eBPF의 system call 트레이싱에 따른 성능 분석

이수현, 박태준*
전남대학교(학부생), *전남대학교(교수)
leesuhueon20020711@jnu.ac.kr, *taejune.park@jnu.ac.kr

Performance Analysis of eBPF according to System Call Tracing

Suhyeon Lee, Taejune Park*
Chonnam National Univ., *Chonnam National Univ.

요 약

네트워크 기술의 발전과 더불어 네트워크 장애를 신속히 감지하고 처리하는 기술의 중요성이 더욱 부각되고 있다. 그러나 기존 방식은 커널 내의 모든 정보를 복사하여 처리하였기에 성능이 느렸다. eBPF가 등장함으로써 커널에 코드를 직접 삽입하여 효율적인 이벤트 감지와 패킷 필터링이 가능해졌다. 그러나 이러한 과정은 이벤트 감지 시 오버헤드를 동반하여 프로그램 성능을 저하할 수 있다. 따라서 본 논문에서는 system call 호출 시 트레이싱 유무와 관측 개수가 실행 시간에 미치는 영향에 대해 알아보고자 한다.

Ⅰ. 서 론

커널 내에서 발생하는 다양한 이벤트를 감지하고 조작하기 위한 수단으로 eBPF는 주목받고 있다. 이는 커널 모듈을 수정하지 않고도 커널의 기능을 확장할 수 있는 도구로, hook, kprobe, uprobe, tracepoint 등을 이용하여 커널 내 이벤트를 트레이싱하고 처리할 수 있다 [1]. 기존 기술은 커널 내 정보를 전부 복사하여 처리하는 반면, eBPF는 원하는 이벤트가 발생하였을 시 필요한 정보만 user space로 복사함으로써 트레이싱 시 발생하는 오버헤드를 최소화한다. 이를 통해 기존 기술에 비해 트레이싱 성능이 향상되며, 불필요한 자원 소모를 줄일 수 있다 [2]. 그러나 이벤트가 대용량으로 발생하였을 시 감지하는 과정에서 오버헤드가 발생할 수 있다. 이는 곧 프로그램을 관측하였을 때 성능 저하 문제로 이어질 수 있다. 따라서 본 논문에서는 오버헤드로 인한 성능 영향을 파악하고자 트레이싱 유무에 따른 system call 실행 시간을 측정하였다.

Ⅱ. 본론

1. 용어 및 개념

1) eBPF

eBPF(extended Berkeley Packet Filter)는 BPF의 커널 내의 패킷 필터링 기능을 범용 커널 가상 머신으로 확장 시킨 도구이다. eBPF 프로그램은 c와 같은 고급 언어로 작성되며 clang 컴파일러를 이용하여 ELF/object code로 컴파일된다. 컴파일된 파일은 프로세스가 실행되는 동안 verifier를 거쳐 안전하다고 판단 되었을 시 JIT를 거쳐 ELF eBPF loader를 통해 커널에 삽입되어 네트워크 트래픽, 시스템 이벤트를 실시간으로 처리하고 관리하는데 사용된다 [3].

2) tracepoint

tracepoint는 특정 이벤트가 발생하는 시점을 트레이싱하기 위한 커널 정적 계측 도구이다. 동작 과정은 다음과 같다: 1) 커널이 컴파일될 때 tracepoint 지점에 아무것도 하지 않는 명령어를 등록한다. 2) 이벤트가 발생시 콜백을 호출한다. 3) 콜백이 호출되면, 해당 이벤트가 발생하는 시점에서 정보를 수집한다. 4) 커널 바이너리로 컴파일하여 수집한 정보를 저장한다. 동적 계측 도구와 달리 지원하는 이벤트 수가 한정적이고, 커널 관리를 요구한다는 단점이 존재하지만, 안정적인 API를 가지고 있는 것이 장점이다 [4].

3) bpftrace

bpftrace는 시스템의 성능을 파악하거나 디버깅하는데 사용되는 eBPF용 고급 트레이싱 언어이다 [5]. 원 라이너 기능을 제공하여 사용자가 한 줄의 간단한 명령만으로 네트워크 트래픽, cpu 사용량, 디스크 I/O와 같은 다양한 시스템 이벤트를 감지할 수 있다. 또한 bpftrace는 사용자가 원하는 이벤트를 선택적으로 수집하고 필터링할 수 있는 기능을 제공한다. 수집된 정보를 그래프로 시각화하는 기능 또한 제공하여 데이터를 더욱 쉽게 이해할 수 있도록 도와준다 [6]. 이를 통해 사용자는 간단하면서도 효과적으로 시스템을 모니터링하고 디버깅할 수 있다.

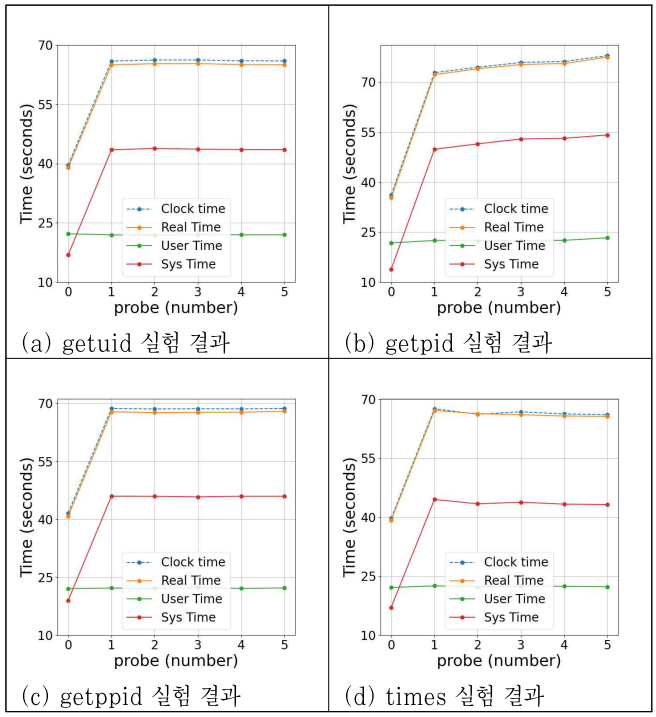
2. 성능 저하 측정 및 분석

[표 1]의 가상 환경에서 시스템 호출 트레이싱에 따른 성능 영향을 파악하기 위해 실험을 진행하였다. 각 system call을 5천만 번 호출하는 C 코드를 작성하여, 트레이싱 유무와 트레이싱 개수에 따른 실행 시간을 측정하였다. 이를 bpftrace의 tracepoint를 이용하여 트레이싱하는데 소요된 시간을 리눅스 time 명령어와 C의 clock()을 사용하여 측정하였고, getuid(), getpid(), getppid(), times() 함수에 대해 각 10번씩 반복 실행한 결과에 대한 평균을 [그림 1]의 그래프로 정리하였다.

[표 1]. 실험 환경

장비	사양
CPU	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz (core : 2개)
RAM	4GB
GCC	11.4.0
OS	Ubuntu Linux(64비트) 22.04.1

〔그림 1〕 실험 결과



(a) getuid 실험 결과 (b) getpid 실험 결과

(c) getppid 실험 결과 (d) times 실험 결과

clock()과 time 명령어로 측정 시 차이가 getuid에서 약 0.10초, getpid에서 약 0.40초, getppid에서 약 0.14초, times에서 약 0.43초 발생하였다.

〔그림 1〕 (a)의 getuid는 time 명령어로 측정한 결과를 기준으로 트레이싱 시 수행 시간이 약 1.68배, 시스템 수행 시간은 약 2.57배 증가하였던 반면 유저 수행 시간은 유일하게 약 0.99배 감소하였다. probe를 추가로 달아주었을 시에는 개수와 관계없이 전부 0.3초 미만의 근소한 차이를 보여주어 실행 시간에 큰 차이가 없었음을 확인하였다.

〔그림 1〕 (b)의 getpid는 time 명령어로 측정한 결과를 기준으로 트레이싱 시 수행 시간이 약 2.03배, 시스템 수행 시간은 약 3.59배 증가하여 가장 큰 증가 폭을 보여주었다. 유저 수행 시간 또한 약 1.03배로 소폭 증가하였다. probe를 추가로 달아주었을 시에는 전체 수행 시간이 최대 5초까지 시간이 지연되는 것을 알 수 있었다.

〔그림 1〕 (c)의 getppid는 time 명령어로 측정한 결과를 기준으로 트레이싱 시 수행 시간이 약 1.65배, 시스템 수행 시간은 약 2.41배 증가하여 가장 작은 증가 폭을 보여주었다. 유저 수행 시간또한 약 1.005배 증가로 가장 작은 변화 폭을 보여주었다. probe를 추가로 달아 주었을 때도 0.1 초 미만의 근소한 차이를 보여주었다.

〔그림 1〕 (d)의 times는 time 명령어로 측정한 결과를 기준으로 트레이싱 시 수행 시간이 약 1.83배, 시스템 수행 시간은 약 2.93배 증가하였다. 유저 수행 시간 또한 약 1.01배 증가하였다. 또한 probe를 추가로 달아주었을 시 2초가량 실행 시간이 감소하는 것을 보여주었다.

3. 결과 분석

clock()을 이용한 측정과 time 명령어를 이용하였을 시 평균 약 0.31초의 실행 결과 차이가 발생하였다. 이는 clock은 프로그램 내에서 cpu의 clock 수를 측정하고, time은 외부에서 프로그램 수행 시간을 측정하여 생긴 차이이다. 또한 4개의 system call 모두에서 트레이싱 시 적게는 1.65배에서 많게는 2.03배까지 수행 시간이 증가하는 것을 확인할 수 있었다. 특히 유저 수행 시간은 0.99배에서 1.03배로 거의 증가하거나 감소하지 않은반면, 시스템 수행 시간은 2.41배에서 3.59배까지 증가하는 것을 알 수 있었다. 따라서 system call 트레이싱 시 시스템 수행 시간에 많은 영향을 미치는 것을 알 수 있었다. probe 개수에 따른 오버헤드는 system call 트레이싱에 따른 오버헤드보다 크지 않음을 알 수 있었다.

Ⅲ. 결론

본 논문에서는 system call 트레이싱에 따른 실행 속도 저하에 대해 실험해 보았다. 시스템 수행 시간이 최대 3.59배까지 증가하는 것을 알 수 있었으며 이는 system call 트레이싱 과정에서 나타나는 성능 저하가 시스템 수행 시간을 지연시키는 주요 원인임을 시사한다. 이러한 결과는 시스템 콜 트레이싱의 오버헤드를 악용하여 시스템 수행 시간을 의도적으로 지연시키는 공격이 가능할 것이라 보인다. 특히 RTOS와 같이 정확한 시간 제약 내에서 작업을 처리하는 환경에서는 프로그램의 종료 시간을 예측하기 어려워질 수 있다. 이는 프로그램의 실행에 예상치 못한 영향을 미칠 수 있으며, 실시간 시스템에서는 이러한 예측 불가능성이 시스템의 응답성과 신뢰성에 부정적인 영향을 미칠 수 있다.

ACKNOWLEDGMENT

본 연구는 한국인터넷진흥원(KISA)-정보보안 특성화대학 지원사업 및 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2022R1C1C1006967).

참 고 문 헌

〔1〕 eBPF.io. 2023. eBPF Documentation. https://ebpf.io/what-is-ebpf

〔2〕 신용윤, et al. "클라우드 네이티브 환경에서 네트워킹및보안을 위한 eBPF 기술 동향." 〔ETRI〕 전자통신동향분석 37.5 (2022).

〔3〕 Vieira, Marcos AM, et al. "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications." ACM Computing Surveys (CSUR) 53.1 (2020): 1-36.

〔4〕 Brendan Gregg, 『BPF 성능 분석 도구』, 이호연 옮김,(서울: 도서출판인사이트,2021) 65-68.

〔5〕 Brendan Gregg, 『BPF 성능 분석 도구』, 이호연 옮김,(서울: 도서출판인사이트,2021) 153.

〔6〕 opensource.com. 2019. An introduction to bpftrace for Linux. https://opensource.com/article/19/8/introduction-bpftrace