eBPF를 활용한 스토리지 내 연산처리 아키텍처

정지윤 O, 이영재, 신동군
성균관대학교 정보통신대학
wjdwldbs1@skku.edu, yjlee4154@gmail.com, dongkun@skku.edu

In-Storage Processing architecture using eBPF

JeeYoon Jung O, Youngjae Lee, Dongkun Shin
College of Information and Communication Engineering, Sungkyunkwan University

요 약

최근 빅데이터 처리 및 머신 러닝 등에서 다루는 데이터 및 데이터 연산이 증가함에 따라 SSD 저장장치 내부에서 data processing을 수행하는 In-Storage Processing(ISP)기법이 활발하게 연구되고 있다. ISP는 data intensive workload에서 CPU의 연산 overhead 및 I/O의 양을 현저히 줄일 수 있다. 하지만 기존 ISP 연구들은 각각 고유의 프로그래밍 인터페이스 및 라이브러리를 활용하고 있어 표준화된 프로그램 환경이 없는 실정이다. 본 연구는 기존에 In-Kernel Processing을 위해서 활용되고 있는 eBPF를 표준 ISP 프로그래밍 인터페이스로 활용하는 방안을 제안한다. eBPF를 실행하기 위한 SSD runtime을 제안하고, 이를 실제 장치인 Cosmos+ OpenSSD에 구현하여 그 효과를 확인하였다. 추가적으로 16개의 SSD를 활용하였을 때, grep workload에서 host 대비 6배의 성능 향상을 보였다.

1. 서 론

최근 데이터 처리 및 머신 러닝 등에서 다루는 데이터의 크기가 점점 증가하고 있다. 그런데 대량의 데이터를 SSD에서 Host 메모리로 이동시키는 것은 Communication 비용이 많이 소모된다. 따라서, 데이터의 이동을 최소화하고, 연산작업을 SSD 내부로 이동시키는 NDP(Near Data Processing)를 활용하는 많은 연구가 진행되고 있다. In-Storage processing(ISP)[1]는 SSD 내부에서 연산을 수행하, 연산 이후의 결과 데이터만을 전송하여 data의 traffic 및 Host CPU의 연산 overhead를 줄여주는 NDP 기법 중 하나이다.

ISP의 효율성이 입증되면서 ISP를 다룬 다양한 논문들이 등장하였다. 하지만 이 논문들은 각각 고유의 programming interface을 통해 ISP를 사용하고 있다. biscuit[2]은 flow-base programming interface을 활용함으로써, SSD 내부에서 ISP를 제공한다. 반면에, Catalina[3]는 SSD에 full-fledged OS를 탑재하고 container 기반 프로그램 실행 환경을 제공하여 프로그램 오프로딩이 가능하도록 했다. 이 외에도 다양한 논문들이 고유의 ISP 구동 방식을 보여주고 있다. 이렇게 파편화된 인터페이스는 ISP의 확산에 걸림돌이 되고 있다.

본 연구는 이 문제에 대한 해결책을 In-Kernel Processing (IKP) model인 eBPF[7]에서 찾는다. eBPF는 커널의 특정 hook point에 user가 수행해야하는 연산을 offloading 시켜 수행할 수 있는 runtime을 제공한다. eBPF의 특징은 첫째, 커널의 소스 코드 수정 없이 특정 hook point에 user가 필요한 연산을 쉽게 offloading 시킬 수 있다. 둘째, 강력한 Verifier을 통해 offloading 된 code의 안정성을 보장한다. 셋째 user가 code를 offloading 할 때 JIT(just-in-time) compile을 통해 eBPF의 ISA(instruction set architecture)을 CPU 고유의 Native 언어로 변환시켜 빠른 성능을 보장한다. 넷째 커널과 runtime이 공유하는 메모리 공간인 MAP을 제공하여, runtime/커널간 통신을 보장하며, eBPF code를 수행하기 위해 필요한 helper function을 제공한다. 마지막으로, C기반 compile이 가능해 쉬운 programming이 가능하다. 이런 특징을 가진 eBPF는 커널에 연산을 offloading 시키는 방식으로 광범위하게 사용되고 있으며, 커널뿐만 아니라 IO 장치인 NIC(network interface card) 내부로 eBPF 코드를 offloading하여 실행하는 In-NIC-processing의 효율성도 증명되었다[4].

본 연구에서는 eBPF를 ISP에 적용하기 위해서 SSD용 eBPF runtime 아키텍처를 제안한다. eBPF를 SSD 내부에서 지원하기 위해서는 아래와 같은 과제들이 해결되어야 한다. 첫째, eBPF 실행이 기존 SSD의 펌웨어 동작을 방해하지 않도록 isolation이 필요하다. 둘째, eBPF 기반 ISP 프로그램이 SSD 내부 자원을 쉽게 접근할 수 있도록 MAP과 helper function 기능을 제공해야 한다. 셋째, SSD Firmware 내부에 다양한 hook point를 정의하여 triggering 방식의 eBPF 코드 실행을 지원해야 한다. 넷째, eBPF의 안정성과 효율성을 위해 Verifier 및 JIT compiler의 기능을 제공해야 한다.

본 연구는 위 과제를 해결하기 위해서, 첫째, eBPF Core와 Firmware Core을 분리하여 기존 펌웨어 성능이 ISP 응용에 의해서 영향을 받지 않도록 보장한다. 둘째, NVMe 표준에서 정의된 CMB[5] 기능을 활용해 eBPF core와 Host가 공유하는 메모리 공간을 제공하여 이를 통해 eBPF MAP의 기능을 구현했다. 또한, 플래시 I/O를 위한 helper function을 정의하여 eBPF가 이를 활용해 플래시메모리의 데이터를 접근할 수 있도록 했다. 셋째, FTL의 read/write에 hook point를 설정하여 I/O의 call back 동작으로 eBPF runtime을 triggering 하는 기능을 제공한다. 넷째. eBPF를 위한 Verifier와 ARM32 아키텍처에 적합한 경량화 된 JIT compiler를 제공한다.

본 연구에서는 위의 과제를 모두 해결한 아키텍처를 ISP-eBPF라 명칭하고, 이를 Cosmos+ OpenSSD[6]에 구현해 eBPF Emulation 환경에서의 runtime 및 JIT을 활용한 eBPF runtime의 성능을 측정한다. 또한, Host에서의 성능과 SSD 내부에서 eBPF를 활용한 성능을 비교한다. 2개의 SSD를 활용하였을 때, Host 대비 우수한 성능을 보이고, 16개의 SSD를 활용하였을 때 Host 대비 6배의 성능 향상을 확인했다.

2. 배경 지식 및 선행 연구

eBPF는 Linux 커널로 연산을 offloading하기 위해서 사용되는 sandboxing을 제공하는 가상머신 환경이다[7]. eBPF 가상머신은 고유의 ISA를 활용하며, 무한루프 방지, 제한된 stack size등의 제약이 있는 C언어를 통해 작성이 가능하다. 이런 제약조건을 극복하기 위해, eBPF는 Helper function 기능을 제공한다. 사전에 커널 내부에서 구현된 helper function을

이 논문은 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.IITP-2017-0-00914, 지능형 IoT 장치용 소프트웨어 프레임워크)

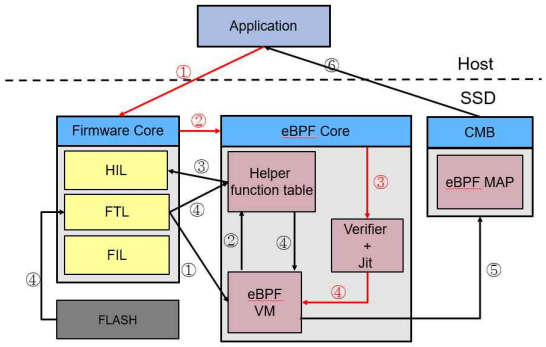


그림 1. ISP-eBPF 의 전체 구조 및 Load/Execute 과정

eBPF 코드가 호출하여 다양한 기능을 구현할 수 있다. 예를 들어, 커널에서는 checksum 기능이나 커널 메모리의 일부분을 접근하기 위한 helper function을 제공하고 있다. 또한, 커널 메모리의 일부분을 읽기 위해 eBPF는 Map 기능을 제공하는 데, 이는 type, ID, size로 구성된 메모리 영역이고 compile time에 정해지는 영역이다. eBPF Map은 eBPF runtime 및 user program이 ID를 통해 접근 가능하며, 이를 통해 eBPF runtime간 통신 및 user program과 eBPF간의 통신을 가능하게 해준다.

User는 GCC 혹은 LLVM[8]과 같은 컴파일러를 통해 바이너리 파일을 만들 수 있다. 하지만 커널에서 eBPF ISA를 활용한 가상머신을 돌리게 된다면 프로세서의 고유 언어로 컴파일 된 바이너리에 비해 매우 나쁜 성능을 보이게 된다. 그래서, 현재 커널은 BPF를 위한 JIT(Just-In-Time) compile 기능을 제공한다. JIT compile을 통해 eBPF의 ISA를 프로세서의 고유 ISA로 변경시켜, Emulation 환경에 비해 더 빠른 성능을 제공할 수 있다.

eBPF를 활용하는 연구는 다양한 분야에서 존재한다. hXDP[4]는 eBPF의 연산 작업을 NIC로 offloading 하여, 연산을 하드웨어 가속기를 통해 수행한다. 이를 통해 네트워크 패킷 처리 작업의 연산 속도를 크게 상승시킨다. 본 연구는 target을 네트워크가 아닌 스토리지 내부로 변경해 진행하며, 하드웨어가 아닌 ARM processor을 통해 연산 처리를 진행한다. BPF For Storage[9]는 eBPF를 활용해 커널의 복잡한 스토리지 스택을 단순화한다. 이 연구는 처음으로 스토리지 시스템에 eBPF를 도입한 연구이다. 하지만 결국 커널에서의 eBPF 활용에 대한 연구이고, 본질적으로 In-Storage Processing을 위한 eBPF에 대한 제안은 존재하지 않는다.

3. ISP-eBPF Architecture

이 절에서는 ISP-eBPF의 전체적인 아키텍처에 대한 설명 및 내부적인 기능에 대해 설명한다. 추가적으로 ISP-eBPF에 eBPF 코드를 넣어주는 eBPF Load와 ISP-eBPF을 실행시키는 eBPF Execute에 대해 설명한다.

3-1. Overall Architecture

우리는 ISP-eBPF의 구현을 Cosmos+ OpenSSD 보드 환경에서 구축하였다. ISP-SSD 내부 아키텍처는 bare-metal을 기반으로 구성하였으며, 전반적인 구조는 그림 1과 같다.

Firmware Core: Firmware Core는 기존 SSD의 기능들을 수행하기 위해, Host와 통신을 담당하는 HIL(Host interface layer), Address translation을 담당하는 FTL(Flash Translation Layer), Flash 내부 실제 I/O를 담당하는 FIL(Flash Interface Layer)로 구성된다.

BPF core: eBPF runtime을 수행하기 위해, SSD는 기존의 I/O 작업 및 eBPF runtime을 위한 연산작업을 동시에 진행해야 한다. eBPF runtime으로 인한 기존 I/O 작업의 방해를 최소한으로 하기 위해, 기존 SSD가 제공하는 모든 기능은 기존 Firmware Core가 담당하고, BPF Core은 ISP용 Computing 작업만 수행한다. 또한, ISP-eBPF 생성 및 실행 혹은 ISP-eBPF 수행 시 발생하는 Core간 통신은 OCM(on-chip memory)을 통해 진행하여, 이를 통해 Core간 통신 overhead를 최소화한다.

Verifier + JIT compiler: eBPF의 ISA는 SSD 내부 프로세서의 고유 ISA와 다르기 때문에 Emulation을 통해서 실행하면 상당한 성능 저하가 있다. 그러므로, eBPF core 내부에 JIT Compiler를 구현하여, Emulation 환경 대비 빠른 ARM native ISP-eBPF 환경을 제공한다. 또한 eBPF code의 안정성을 보장하기 위해 기존 커널에서 활용하는 Verifier을 SSD 내부에 구현하였다.

Helper function Table: Firmware Core에서 SSD에서 제공하는 모든 기능들을 담당하고 있기 때문에, eBPF core의 instruction 만으로는 플래시 접근이 불가능하다. 우리는 eBPF의 Helper function 기능을 활용하여 플래시 접근용 Helper function을 제공한다. 이 Helper function을 통해 ISP-eBPF가 Firmware Core와 통신하여 플래시 Read/Write가 가능하도록 했다.

eBPF MAP: ISP-eBPF와 Host간 통신 및 데이터 공유를 위한 메모리 공간이 필요하다. 커널 eBPF에선, 이를 Map으로 칭하고 있으며, 커널은 이를 통해 ISP-eBPF와 데이터를 공유한다. NVMe interface의 CMB (Controller Memory Buffer)[5]는 Host와 SSD가 공유하는 메모리 영역이며, Host는 CMB영역을 자신의 메모리 영역인 것처럼 활용이 가능하다. 이 공간을 통해 기존 커널에서 활용하는 Map의 기능과 동일한 역할을 할 수 있는 인터페이스를 제공한다.

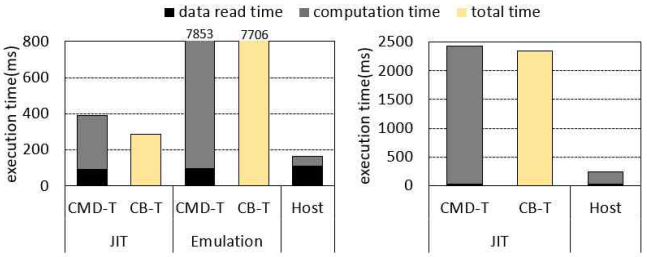
3-2 eBPF Load

SSD 내부에 eBPF 코드를 넣어 ISP-eBPF을 생성하기 위해선 eBPF load 과정이 필요하다. ISP-eBPF의 Load과정은 그림 1의 빨간색 선과 같다. 먼저 NVMe Command를 통해 eBPF load에 필요한 데이터를 전송해준다(①). 이때 전송해주는 정보는 Host에서 eBPF ISA로 컴파일 된 binary 파일과 Instruction의 size, ISP-eBPF ID를 전송해준다. 다음으로, Firmware Core가 바이너리파일 및 load에 필요한 정보를 eBPF Core에게 전달해준다(②). 이때 OCM을 활용해 데이터를 전달해 Core간의 통신 overhead를 최소화한다. eBPF core는 전달받은 데이터를 바탕으로 Verify 및 JIT compile을 수행한다(③). Cosmos+는 ARM32를 활용하므로, JIT을 통해 ARM32 native instruction을 생성한다. 마지막으로, 이 ARM instruction을 활용해 ISP-eBPF를 최종적으로 생성한다(④).

3-3 eBPF Execute

SSD에서 ISP-eBPF를 실행시키기 위해선 eBPF Execute 과정이 필요하다. ISP-eBPF를 수행시키기 위해 Command trigger 방식과 Hook Point에서의 Call-back trigger 방식을 제공한다. Command trigger 방식은 NVMe Command를 통해 ISP-eBPF를 실행시키는 방식이며, Call-back trigger 방식은 특정 펌웨어 함수를 hook point로 지정하고, 그 함수의 동작이 끝났을 때 call back으로 ISP-eBPF가 실행시키는 방식이다. 예를 들어, Nand Read 함수에 hook point를 설정하면, FLASH에서 데이터가 SSD DRAM으로 read 될 때마다 ISP-eBPF를 수행시킨다.

그림 1의 검은색 선은 ISP-eBPF의 실행과정이다. 먼저, FTL은 Call-back trigger 혹은 Command trigger에 의해 ISP-eBPF를 실행시킨다(①). 만약 eBPF code에서 Flash 접근이 필요한 경우, 플래시 접근용 Helper function을 호출한다(②). Helper function은 필요한 logical page 및 length를 HIL layer



(a) 32 MB Grep workload (b) 10 MB Word Count workload

그림 2. Workload 별 수행시간

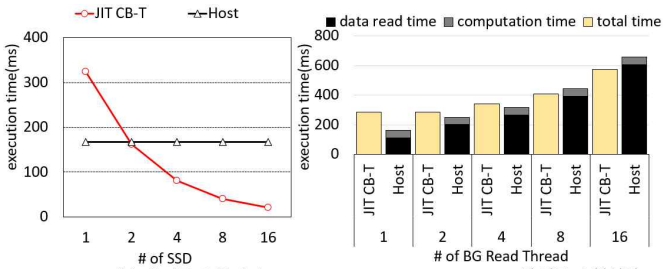
에 전달한다(③). Firmware Core은 Helper function에게 받은 정보를 바탕으로, Flash I/O를 수행하게 되며 Flash의 DATA가 SSD DRAM에 다 loading 되게 되면, data가 담긴 주소 값을 Helper function에게 전달해 주고, 최종적으로 이 주소 값을 eBPF runtime에게 전달한다(④). NAND write 또한 비슷한 방식으로 helper function을 통해 수행된다. 이후, ISP-eBPF가 Code 수행을 마치게 되면, 결과 값을 CMB에 할당된 eBPF MAP에 기록한다(⑤). Host는 해당하는 ISP-eBPF의 ID를 통해 결과 값이 담겨있는 CMB영역을 접근하여 eBPF execution의 결과값을 읽어오게 된다(⑥).

4. 실험

우리는 ISP-eBPF를 1GHz Dual-core ARM-cortex A9이 탑재된 Cosmos+ OpenSSD 보드를 사용하여 구현하고 실험하였다. Host PC는 4GHz quad-Core Intel Core i7-4790을 사용하였다. 실험에서는 Emulation 기반 ISP-eBPF 환경과 JIT을 통해 만들어진 ISP-eBPF 환경을 Host와 비교한다. NVMe Command를 통해 ISP-eBPF를 trigger 시킨 것을 CMD-T로 명칭했으며, Hook point에서 Calll-back방식을 통해 ISP-eBPF를 trigger 시킨 것을 CB-T로 명칭한다. CB-T의 경우 eBPF Core와 Firmware Core가 독립적으로 수행되기 때문에 연산작업과 data read작업이 병렬화 되어, data read time과 computation time으로 나누어 측정하지 않고 total time 하나로 측정하였다.

그림 2은 Host 및 ISP-eBPF에서 Grep과 Word Count workload의 수행시간을 나타낸 그래프이다. Grep의 경우 I/O의 양 대비 연산이 적은 workload이며, Word Count의 경우 Grep에 비해 연산량이 더 많은 workload이다. 그림 2(a)는 Grep workload에서의 수행 시간을 나타낸다. JIT 환경의 ISP-eBPF는 Emulation 환경의 ISP-eBPF보다 약 1/20배로 짧은 수행시간을 가진다. 이는 eBPF ISA를 ARM ISA로 변환하는 JIT compiler의 효율성을 보여준다. Host와 JIT CMD-T의 data read time을 비교했을 때, JIT CMD-T의 경우가 더 짧은 수행시간을 가진다. 이는 Host data read time에는 ISP 환경과 동일하게 NAND에서 SSD DRAM으로 data를 읽어오는 시간에, Host 커널 DRAM으로의 DMA, Host 커널 DRAM에서 Host User DRAM 영역으로의 이동이 추가된다. 하지만 전체 수행시간은 Host가 JIT CMD-T 대비 더 짧으며, data read와 연산작업이 병렬화 되는 JIT CB-T와 비교해 봐도 Host의 수행시간이 더 짧다. 이는 SSD ARM core와 Host X86 core의 clock speed의 차이에 의한 연산성능차이 때문이다. 이 현상은 연산이 더 많은 workload에서 더 심하게 나타난다. 그림 2(b) Grep 대비 연산량이 많은 Word Count workload에서 Host와 JIT 환경의 ISP-eBPF의 수행시간을 비교한 그래프이다. 연산량의 증가에 따라 Host의 수행시간과 eBPF-ISP의 수행시간 차이가 Grep workload에 비해 더 커지게 된다.

위 실험에서, ISP-eBPF의 성능이 Host 대비 더 낮은 이유는 (1) SSD를 1대만 사용하였기 때문에 낮은 성능의 core에



(a) SSD 개수에 따른 수행시간 (b) Back ground thread에 따른 수행시간

그림 3. SSD수/ BG Read Thread에 따른 수행시간

의한 성능 저하가 더 크게 드러났으며, (2) ISP에 유리한 I/O heavy한 workload가 아니기 때문이다. 따라서 Grep workload에서 SSD를 여러 개 사용하였을 경우의 예측실험과 I/O heavy workload에서의 실험을 진행했다. 그림 3(a)는 SSD 개수에 따른 Host와 JIT CB-T의 수행 시간 변화 예측 그래프이다. Host는 SSD를 여러 개 사용하더라도 성능 증가가 없을 것으로 예측되나, hook의 경우 여러 SSD를 활용하면, data read 및 연산 성능이 SSD 개수의 곱만큼 증가하게 될 것이다. 따라서 2개의 SSD 부턴 Host에서의 성능보다 앞서고, 16개의 SSD를 활용하면 호스트 대비 6배의 성능 증가가 될 것으로 예측된다. 그림 3(b)는, background read thread의 증가에 따른 Host와 JIT CB-T 방식의 수행시간 그래프이다. Background thread는 eBPF-ISP가 수행되는 동일한 SSD에서 sequential read를 수행한다. Background thread가 4개일 때까지는 JIT CB-T의 연산 성능에 의한 손해가 더 크게 드러나서 Host 대비 더 느린 수행시간을 가진다. 하지만 Background thread가 8개가 넘어가면 JIT CB-T의 연산이 data read에 비해 더 빨리 처리가 되어, Host 대비 더 빠른 data read로 인해 더 빠른 수행 시간을 가지게 된다. Back ground thread가 16개가 되면, JIT CB-T의 경우 Host의 data read 성능보다 더 빠른 성능을 가진다.

5. 결론 및 향후 계획

본 연구에서는 eBPF를 활용한 ISP 구조를 제안하고, 이를 Cosmos+ OpenSSD에 구현하였다. 또한 JIT Compiler을 통해 Emulation 환경 대비 성능을 크게 증가시켰다. 또한 본 연구의 ISP-eBPF 환경에서 하드웨어 가속기을 통해 연산 성능을 증가시킨다면, ISP에 유리한 환경이 아니더라도, Host 대비 더 빠른 성능을 낼 수 있을 것으로 기대한다.

참고 문헌

[1] R. Balasubramonian et al., “Near-data processing: Insights from a micro-46 workshop,” Micro, IEEE, 2014.
[2] B. Gu et al., “Biscuit: a framework for near-data processing of big data workloads,” ISCA, 2016.
[3] M. Torabzadehkashi, et al., “Computational storage: an efficient and scalable platform for big data and hpc applications,” Journal of Big Data, 2019.
[4] B. Macro, et al., “hXDP: Efficient Software Packet Processing on FPGA NICs,” OSDI, 2020.
[5] NVM Express Specification. URL https://nvmexpress.org/ 2020.
[6] J. Kwak, et al., “Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems,” ACM Trans. Storage, 2016
[7] A. M. Marcos, et al., “Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications,” ACM Comput. Surv, 2020
[8] C. Lattner, et al., “LLVM: A Complication Framework for Lifelong Program Analysis and Transformation,” CGO, 2004.
[9] Y. Wu, et al., “BPF for storage: an exokernel-inspired approach,” HotOS, 2021.