토어로서의 잠재력을 가지고 있음을 보여준다.

본 논문은 7장으로 구성되며, 각 장은 다음과 같이 구성된다. 2장에서는 키-밸류 스토어에 대해 서술한 뒤 Redis의 지연 시간에서 네트워크 스택 오버헤드가 차지하는 비율을 분석한다. 또한, 네트워크 스택을 우회하기 위해 본 논문에서 사용한 eBPF와 XDP 기술에 관해 서술한다. 3장에서는 eBPF/XDP 기반 인-메모리 키-밸류 스토어의 설계와 구현 방식을 설명한다. 4장에서는 Redis와 비교하여 eBPF/XDP 기반 인-메모리 키-밸류 스토어의 성능을 비교하여 분석한다. 5장에서 추가적인 논의와 향후 연구 방향을 다루고, 6장에서는 eBPF와 XDP와 관련된 선행 연구 및 인-메모리 키-밸류 스토어의 성능 향상과 관련된 기존 연구를 소개한다. 마지막으로 7장에서 논문을 마무리한다.

Ⅱ. 배 경

본 장에서는 기존 키-밸류 스토어의 종류 및 특징을 서술한다. 또한, 널리 사용되는 인-메모리 키-밸류 스토어인 Redis를 사용할 때 발생하는 커널 네트워크 스택의 오버헤드를 측정하여, 본 논문이 해결하고자 하는 문제점을 논의한다. 또한 이를 해결하기 위해 본 논문에서 활용한 기술인 eBPF와 XDP에 대해 설명한다.

1. 키-밸류 스토어

키-밸류 스토어는 간단한 키-밸류 쌍 형태로 데이터를 저장하는 비관계형 데이터베이스로, NoSQL 데이터베이스에 속한다. 관계형 데이터베이스는 데이터를 테이블 형식으로 저장하고 작업을 위해 테이블 간의 조인을 필요로 하지만, 키-밸류 스토어는 테이블 조인 없이 데이터를 저장하고 조회할 수 있어 읽기 및 쓰기 속도가 빠른 것이 특징이다. 이러한 단순한 구조 덕분에 데이터 분산이 용이하여 여러 서버에 쉽게 분배할 수 있다.

키-밸류 스토어는 저장 방식에 따라 디스크 기반 키-밸류 스토어와 인-메모리 키-밸류 스토어로 구분된다. 디스크 기반 키-밸류 스토어는 데이터를 메모리가 아닌 하드디스크에 저장하여 대용량 데이터를 효율적으로 처리할 수 있다. 전원이 꺼져도 데이터가 유지되는 비휘발성 특성을 가지지만, 메모리에서 데이터를 처리하는 방식에 비해 속도가 느리다는 단점이 있다. 대표적인 디스크 기반 키-밸류 스토어에는 Facebook의 RocksDB [12], Google의 LevelDB [13]가 있다.

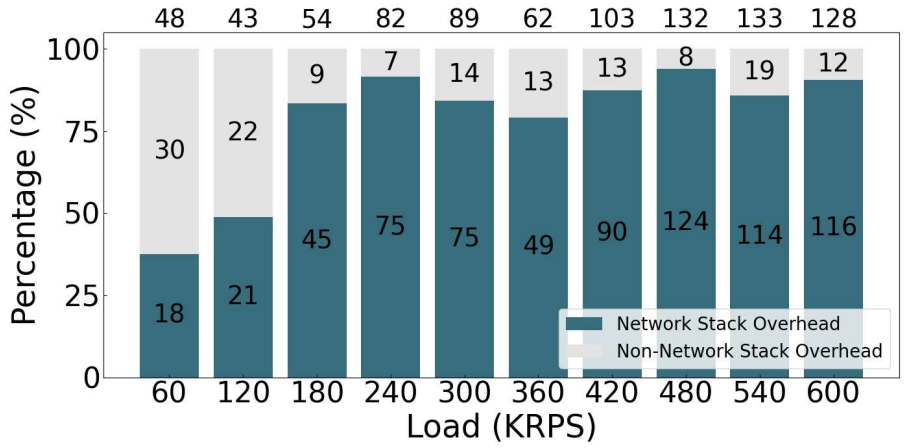
반면, 인-메모리 키-밸류 스토어는 데이터를 메모리에 저장하여 접근 속

도를 극대화한다. 메모리에서 직접 데이터를 읽거나 쓰기 때문에 마이크로초 규모의 매우 빠른 응답 시간이 요구되는 애플리케이션에 유리하다. 그러나 데이터를 휘발성 메모리에 저장하기 때문에 전원이 꺼지면 데이터가 손실될 수 있다. 이러한 이유로 인-메모리 키-밸류 스토어는 주로 캐시 용도로 사용되며, 대표적인 예로 Redis [1]와 Memcached [2]가 있다.

인-메모리 키-밸류 스토어의 빠른 처리 속도가 디스크 기반 데이터베이스의 느린 속도를 보완해줄 수 있어 대규모의 웹 서비스에서 Redis와 Memcached를 캐시 시스템으로 주로 사용하고 있다. 예를 들어 X(구 Twitter)에서는 캐싱 솔루션으로 Redis와 Memcached를 모두 활용하였고 [4], Facebook에서는 Memcached를 통해 캐시 시스템을 구축하여 데이터베이스의 부하를 줄였다 [3].

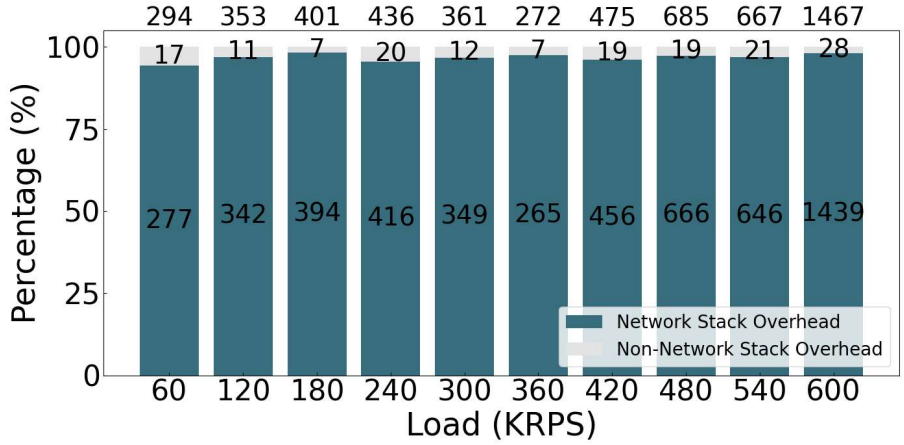
최근에는 키-밸류 스토어의 모든 데이터가 메모리로 이동하면서 인-메모리 키-밸류 스토어를 캐싱 솔루션뿐만 아니라 완전한 키-밸류 스토어로 활용하는 추세이다 [9].

2. 네트워크 스택 오버헤드



[그림 1] Redis 중앙 지연 시간 네트워크 스택 오버헤드

(단위: 마이크로초, μs)



[그림 2] Redis 꼬리 지연 시간 네트워크 스택 오버헤드

(단위: 마이크로초, μs)

인-메모리 키-밸류 스토어가 겪는 네트워크 스택의 오버헤드가 지연 시간에 미치는 영향을 분석하기 위해, 대표적인 인-메모리 키-밸류 스토어인

Redis를 이용하여 실험을 수행하였다. 본 실험에서는 VMA 라이브러리를 사용하여 클라이언트의 네트워크 패킷 송수신 시 커널 네트워크 스택을 우회하여 사용자 공간에서 직접 처리함으로써 클라이언트 측 네트워크 지연 시간을 최소화하였다. 이를 통해 클라이언트가 요청 패킷을 송신하고 응답 패킷을 수신할 때까지의 총 지연 시간에서 Redis의 GET 작업 처리에 소요된다.

그림 1은 다양한 부하(Load) 조건에서 전체 지연 시간의 50번째 백분위수(중앙값)에서 네트워크 스택 오버헤드가 차지하는 비율을 보여준다. 실험 결과, 부하가 최대 처리량의 30%인 180 KRPS 이상일 때 지연 시간의 80% 이상이 네트워크 스택 오버헤드에 해당하는 것으로 나타났다.

또한, 그림 2는 전체 지연 시간의 99번째 백분위수(꼬리 지연 시간)에서 네트워크 스택 오버헤드가 차지하는 비율을 보여준다. 이 결과에서는 모든 부하 조건에서 꼬리 지연 시간의 94% 이상을 네트워크 스택 오버헤드가 차지함을 확인하였다. 또한, 부하가 증가할수록 네트워크 스택 오버헤드가 차지하는 비율이 높아지는 것을 확인하였다. 이는 네트워크 스택 오버헤드가 꼬리 지연 시간에 상당한 영향을 미쳐 서비스의 응답 성능을 결정짓는 주요 요소로 작용함을 의미한다.

꼬리 지연 시간은 전체 서비스 지연 시간의 상한을 결정짓는 중요한 요소로 작용한다. 마이크로초 규모의 지연 시간을 요구하는 엄격한 서비스 수준 목표(SLO)를 만족시키기 위해서는 꼬리 지연 시간을 줄이는 것이 필수적이다. 위 실험의 결과는 네트워크 스택 오버헤드를 최소화하는 것이 인-메모리 키-밸류 스토어의 성능 향상에 있어 중요한 과제임을 보여준다.

3. eBPF

eBPF는 패킷 필터링을 위한 기술인 BPF(Berkeley Packet Filter)를 확장하여 발전한 기술로, Linux 커널에서 커널 소스를 수정하거나 재컴파일하지 않고도 사용자 정의 프로그램을 커널 수준에서 안전하게 실행할 수 있도록 설계된 차세대 커널 확장 기술이다 [8]. eBPF는 네트워크 트래픽 관리, 성능 모니터링, 보안 강화 등 다양한 용도로 사용되며, 특히 시스템 레벨에서 실시간 데이터를 효율적으로 수집하고 분석할 수 있는 강력한 도구로 자리 잡고 있다.

eBPF 프로그램은 주로 C언어로 작성되며, LLVM 컴파일러를 통해 eBPF 바이트 코드로 컴파일된 후 커널에 로드된다. 로드된 eBPF 프로그램은 커널 내에서 제한된 명령어 집합과 정적 메모리 할당만을 사용할 수 있는 환경에서 실행되지만, 커널의 다양한 데이터와 이벤트에 접근할 수 있어 실시간 모니터링과 성능 최적화에 매우 적합하다. 이러한 제한적 실행 환경 덕분에 eBPF 프로그램은 커널의 안정성을 유지하면서도 높은 유연성과 효율성을 제공한다.

eBPF의 유연성은 단순한 패킷 필터링을 넘어서 다양한 커널 기능을 확장할 수 있도록 하였다. 특히 eBPF를 통해 Linux 커널이 제공하는 데이터와 이벤트에 실시간으로 접근함으로써 시스템 성능을 더욱 정밀하게 조정할 수 있다. 예를 들어, eBPF는 네트워크 패킷 처리 성능을 개선하거나, 파일 시스템 및 메모리 관리와 같은 영역에서 세밀한 제어를 가능하게 한다. 이처럼 eBPF는 커널의 근본적인 보안성과 안정성을 유지하면서 운영 체제를 보다 정교하게 제어하고 확장할 수 있는 혁신적인 방법을 제공하고 있다.

1) BPF 검증기

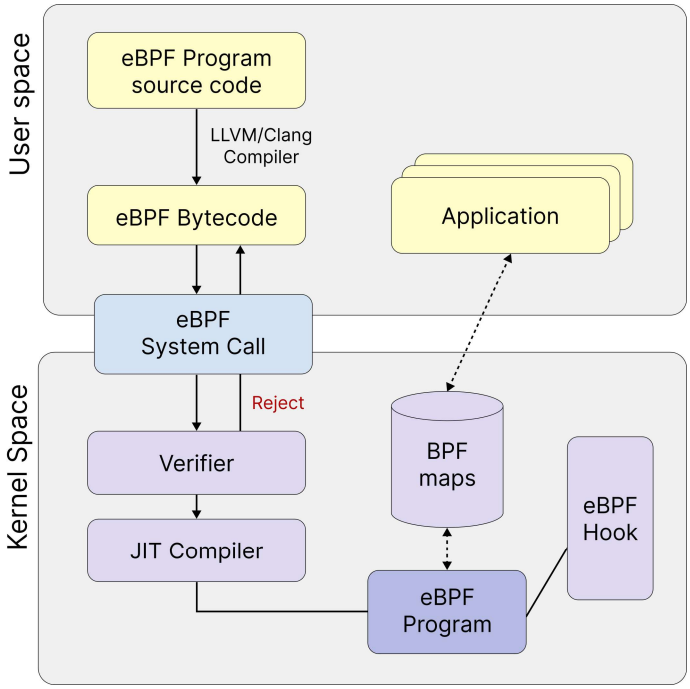
커널 내에서 실행되는 eBPF 프로그램의 안정성을 보장하기 위해 BPF 검증기(verifier)가 존재한다. BPF 검증기는 eBPF 바이트코드를 통해 프로그램의 모든 경로를 확인하여 프로그램이 최종적으로 항상 종료되는지를 확인한다. 또한, 모든 메모리 포인터가 유효한 범위의 메모리를 참조하는지를 검증한다. 이 두 가지 검증을 통과한 프로그램만이 커널에 적재되어 실행될 수 있다.

2) BPF 맵

BPF 맵(map)은 커널 안에 존재하는 자료 구조이다. 커널 내에서 실행되는 eBPF 프로그램뿐만 아니라 사용자 공간에서 실행되는 프로그램도 BPF 맵에 접근할 수 있다. 해시 테이블 맵, 배열 맵, 프로그램 배열 맵 등 다양한 용도의 BPF 맵을 지원하고 있다. 이러한 맵에 저장된 요소들은 보조 함수(helper function)를 통해 조회, 갱신, 삭제가 가능하다.

3) eBPF 훅

eBPF 프로그램은 커널의 다양한 실행 지점에서 트리거될 수 있으며, 이러한 지점을 훅(hook)이라고 한다. eBPF 훅은 사용자가 프로그램이 실행되기를 원하는 커널의 특정 위치를 지정하여, 특정 이벤트 발생 시 eBPF 프로그램이 실행되도록 한다. 네트워크 관련 훅으로는 전송 계층을 다루는 TC 훅과 네트워크 인터페이스 수준에서 작동하는 XDP 훅이 있으며, 각 훅은 해당 레이어에서 패킷 필터링, 변형, 통계 수집 등의 다양한 작업을 수행할 수 있다.

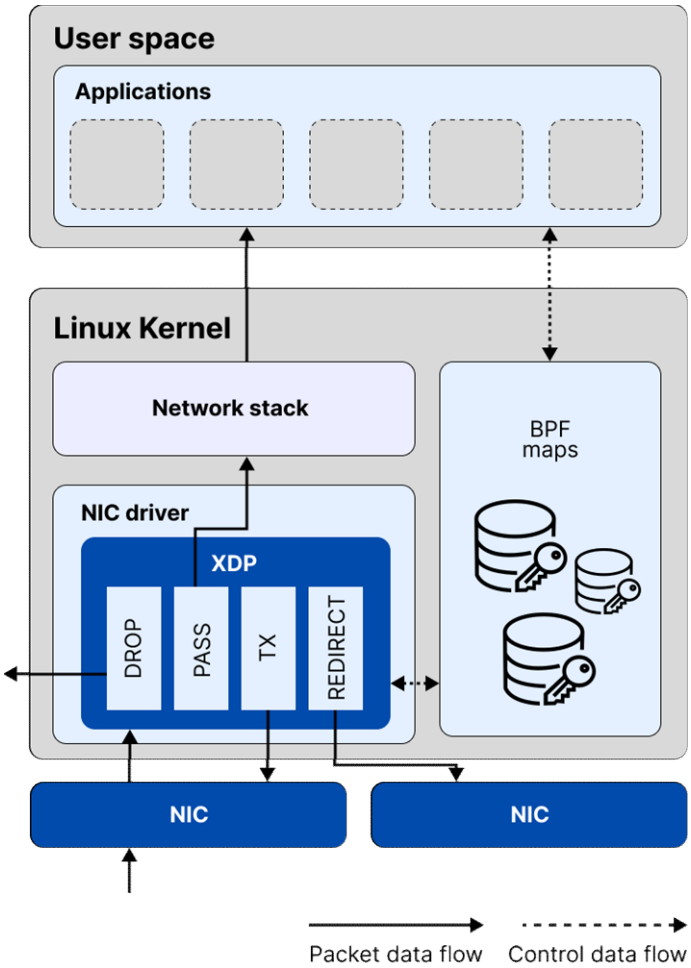


[그림 3] eBPF 작업 흐름

그림 3은 eBPF의 전체적인 작업 흐름을 보여준다. eBPF 프로그램 실행을 위해서는 먼저 C 또는 Rust와 같은 고급 언어로 소스 코드를 작성하고, 이를 LLVM을 통해 컴파일하여 eBPF 바이트 코드를 생성한다. 생성된 바이트 코드는 eBPF 시스템 호출을 통해 커널에 적재되며, 이 과정에서 eBPF 검증기가 메모리 안전성과 프로그램 종료 가능성을 검사한다. 검증을 통과하지 못할 경우 적재가 거부되며, 통과한 경우 JIT 컴파일러가 바이트 코드를 커널의 네이티브 코드로 변환한다. 이후 eBPF 프로그램은 커널의 특정 eBPF 훅(Hook)에 연결되어, 설정된 이벤트나 조건이 발생할 때 자동으로 실행될 수 있다.

이러한 구조를 통해 eBPF는 커널 레벨에서 안전하게 다양한 기능을 확장할 수 있으며, 사용자 공간과의 데이터 통신을 통해 높은 성능을 유지하면서도 풍부한 기능을 제공한다.

4. XDP



[그림 4] XDP 기반 네트워크 패킷 처리 구조

XDP(eXpress Data Path)는 리눅스 커널에서 고성능 패킷 처리를 위해 설계된 프레임워크이다 [7]. 이 프레임워크는 네트워크 인터페이스 카드(NIC) 드라이버 수준에서 패킷을 처리하여 커널 네트워크 스택을 우회함으로써 지연 시간을 최소화하고, 높은 패킷 처리 속도를 달성할 수 있도록 한다.

XDP는 eBPF 프로그램과 통합되어 동작한다. eBPF 프로그램은 XDP 훅에

로드되어, 패킷이 커널 네트워크 스택에 진입하기 전에 직접 처리할 수 있게 한다. 이를 통해 패킷 필터링, 로드 밸런싱, DDoS 방어 등의 다양한 네트워크 작업을 효율적으로 수행할 수 있으며, 커널에서의 사용자 정의 패킷 처리 로직을 실시간으로 구현할 수 있다.

그림 4는 XDP 기반 네트워크 패킷의 처리 구조를 보여준다. 클라이언트에서 보낸 요청이 NIC에서 수신되면, NIC 드라이버에서 즉시 XDP 프로그램이 실행되어 해당 패킷을 처리할 기회를 제공받는다. XDP 프로그램은 패킷을 특정 작업에 따라 다음과 같이 처리할 수 있다:

- XDP_DROP: 패킷을 즉시 폐기한다.
- XDP_PASS: 패킷을 커널 네트워크 스택으로 전달한다.
- XDP_TX: 패킷을 수신한 인터페이스로 다시 전송한다.
- XDP_REDIRECT: 패킷을 다른 인터페이스로 리디렉션한다.

이러한 XDP의 패킷 처리 구조는 패킷이 커널 네트워크 스택을 통과하기 전에 고속으로 처리할 수 있는 기회를 제공하므로, 낮은 지연 시간과 높은 처리량을 요구하는 경우에 적합하다.

Ⅲ. eBPF/XDP 기반 인-메모리 키-밸류 스토어

2장에서 살펴본 바와 같이, XDP 훅에서 eBPF 프로그램을 실행하면 네트워크 패킷을 NIC 드라이버 단계에서 직접 처리할 수 있으며, 이를 통해 커널 네트워크 스택의 복잡한 과정을 우회하여 성능을 향상시킬 수 있다. 이러한 특성을 활용하여 eBPF와 XDP를 기반으로 커널 내에서 인-메모리 키-밸류 스토어를 설계함으로써 커널 네트워크 스택의 오버헤드를 최소화할 수 있다.

본 장에서는 eBPF/XDP 기반 인-메모리 키-밸류 스토어의 설계 및 구현 방안을 상세히 설명한다.

1. 해시 테이블 맵

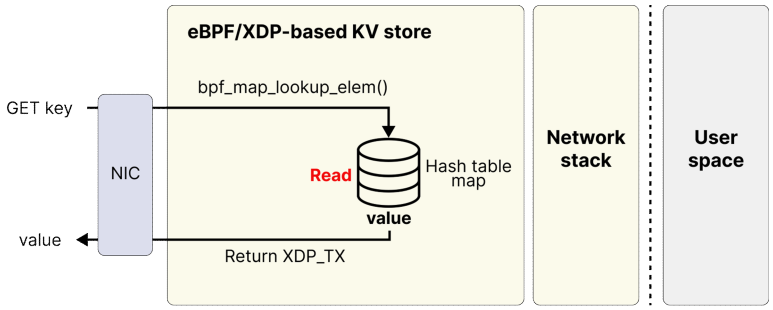
키-밸류 스토어를 구현하기 위해서는 데이터를 효율적으로 저장하고 빠르게 조회할 수 있는 자료구조가 필요하다. 상용 키-밸류 스토어인 Redis와 Memcached는 높은 성능을 위해 해시 테이블을 이용하여 데이터를 관리한다. 본 연구에서도 이러한 해시 테이블의 장점을 활용하여 BPF 맵(map)의 한 종류인 해시 테이블 맵을 채택하여 키-밸류 쌍을 저장하도록 하였다.

해시 테이블 맵은 다양한 형식과 크기의 키-밸류 쌍을 저장할 수 있으며, 리눅스 커널 내에서 제공되는 해시 함수를 통해 빠른 조회와 갱신을 지원한다. 리눅스 커널에서는 해싱 작업을 위해 Jenkins Hash(jhash)를 사용하며, 해시 충돌 발생 시에는 체이닝 방식을 통해 충돌을 해결한다. 또한, 해시와 관련한 모든 기능은 리눅스 커널에서 제공되므로 키-밸류 스토어를 설계할 때 사용자 정의 데이터에 맞는 해시 테이블 맵을 정의하여 사용하기만 하면

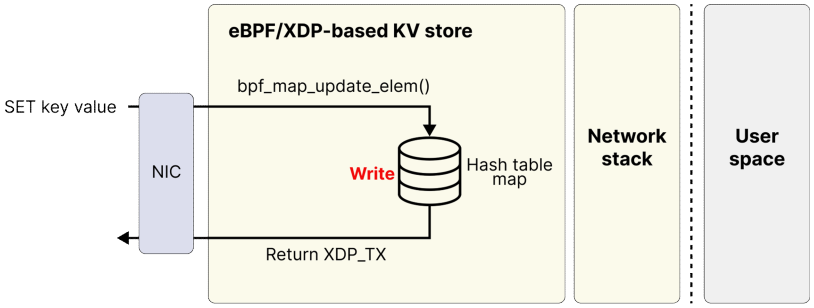
되어, 설계와 구현이 비교적 간단하다는 장점이 있다.

해시 테이블 맵을 활용한 키-밸류 스토어 자료구조는 XDP 프로그램 내에서 정의된다. XDP 프로그램이 커널에 적재되면 메모리 내에 해시 테이블 맵을 위한 공간이 할당된다. BPF 맵이 커널 메모리에 위치하기 때문에 이를 빠르고 효과적으로 관리할 수 있게 된다.

2. 요청 처리 방법



(a) Read request



(b) Write request

[그림 5] eBPF/XDP 기반 인-메모리 키-밸류 스토어 요청 처리 방법

1) 읽기 요청

그림 5의 (a)는 읽기 요청 처리 과정을 나타낸다. 읽기(GET) 요청이 NIC을 통해 들어오면, 즉시 NIC 드라이버에서 XDP 프로그램을 실행한다. XDP 프로그램은 패킷에서 키를 추출하고 해시 테이블 맵에서 해당 키에 대응하는 밸류를 조회한다. 이를 처리하기 위해 XDP 프로그램에서는 *bpf_map_lookup_elem()* 이라는 헬퍼 함수를 사용한다. 해시 테이블 맵의 포인터와 키의 포인터를 인자로 넘겨주면, 해시 테이블 맵에서 해당 키에 매

핑된 밸류를 반환한다. 이렇게 구한 밸류를 패킷에 넣어 클라이언트에 전달한다.

2) 쓰기 요청

그림 5의 (b)는 읽기 요청 처리 과정을 나타낸다. NIC 드라이버가 패킷을 수신했을 때 XDP 프로그램을 실행하는 것은 읽기 요청 처리와 동일하다. XDP 프로그램은 패킷에서 키와 밸류를 추출하고 이를 해시 테이블 맵에 저장한다. 이를 위해 *bpf_map_update_elem()*을 호출한다. 해시 테이블 맵의 포인터와 키, 밸류의 포인터를 인자로 넘겨주면 해시 테이블에 데이터를 저장하고, 만약 동일한 키가 이미 존재할 경우, 기존 밸류를 덮어쓴다. 쓰기 요청의 경우, 패킷에 추가할 데이터가 없기 때문에 수신한 패킷 그대로 클라이언트에 전달한다.

3) 요청 반환

XDP 프로그램에서 읽기 또는 쓰기 요청을 처리하고 나면 패킷을 커널 네트워크 스택이나 유저 공간으로 보낼 필요가 없기 때문에 클라이언트로 바로 반환할 수 있다. 패킷의 source와 destination MAC, IP, L4 포트 번호를 교환한 후 XDP_TX를 반환(return)하면, 패킷은 네트워크 인터페이스 카드로 즉시 전송된다. 이렇게 하면 패킷이 NIC 드라이버 수준에서 처리된 후 바로 클라이언트로 반환되어 네트워크 스택의 오버헤드를 회피하게 된다.

Ⅳ. 성능 분석

본 장에서는 eBPF/XDP 기반 인-메모리 키-밸류 스토어의 성능을 평가한다. 이를 위해 널리 사용되는 인-메모리 키-밸류 스토어인 Redis와의 성능 비교를 수행하였다. 최대 처리량과 지연 시간을 측정하여 성능을 비교하고, 데이터 크기, 쓰기 요청 비율, 스캔 요청 비율에 따른 성능 변화를 분석함으로써, eBPF/XDP 기반 인-메모리 키-밸류 스토어가 다양한 워크로드에서 안정적으로 동작하며 기존 인-메모리 키-밸류 스토어 대비 우수한 성능을 달성하는지 평가한다.

eBPF/XDP 기반의 인-메모리 키-밸류 스토어를 구현하기 위해 XDP 프로그램과 이를 로드하는 역할의 유저 프로그램을 작성하였다. XDP 프로그램과 유저 프로그램 모두 C언어로 작성되었으며, XDP 프로그램에서 데이터를 저장할 해시 테이블 맵을 정의하고, 유저 프로그램에서 해당 해시 테이블 맵에 접근하여 실험을 위한 임의의 초기 데이터를 삽입하도록 하였다.

1. 실험 환경

1) 테스트베드

모든 실험은 100Gbps NIC을 장착한 8대의 서버 노드와 이를 연결하는 스위치로 구성된 클러스터 환경에서 진행되었다. 이 중 7대의 노드는 요청을 생성하는 클라이언트로 사용되었으며, 나머지 1대의 노드는 Redis 또는 XDP 프로그램을 실행하는 서버로 사용되었다. XDP 프로그램은 서버에서 XDP 네이티브(Native) 모드로 실행되었다. 자세한 서버 사양은 <표 1> 에서 확인할 수 있다.

<표 1> 서버 사양

OS	Ubuntu 20.04 LTS
Kernel	Linux Kernel 6.8.0
CPU	Intel i5-12600K @ 3.7 Ghz
RAM	32 GB of DDR5 memory
NIC	Nvidia ConnectX-5 100G NIC
Switch	APS Networks BF6064X-T switch

2) 비교군

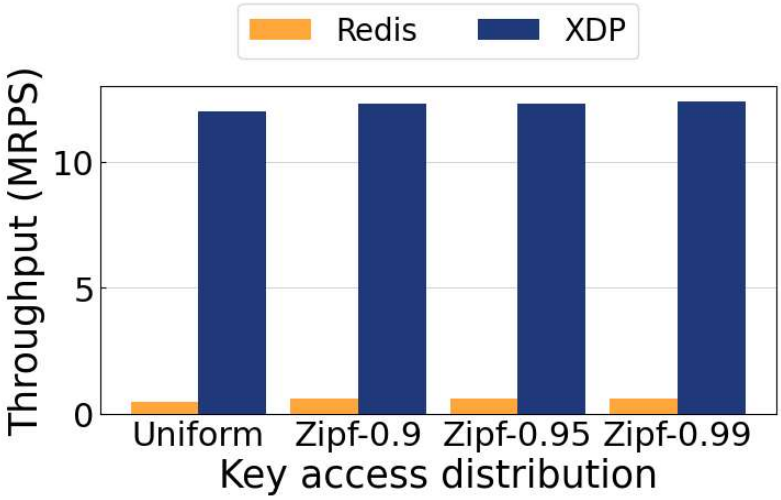
단일 스레드 기반으로 동작하는 Redis와 멀티 코어에서 동작하는 XDP의 성능을 비교하기 위해, Redis 인스턴스를 CPU 코어 개수만큼 생성하고 각 인스턴스를 별도의 CPU 코어에 고정하여 실행하였다. 각 코어는 독립적으로 Redis 인스턴스를 실행하며, 클라이언트는 패킷에 포함된 키의 해시값을 기반으로 요청을 적절한 코어(즉, Redis 인스턴스)로 전송하도록 구성하였다.

3) 워크로드

실험에서 사용한 데이터셋은 총 100만 개의 아이템으로 구성하였으며, 각 아이템의 키의 크기는 16바이트, 밸류의 크기는 128바이트로 설정하였다 [9]. 요청되는 키의 접근 분포는 널리 사용되는 YCSB 워크로드를 고려하여 파라미터 값 0.99의 Zipfian 분포 (zipf-0.99)를 사용하였다 [10]. 기본적으로 모든 요청은 읽기 요청이며, 쓰기 요청 및 스캔 요청 비율이 성능에 미치는 영향은 별도로 실험하여 분석하였다.

2. 실험 결과

1) 처리량



[그림 6] 키 접근 패턴별 최대 처리량

그림 6은 키 접근 빈도 분포가 균일한 경우(uniform)와 zipf-0.9, zipf-0.95, zipf-0.99로 편향된 경우에 대해 최대 처리량을 측정한 결과를 보여준다. 실험 결과, 키 접근 분포가 균일할 때와 편향될 때 모두 eBPF/XDP 기반 인-메모리 키-밸류 스토어가 Redis에 비해 월등히 높은 처리량을 기록한 것을 확인할 수 있다.

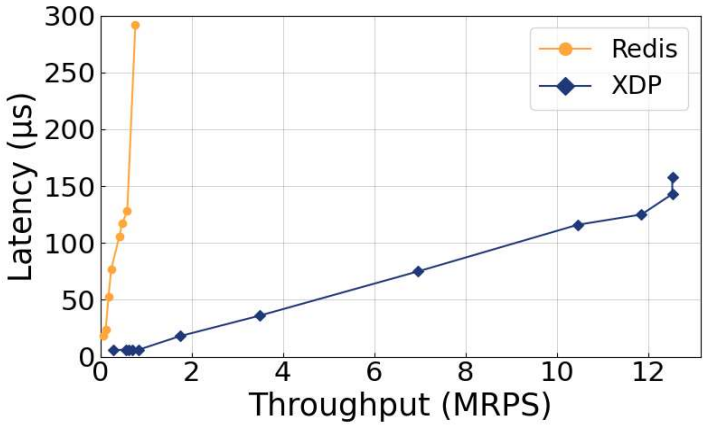
먼저 균일한 분포에서 eBPF/XDP 기반 인-메모리 키-밸류 스토어는 약 11,984 KRPS를 기록하여 Redis의 약 486 KRPS와 비교했을 때 약 24배에 달하는 높은 처리량을 보였다. 편향된 Zipf 분포의 경우, zipf-0.9, zipf-0.95, zipf-0.99 순으로 실험한 결과 각각 12,308 KRPS, 12,324 KRPS, 12,370 KRPS로 측정되었으며, 이는 Redis의 약 590 KRPS에 비해 약 20배에 달하는 성능 차이를 나타낸다.

이와 같은 성능 차이는 eBPF/XDP 기반 인-메모리 키-밸류 스토어가 네트워크 스택을 통과하지 않고 NIC 드라이버 수준에서 요청을 처리하고, 바로 응답을 반환할 수 있기 때문에 발생한다. Redis는 애플리케이션 레벨에서 동작하며, 네트워크 스택을 통과하면서 발생하는 오버헤드가 누적되므로 처리량이 제한된다.

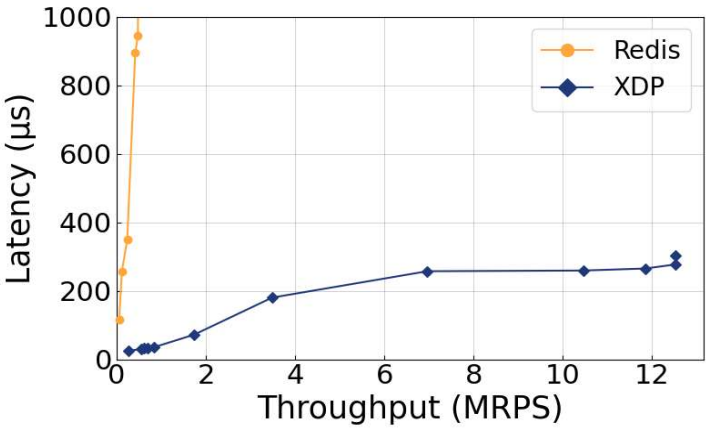
또한, 키 접근 분포에 따른 성능 변화를 살펴보면, eBPF/XDP 기반 인-메모리 키-밸류 스토어의 경우 분포가 편향될수록 처리량이 다소 증가하는 경향을 보인다. zipf-0.9에서 zipf-0.99로 갈수록 성능이 점진적으로 향상되는데, 이는 자주 요청되는 키에 대한 캐시 효율성이 높아지기 때문이다. Redis 역시 분포에 따른 처리량 증가가 관찰되었으나, eBPF/XDP 기반 시스템에 비해서는 미미한 수준으로, 네트워크 스택을 통한 접근이 지속적으로 성능 병목으로 작용하는 것을 보여준다.

이 실험 결과는 eBPF/XDP 기반 인-메모리 키-밸류 스토어가 다양한 키 접근 패턴에서 Redis보다 높은 처리량을 제공할 수 있음을 입증하며, 네트워크 스택을 우회하는 방식이 고성능 데이터 서비스에 유리한 구조임을 확인시켜 준다.

2) 지연 시간



[그림 7] 처리량 대비 중앙 지연 시간



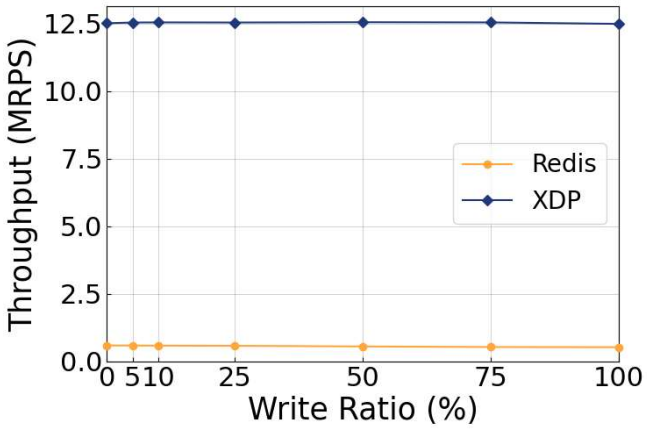
[그림 8] 처리량 대비 꼬리 지연 시간

그림 7과 그림 8은 처리량 대비 지연 시간을 그래프로 나타낸 것이다. 그림 7은 전체 지연 시간의 50%에 해당하는 지연 시간의 중앙값을 나타낸 것이고, 그림 8는 전체 지연 시간의 99%에 해당하는 꼬리 지연 시간을 나타낸 것이다. eBPF/XDP 기반 인-메모리 키-밸류 스토어는 Redis에 비해 높은 처리량을 제공하면서 매우 낮은 지연 시간을 유지함을 확인할 수 있다.

중앙 지연 시간에서 Redis의 지연 시간은 처리량이 증가할수록 눈에 띄게 상승하는 경향을 보인다. 예를 들어, Redis는 59 KRPS에서 약 18 μs의 중앙값 지연 시간을 기록하였으나, 처리량이 590 KRPS에 도달할 때는 지연 시간이 128 μs로 증가하였다. 반면, eBPF/XDP 기반 인-메모리 키-밸류 스토어의 경우 Redis와 유사한 처리량을 달성할 때 중앙 지연 시간이 6 μs로 매우 낮은 수준을 유지한다. 또한, 최대 처리량에 가까운 12,532 KRPS에서도 중앙값 지연 시간이 143 μs에 불과하여, 대규모 트래픽을 처리하는 데 강점을 보인다. 이는 eBPF/XDP 기반 인-메모리 키-밸류 스토어가 커널 네트워크 스택을 우회하여 NIC 드라이버에서 패킷을 먼저 처리하기 때문에 네트워크 스택을 통과할 때 발생하는 오버헤드가 현저히 줄어든 결과로 해석할 수 있다.

꼬리 지연 시간 측면에서도 eBPF/XDP 기반 인-메모리 키-밸류 스토어는 Redis 대비 훨씬 뛰어난 성능을 보인다. Redis는 처리량이 증가함에 따라 꼬리 지연 시간이 급격히 상승하는 경향을 보인다. 구체적으로 살펴보면, Redis는 처리량이 59 KRPS일 때 약 117 μs의 꼬리 지연 시간을 기록했으나, 처리량이 증가할수록 지연 시간이 빠르게 증가하여, 590 KRPS에서는 280,924 μs로 매우 높은 지연 시간을 보였다. 이는 고부하 환경에서 애플리케이션 수준의 오버헤드와 네트워크 스택 오버헤드가 누적되어 Redis의 꼬리 지연 시간이 급격히 증가함을 보여준다.

반면, eBPF/XDP 기반 인-메모리 키-밸류 스토어는 처리량이 증가해도 꼬리 지연 시간의 증가 폭이 제한적이다. 예를 들어, 275 KRPS에서 25 μs의 지연 시간을 기록했으며, 최종적으로 12,532 KRPS에서도 277 μs로 Redis 대비 여전히 낮은 지연 시간을 유지하고 있다. 이는 eBPF/XDP 인-메모리 키-밸류 스토어가 커널 네트워크 스택을 거치지 않고 NIC 드라이버에서 직접 패킷을 처리함으로써, 고부하 환경에서도 일관되게 낮은 지연 시간을 유지



[그림 9] 쓰기 요청 비율별 처리량

할 수 있음을 의미한다.

이와 같은 결과는 eBPF/XDP 기반 인-메모리 키-밸류 스토어가 Redis보다 상당히 낮은 중앙값 및 꼬리 지연 시간을 유지할 수 있음을 입증하며, 고부하 환경에서도 일관된 성능을 발휘할 수 있음을 나타낸다. 특히, 꼬리 지연 시간은 전체 시스템의 성능을 결정짓는 주요 요소로 작용하는데, eBPF/XDP 기반 시스템이 높은 안정성과 일정한 응답 시간을 유지한다는 점에서 대규모 데이터 센터나 실시간 데이터 처리가 요구되는 환경에 적합한 솔루션임을 확인할 수 있다.

3) 쓰기 비율의 영향

다음으로, 쓰기 요청 비율이 XDP 기반 키-밸류 스토어의 성능에 미치는 영향을 분석하기 위해 실험을 진행하였다. 그림 9은 쓰기 요청 비율을 0%에서 100%까지 증가시키며 측정한 처리량을 나타낸다. 실험 결과, 쓰기 요청 비율이 증가하더라도 eBPF/XDP 기반 인-메모리 키-밸류 스토어의 처리량 감소는 미미한 수준에 그친 반면, Redis는 쓰기 요청 비율에 따라 성능이

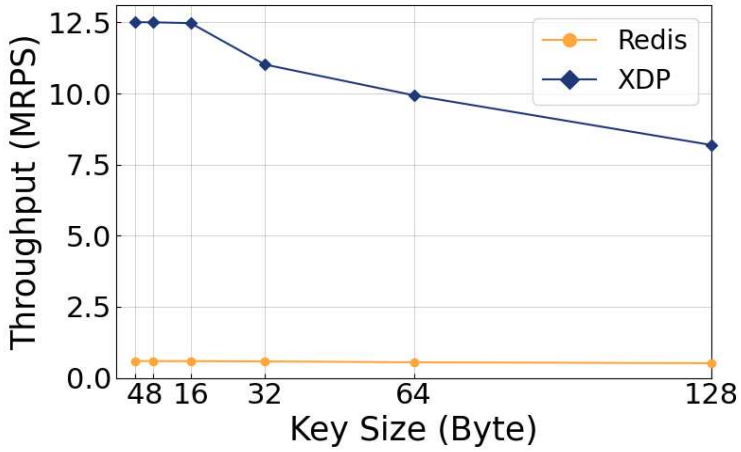
지속적으로 감소하는 경향을 보였다.

쓰기 요청 비율이 0%일 때, Redis는 약 589 KRPS의 처리량을 기록했지만, eBPF/XDP 기반 인-메모리 키-밸류 스토어는 약 12,518 KRPS의 높은 처리량을 기록하였다. 이후 쓰기 요청 비율을 점진적으로 높였을 때, Redis의 처리량은 점차 감소하여, 쓰기 요청 비율이 100%에 이르렀을 때 524 KRPS로 약 10%의 성능 저하를 보였다. 반면, eBPF/XDP 기반 인-메모리 키-밸류 스토어는 쓰기 요청 비율이 100%일 때도 12,494 KRPS로, 초기 처리량에서 단 0.19% 감소하는 데 그쳤다.

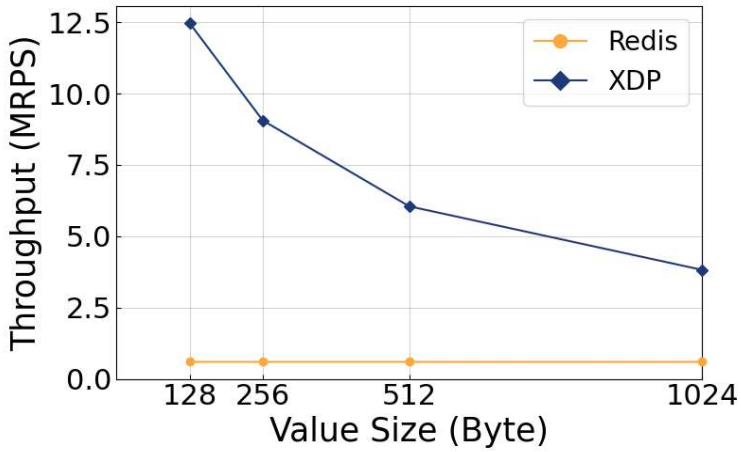
이 결과는 eBPF/XDP 기반 인-메모리 키-밸류 스토어의 읽기와 쓰기 작업에서 처리 속도 차이가 거의 없음을 시사하며, 이는 eBPF/XDP 인-메모리 키-밸류 스토어가 네트워크 스택을 우회하여 요청을 직접 처리하는 과정에서 해시 테이블 맵 조회와 갱신 속도가 거의 동일하게 빠르기 때문으로 해석할 수 있다.

본 실험의 결과는 eBPF/XDP 기반 인-메모리 키-밸류 스토어가 쓰기 요청 비율이 높은 환경에서도 효율적이고 일관된 성능을 제공할 수 있음을 입증하며, 다양한 비율의 읽기 또는 쓰기 요청을 처리해야 하는 실시간 데이터 서비스에서도 적합한 솔루션이 될 수 있음을 보여준다.

4) 데이터 크기의 영향



[그림 10] 키 데이터 크기별 처리량



[그림 11] 밸류 데이터 크기별 처리량

데이터 크기가 성능에 미치는 영향을 분석하기 위해 키와 밸류의 데이터 크기를 증가시키면서 처리량 변화를 측정하였다.

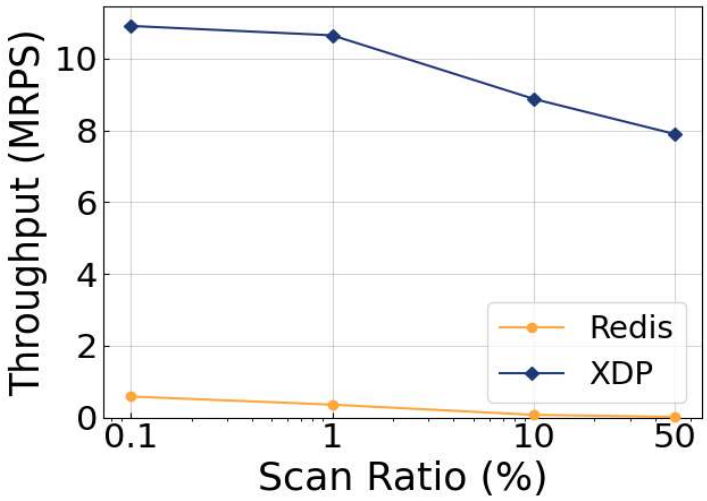
그림 10은 키의 크기를 증가시켰을 때의 처리량 변화를 나타낸다. 키 크기 증가에 따른 처리량 변화를 보면, eBPF/XDP 기반 인-메모리 키-밸류 스

토어는 키의 크기가 16바이트에서 128바이트로 증가할 때 12,473 KRPS에서 8,189 KRPS로 약 34% 감소하였다. 반면, Redis는 동일한 상황에서 589 KRPS에서 517 KRPS로 약 13.2% 감소하여, Redis보다 eBPF/XDP 기반 인-메모리 키-밸류 스토어의 성능이 데이터 크기 증가에 더 민감함을 확인할 수 있다.

밸류 크기 증가에 따른 처리량 변화는 그림 11에 나타나 있으며, 밸류 크기를 128바이트에서 1024바이트로 증가시켰을 때 성능이 큰 폭으로 감소하였다. eBPF/XDP 기반 인-메모리 키-밸류 스토어는 밸류 크기가 128바이트일 때 12,473 KRPS를 기록했지만, 1024바이트로 증가하면 3,821 KRPS로 약 70% 감소하였다. 이는 eBPF/XDP 인-메모리 키-밸류 스토어가 패킷 크기에 매우 민감하게 반응함을 보여주며, 패킷 크기가 커질수록 NIC 드라이버에서 패킷을 처리하는 데 필요한 자원이 증가하여 성능이 저하됨으로 해석할 수 있다. 그러나, 밸류 크기가 1024바이트인 경우에도 eBPF/XDP 기반 인-메모리 키-밸류 스토어는 Redis의 처리량(597 KRPS)에 비해 약 6.4배 높은 성능을 유지하고 있다.

이러한 실험 결과는 eBPF/XDP 기반 인-메모리 키-밸류 스토어가 데이터 크기 증가에 따라 성능 저하가 발생할 수 있지만, 여전히 Redis보다 월등히 높은 처리량을 제공할 수 있음을 보여준다.

5) 스캔 비율의 영향

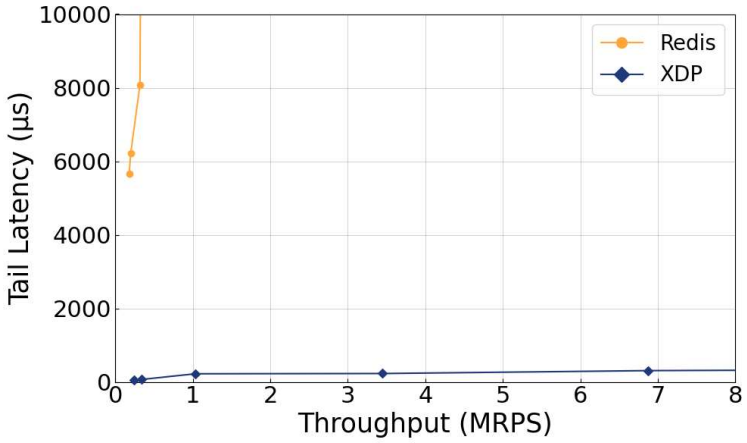


[그림 12] 스캔 요청 비율별 처리량

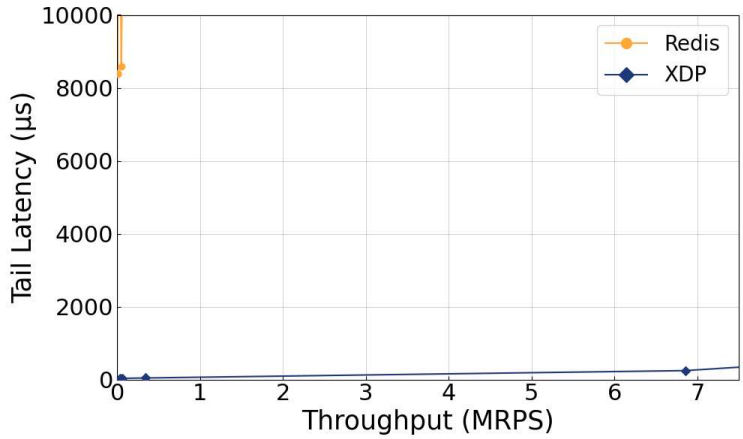
스캔(scan) 작업은 여러 개의 키를 순차적으로 조회하는 것으로, 데이터 집합의 크기가 크거나, 조회 범위가 넓을 때 성능에 큰 영향을 줄 수 있다. 본 실험에서는 스캔 요청이 읽기 작업을 100회 수행하도록 설정하였다 [14].

그림 12는 스캔 요청 비율을 0.1, 1, 10, 50으로 증가시키며 처리량을 측정한 실험 결과를 나타낸다. 실험 결과, eBPF/XDP 기반 인-메모리 키-밸류 스토어와 Redis 모두 반복적인 요청 처리 작업의 증가에 따라 성능 저하가 발생하였다. 그러나 eBPF/XDP 기반 인-메모리 키-밸류 스토어는 Redis와 비교하여 일관되게 더 높은 처리량을 유지하는 것으로 확인되었다.

이는 eBPF/XDP 기반 인-메모리 키-밸류 스토어가 반복적인 요청 처리 작업에서도 효율적으로 동작할 수 있는 설계와 최적화가 이루어졌음을 시사한다.



[그림 13] 처리량 대비 꼬리 지연 시간 (99%-읽기, 1%-스캔)



[그림 14] 처리량 대비 꼬리 지연 시간 (90%-읽기, 10%-스캔)

다음으로, 스캔 요청의 비율을 1%와 10%로 설정한 상황에서 eBPF/XDP 기반 인-메모리 키-밸류 스토어와 Redis의 처리량 대비 꼬리 지연 시간을 측정하여 성능을 비교하였다. 그림 13는 스캔 비율이 1%일 때 두 시스템의 처리량 대비 꼬리 지연 시간을 보여준다. Redis는 최대 처리량에서 46,261μs의 높은 꼬리 지연 시간을 기록하였다. 반면, eBPF/XDP 기반 인-메모리 키

-밸류 스토어는 높은 처리량을 달성하면서도 775μs 의 낮은 꼬리 지연 시간을 달성하였다.

그림 14은 스캔 비율이 10%로 증가한 경우를 나타낸다. Redis는 스캔 비율 증가로 인해 처리량이 크게 감소하고 지연 시간이 더욱 높아진 반면, eBPF/XDP 기반 인-메모리 키-밸류 스토어는 여전히 높은 처리량을 유지하며 상대적으로 낮은 지연 시간을 기록하였다. Redis의 최대 처리량에서 꼬리 지연시간은 54,774μs에 이르렀다. 반면, eBPF/XDP 기반 인-메모리 키-밸류 스토어의 최대 처리량에서의 꼬리 지연 시간은 1,214μs였다.

이러한 결과는 스캔 작업이 빈번하게 발생할 때도 eBPF/XDP 기반 인-메모리 키-밸류 스토어가 높은 처리량을 유지하며 안정적으로 낮은 지연 시간을 제공할 수 있음을 나타낸다.

V. 논의 및 향후 연구

1. eBPF 제약사항

1) 정적 메모리 할당

현재 eBPF의 정적 메모리 할당 방식은 다양한 길이의 데이터를 효율적으로 저장하는 데 한계가 있다. eBPF 맵에서 키와 밸류의 크기는 맵 생성 시 고정되며, 이후에는 변경할 수 없다. 이는 다양한 길이의 데이터를 저장하거나 처리해야 하는 경우 비효율적이다. 예를 들어, 짧은 길이의 데이터 저장 시 불필요한 메모리 사용이 발생하고, 긴 데이터의 경우 고정된 메모리 크기 제한 때문에 저장 자체가 불가능해질 수 있다. 고정된 크기 설정으로 인해 메모리 낭비와 함께 성능 문제도 발생할 수 있다. eBPF의 정적 메모리 할당 방식은 다양한 데이터 크기를 다루는 경우 확장성과 유연성에서 제한적이다. 이를 해결하기 위해 eBPF에서 동적 메모리 할당을 지원하는 것이 필요하다. eBPF에서 동적 메모리 할당을 지원하게 되면 다양한 길이의 데이터를 효율적으로 저장하고 처리할 수 있어, 메모리 사용 효율이 크게 개선될 것이다. 또한, 응용 프로그램이 요구하는 데이터 크기에 따라 메모리를 유연하게 할당하고 해제할 수 있어, 메모리 낭비를 줄이고 성능 최적화가 가능해질 것이다. 이를 통해 eBPF 프로그램이 네트워크 패킷 처리뿐만 아니라 데이터 크기가 다양한 복잡한 애플리케이션에서도 활용될 수 있을 것이다.

현재 해시 테이블 맵은 *BPF_F_NO_PREALLOC* 라는 플래그를 지원한다 [20]. 이것은 맵의 엔트리를 사전에 할당하지 않고 필요할 때 동적으로 할당하도록 설정하는 옵션이다. 이 플래그를 사용하면 맵이 생성될 때 전체 엔트리 공간을 미리 확보하지 않으며, 엔트리가 추가될 때마다 메모리를 할당

하게 된다. 기본적으로 eBPF 맵은 모든 엔트리 공간을 사전에 할당하는 방식으로 빠른 접근을 보장한다. 그러나 이 방식은 초기 메모리 사용량이 크며, 사용하지 않는 엔트리도 메모리를 차지하게 된다. 따라서 *BPF_F_NO_PREALLOC* 플래그를 설정하여 필요할 때만 메모리를 할당하도록 할 수 있다. 이를 통해 메모리 사용량을 효율적으로 관리할 수 있지만, 동적 할당에 따른 약간의 성능 저하가 있을 수 있다. 키와 밸류의 크기는 고정되지만 엔트리 수에 따라 동적으로 메모리를 할당하도록 하여 메모리 사용을 최적화할 수 있다. 향후 연구에서는 eBPF의 정적 메모리 할당 제약 하에서 데이터를 효율적으로 저장 및 관리할 수 있는 방법을 탐구하고, 다양한 크기의 데이터가 섞인 워크로드에서의 성능을 분석하고자 한다.

2) 제한된 프로그래밍

eBPF는 커널에서 안전하게 실행될 수 있도록 설계되었기 때문에 여러 가지 프로그래밍 제한이 존재한다. 가장 큰 제약 중 하나는 동적 메모리 할당의 부재와 함께, 제한된 명령어 집합과 고정된 루프 구조만을 지원한다는 점이다. eBPF 프로그램에서 사용할 수 있는 명령어는 커널 내에서 안전하게 실행될 수 있도록 제한되어 있으며, 이를 통해 시스템 안정성과 보안성을 유지하고자 한다. 그러나 이러한 제한은 복잡한 작업을 구현하거나, 다양한 연산이 필요한 응용 프로그램에서 비효율적일 수 있다.

또한, 루프 구조에 대한 제한은 eBPF 검증기가 모든 루프가 종료될 수 있음을 보장하기 위해, 루프가 반드시 고정된 횟수 내에서 종료되도록 요구한다. 이는 무한 루프나 가변 반복 횟수를 가진 루프가 허용되지 않는다는 것을 의미한다. 이로 인해, 반복 횟수가 동적으로 결정되거나 조건에 따라 유동적인 반복이 필요한 경우, eBPF 프로그램 내에서 구현이 어려워진다. 이러한 제한은 주로 복잡한 데이터 처리, 조건 기반 로직이 필요한 네트워크

애플리케이션에서 성능 문제와 유연성 부족을 초래할 수 있다.

향후 eBPF에 확장된 명령어 집합과 유연한 루프 구조, 더 강력한 동기화 메커니즘이 도입된다면, eBPF의 활용 가능성은 더욱 확대될 것이다. 특히, 다양한 데이터 처리 및 네트워크 기능 구현이 가능해져 다양한 기능을 제공하는 키-밸류 스토어를 구축할 수 있을 것으로 기대한다.

2. 데이터 백업

eBPF/XDP 인-메모리 키-밸류 스토어가 Redis처럼 영속성을 제공하기 위해서는 데이터를 백업하는 메커니즘이 필요하다. 본 논문에서 데이터 저장을 위해 사용하는 BPF 맵은 다른 eBPF 프로그램이나 유저 애플리케이션에서도 접근이 가능하다. 따라서 eBPF/XDP 기반 인-메모리 키-밸류 스토어에서 데이터의 백업 및 복원 메커니즘을 구현할 수 있을 거라고 기대한다. 예를 들어, 사용자 공간 애플리케이션에서 eBPF 맵의 데이터를 주기적으로 읽어와 디스크에 저장할 수 있다. 이를 통해 Redis의 스냅샷 기능처럼 데이터의 영속성을 제공할 수 있으며, 데이터 손실 방지와 빠른 복구가 가능하다. 또는 쓰기 요청에 대해서만 로그를 기록하여 관리할 수도 있다.

하지만 데이터를 백업할 때에는 백업 주기에 따라 시스템 성능에 미치는 영향을 고려해야 한다. 백업 주기가 짧을수록 데이터 손실 위험은 줄어들지만, 그만큼 성능 오버헤드가 발생할 수 있다. 반대로 백업 주기가 길다면 성능에는 긍정적이지만, 시스템 장애 발생 시 복구 가능한 데이터 시점이 멀어질 수 있다. 또한, 백업된 데이터의 저장 위치와 방식도 중요한 고려 사항이다.

VI. 관련 연구

eBPF와 XDP를 활용하여 시스템 성능을 향상시킨 연구들이 다수 존재한다. 이러한 연구들은 모두 XDP 수준에서 가능한 많은 패킷을 처리하여 커널 네트워크 스택 오버헤드를 회피함으로써 성능을 향상시킨다.

BMC [11]는 Memcached의 성능을 높이기 위해 설계된 커널 내 캐시 시스템으로, eBPF와 XDP를 활용하여 NIC 드라이버에서 패킷을 수신하자마자 캐시를 처리함으로써 네트워크 스택 오버헤드와 사용자 공간으로의 전환 오버헤드를 줄인다. 읽고자 하는 데이터가 캐시(BMC)에 존재할 경우 요청 패킷은 네트워크 스택을 거치지 않고 NIC 드라이버에서 처리된 후 바로 응답을 보낼 수 있어 높은 처리 성능을 제공한다. 캐시 적중 시에는 BMC에서 응답을 처리하고, 적중되지 않는 경우 패킷은 네트워크 스택 및 사용자 공간으로 넘어가 Memcached 애플리케이션에서 처리된다.

본 논문은 BMC와 유사한 아이디어를 기반으로 고성능 인-메모리 키-밸류 스토어를 설계했지만, 몇 가지 차이점이 있다. BMC는 Memcached의 캐시 용도로 설계된 시스템으로, 배열 맵을 사용해 키-밸류 데이터를 저장하고, FNV-1a [17] 해시 함수를 통해 키의 해시 값을 계산하여 배열 맵의 인덱스로 활용한다. 이 때문에 해시 충돌이 발생하면 기존 데이터를 새로운 데이터로 덮어쓰게 된다. 이는 BMC가 Memcached 캐시 역할에 초점을 맞추기 때문이다. 반면, 본 논문에서 제안하는 eBPF/XDP 기반 인-메모리 키-밸류 스토어는 독립적으로 작동하는 시스템으로, 해시 테이블 맵을 사용하여 키와 밸류 데이터를 그대로 저장한다. 또한, 리눅스 커널에서 해시 충돌이 발생할 경우 체이닝 방식을 통해 충돌을 관리한다는 점에서 차별화된다.

Electrode [15]는 eBPF와 XDP를 활용하여 Paxos와 같은 분산 프로토콜의 성능을 개선하였다. Electrode는 분산 시스템에서 자주 발생하는 메시지 브

로드캐스팅, 빠른 ACK 응답, 쿼럼 확인 등 분산 프로토콜의 핵심 작업을 커널 내 eBPF로 오프로드함으로써 커널 네트워킹 스택의 오버헤드를 줄이고, 처리량과 지연 시간을 개선한다. 특히 Electrode는 XDP 및 TC 레벨에서 메시지 처리를 통해 사용자-커널 간 컨텍스트 스위칭과 네트워킹 스택을 여러 번 통과하는 과정을 줄여준다.

Dint [16]는 eBPF를 활용하여 리눅스 커널 내에서 고성능 분산 트랜잭션 시스템을 설계하였다. 트랜잭션의 주요 연산인 lock 관리, 키-밸류 저장, 로깅(logging)을 eBPF를 통해 커널에 오프로드하여 네트워크 스택 오버헤드를 피하고 커널과 유저 공간 사이 컨텍스트 스위칭을 최소화하였다.

커널 네트워크 스택의 오버헤드를 회피하여 인-메모리 키-밸류 스토어의 성능 향상을 위한 연구도 진행된 바 있다.

MICA [6]는 커널 바이패스 기술인 DPDK [19]를 사용하여 커널 네트워크 스택을 완전히 우회하여 모든 키-밸류 쿼리를 사용자 공간에서 처리한다. 이 과정에서 해시맵을 각 코어에 파티셔닝하여 동기화 오버헤드를 줄이고, 클라이언트 정보를 기반으로 특정 코어에 쿼리를 매핑하는 전용 프로토콜을 사용한다. MICA는 DPDK 라이브러리를 통해 구현되었기 때문에, DPDK의 제약 사항을 따르게 된다. 예를 들어 패킷 수신을 위해 전용 CPU 코어가 필요하고, 하드웨어에 의존적이라는 특징 때문에 기존 애플리케이션의 구조를 상당히 수정해야 한다. 반면 eBPF/XDP 기반 인-메모리 키-밸류 스토어는 기존 애플리케이션의 구조 변경 없이도 구현할 수 있어 개발 난이도가 낮고, 리눅스 커널이 제공하는 보안 검증 및 제어 메커니즘을 유지하면서도 높은 성능을 달성할 수 있다.

KV-Direct [18]는 RDMA 기능을 확장하여 프로그래머블 NIC를 활용한 고성능 인-메모리 키-밸류 스토어를 제안한다. KV-Direct는 호스트 CPU를 우회하고, 호스트 메모리에서 직접 데이터를 가져와 키-밸류 요청을 처리하고

업데이트를 수행한다. 이러한 구조는 키-밸류 연산을 프로그래머블 NIC에 오프로드하여 성능을 높이는 방식이므로 특수한 하드웨어가 요구된다. 따라서 일반적인 환경에서는 널리 사용되기 어렵다.

반면, eBPF/XDP 기반 인-메모리 키-밸류 스토어는 특수한 하드웨어 장비 없이도 구현할 수 있으며, 네이티브 모드로 실행하려면 NIC 드라이버가 XDP를 지원하기만 하면 된다. Mellanox, Broadcom 등의 XDP 지원 NIC가 데이터센터에서 널리 사용되고 있다. 만약 NIC 드라이버가 XDP 프로그램을 지원하지 않는 경우에는, 리눅스 커널 4.12 버전부터 제공하는 일반 모드로 실행할 수 있다. 일반 모드는 네트워크 스택 상단에서 실행되어 성능 향상 효과는 줄어들지만, 유저 공간으로 패킷이 전달되기 전에 처리가 가능하다. 이처럼 eBPF/XDP 기반 인-메모리 키-밸류 스토어는 배포와 유지 관리가 상대적으로 용이하다는 장점이 있다.

Ⅶ. 결론

본 논문에서는 널리 사용되는 인-메모리 키-밸류 스토어의 성능을 제한하는 커널 네트워크 스택 오버헤드를 측정하고, 이를 제거하기 위해 eBPF와 XDP를 활용하는 방법을 제안한다. 가장 널리 사용되는 인-메모리 키-밸류 스토어인 Redis를 활용하여 커널 네트워크 스택 오버헤드가 지연 시간에 미치는 영향을 분석하였으며, 이러한 오버헤드를 최소화하기 위해 새로운 eBPF/XDP 기반 인-메모리 키-밸류 스토어를 설계하였다. eBPF/XDP 기반 인-메모리 키-밸류 스토어는 XDP 프로그램을 통해 해시 테이블 맵에 키-밸류 쌍을 저장하거나 조회한 후, 클라이언트에 즉시 반환한다. 패킷이 수신되는 즉시 NIC 드라이버에서 요청을 처리하므로 높은 처리량과 낮은 지연 시간을 달성할 수 있음을 확인하였다. 실험 결과, eBPF/XDP 기반 인-메모리 키-밸류 스토어는 Redis 대비 최대 24배 높은 처리량을 달성하면서 현저히 낮은 지연 시간을 달성하는 것을 확인할 수 있었다.

그러나 eBPF/XDP 기반 인-메모리 키-밸류 스토어는 eBPF 기술의 특성상 메모리를 정적으로 미리 할당해야 하는 한계가 존재한다. 이에 향후 연구에서는 eBPF/XDP 기반 인-메모리 키-밸류 스토어의 메모리 측면에서의 한계점을 분석하고 다양한 크기의 패킷이 혼합된 워크로드에서의 성능을 평가할 예정이다. 또한, 데이터 영속성을 보장하기 위한 백업 메커니즘을 설계 및 구현하는 연구를 진행할 계획이다.

참고문헌

[1] Redis. http://redis.io.

[2] Memcached. https://memcached.org/.

[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, “Scaling Memcache at Facebook,” in *Proc. of USENIX NSDI,* 2013.

[4] J. Yang, Y. Yue, and K. V. Rashmi, “A large scale analysis of hundreds of in-memory cache clusters at Twitter,” in *Proc. of USENIX OSDI,* 2020.

[5] J. Yang, Y. Yue, and R. Vinayak, “Segcache: a memory-efficient and scalable in-memory key-value cache for small objects,” in *Proc. of USENIX NSDI,* 2021.

[6] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage,” in *Proc. of USENIX NSDI,* 2014.

[7] 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 *Proc. of ACM CoNEXT*, 2018.

[8] eBPF. https://ebpf.io/.

[9] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, “Netcache: Balancing key-value stores with fast in-network caching,” in *Proc. of ACM SOSP,* 2017.

[10] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears,

“Benchmarking cloud serving systems with YCSB,” in *Proc. of ACM SOCC*, 2010.

[11] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and Gilles Muller, "BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing," in *Proc. of USENIX NSDI,* 2021.

[12] RocksDB. https://rocksdb.org/.

[13] LevelDB. https://github.com/google/leveldb.

[14] G. Kim, “NetClone: Fast, Scalable, and Dynamic Request Cloning for Microsecond-Scale RPCs,” in *Proc. of ACM SIGCOMM*, 2023.

[15] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating Distributed Protocols with eBPF." in *Proc. of USENIX NSDI*, 2023.

[16] Y.Zhou, X. Xiang, M. Kiley, S. Dharanipragada, and Minlan Yu, "DINT: Fast In-Kernel Distributed Transactions with eBPF," in *Proc. of USENIX NSDI*, 2024.

[17] D. Eastlake, T. Hansen, G. Fowler, K.-P. Vo, and L. Noll. The FNV Non-Cryptographic Hash Algorithm. 2019.

[18] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, “Kv-direct: High-performance in-memory key-value store with programmable nic,” in *Proc. of ACM SOSP,* 2017.

[19] Linux Foundation. 2018. Data Plane Development Kit. https://www.dpdk.org/

[20] BPF_MAP_TYPE_HASH. https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_HASH/

ABSTRACT

Performance Analysis of eBPF/XDP-based In-memory Key-value stores

Lee, Jihyun
Department of Computer Science
Graduate School of
Sungshin Women's University

In-memory key-value stores are a fundamental component of modern online services. However, passing through the kernel network stack introduces significant overhead, which increases data access latency. In this paper, we design and evaluate the performance of an in-memory key-value stores leveraging eBPF and XDP (eXpress Data Path), a programmable packet processing technology. XDP enables packet processing at the NIC driver level, bypassing the kernel network stack and reducing overhead. We conducted experiments comparing the performance of the eBPF/XDP-based in-memory key-value stores to Redis, a widely used in-memory key-value store. The results show that the eBPF/XDP-based in-memory key-value store achieves up to 24× higher throughput than Redis while maintaining lower latency.

ACKNOWLEDGEMENTS

학위과정을 무사히 마칠 수 있도록 도와주신 분들께 감사를 표합니다.

먼저 지도교수이신 김규영 교수님께 진심으로 감사드립니다. 2022년 1학기에 학부연구생으로 시작한 이후 약 3년 동안 교수님께 많은 가르침을 받았습니다. 교수님께서 다양한 배경 지식들을 아낌없이 알려주시고 저의 연구에 많은 조언을 해주신 덕분에 한층 성장할 수 있었으며, 이렇게 논문을 완성할 수 있었습니다. 연구에 집중할 수 있도록 신경 써주신 교수님께 진심으로 감사의 말씀 올립니다.

또한, 바쁘신 와중에도 학위논문 심사위원을 맡아주신 박지웅 교수님, 임연섭 교수님께 감사드립니다. 교수님들의 귀중한 코멘트 덕분에 논문을 완성할 수 있었습니다.

연구실 동기 방지윤을 비롯하여 지금도 열심히 연구에 매진하고 있는 탁유제 언니, 김정은, 조은재, 이나경 후배들에게도 감사합니다. 함께 의견을 나누면서 더욱 성장할 수 있었고 연구실 생활을 즐겁게 할 수 있었습니다.

석사과정 동안 저를 위로해주고 격려해준 친구들이 많습니다. 늘 같은 고민을 나누고 서로를 응원해주는 김승현, 김지연, 송유진, 신명지, 양성원, 이지우, 최우영, 황지수 등 컴퓨터공학과 20학번 동기들에게 감사합니다. 학교에서 마주칠 때마다 나누었던 반가운 인사와 대화 덕분에 힘을 얻을 수 있었습니다. 예전처럼 자주 만나지 못하지만 만날 때마다 10년 전이나 다름없이 큰 웃음을 주는 김나린, 박지원에게도 정말 감사합니다. 꾸준히 연락을 이어온 서정희, 김예지 언니, 변정아, 이애은, 진송현 언니에게도 감사합니다. 일상 대화 속 짧은 응원의 메시지도 큰 힘이 되었습니다.

마지막으로 제가 선택한 모든 일에 무조건적인 응원을 보내주신 부모님께도 감사와 사랑을 드립니다.