(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2022-0088261
(43) 공개일자 2022년06월27일

(51) 국제특허분류(Int. Cl.)
H04L 49/00 (2022.01) *G06F 9/455* (2018.01)
H04L 9/40 (2022.01)
(52) CPC특허분류
H04L 49/103 (2022.05)
G06F 9/45545 (2013.01)
(21) 출원번호 10-2021-0030768
(22) 출원일자 2021년03월09일
심사청구일자 2021년03월09일
(30) 우선권주장
1020200178994 2020년12월18일 대한민국(KR)

(71) 출원인
인하대학교 산학협력단
인천광역시 미추홀구 인하로 100(용현동, 인하대학교)
(72) 발명자
박준석
서울특별시 강남구 삼성로 151, 10동 602호(대치동, 선경아파트)
민지홍
인천광역시 미추홀구 한나루로501번길 53, 405호(용현동)
(74) 대리인
양성보

전체 청구항 수 : 총 6 항

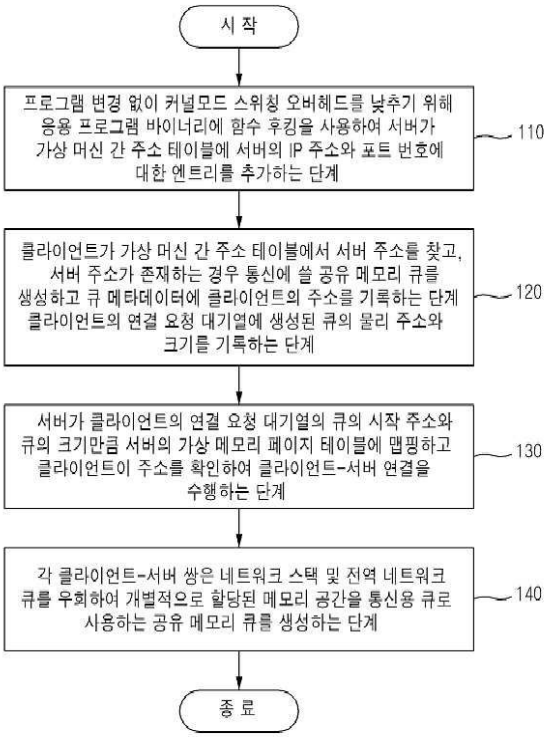
(54) 발명의 명칭 **동일 물리 머신에서 공유메모리를 이용한 가상 머신 간 고성능 통신 기법**

(57) 요 약

공유 메모리를 이용한 유저모드 폴링 기반 바이너리 호환 가상 머신 간 통신 방법 및 시스템이 제시된다. 본 발명에서 제안하는 공유 메모리를 이용한 유저모드 폴링 기반 바이너리 호환 가상 머신 간 통신 방법은 프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹(Function Hooking)을

(뒷면에 계속)

대 표 도 - 도1



사용하여 서버가 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하는 단계, 클라이언트가 가상 머신 간 주소 테이블에서 서버 주소를 찾고, 서버 주소가 존재하는 경우 통신에 쓸 공유 메모리 큐를 생성하고 큐 메타데이터에 클라이언트의 주소를 기록하는 단계, 서버가 클라이언트의 연결 요청 대기열에 생성된 큐의 물리 주소와 크기를 기록하는 단계, 서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트의 주소를 확인하여 클라이언트-서버 연결을 수행하는 단계 및 각 클라이언트-서버 쌍은 네트워크 스택 및 전역 네트워크 큐를 우회하여 개별적으로 할당된 메모리 공간을 통신용 큐로 사용하는 공유 메모리 큐를 생성하는 단계를 포함한다.

(52) CPC특허분류
G06F 9/45558 (2013.01)
H04L 69/16 (2022.05)
G06F 2009/45583 (2019.08)
G06F 2009/45595 (2019.08)

이 발명을 지원한 국가연구개발사업
과제고유번호 1711120431
과제번호 63663-1
부처명 과학기술정보통신부
과제관리(전문)기관명 한국연구재단
연구사업명 사회문제해결형기술개발(R&D)
연구과제명 [Ezbaro] 디지털 컴패니언 기반의 고령자 모니터링 및 대응소프트웨어 기술
기 여 율 1/2
과제수행기관명 인하대학교
연구기간 2020.07.20 ~ 2021.06.19
이 발명을 지원한 국가연구개발사업
과제고유번호 1711106050
과제번호 62244-1
부처명 과학기술정보통신부
과제관리(전문)기관명 한국연구재단
연구사업명 개인기초연구(과기정통부)(R&D)
연구과제명 [Ezbaro] 멀티쓰레드 및 SIMD를 지원하는 범용 프로세서환경에서 소프트에러로부터의 코드 보호를 위한 컴파일러 분석, 변환 기술 연구
기 여 율 1/2
과제수행기관명 인하대학교
연구기간 2020.03.01 ~ 2021.02.28

명 세 서

청구범위

청구항 1

프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹(Function Hooking)을 사용하여 서버가 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하는 단계;

클라이언트가 가상 머신 간 주소 테이블에서 서버 주소를 찾고, 서버 주소가 존재하는 경우 통신에 쓸 공유 메모리 큐를 생성하고 큐 메타데이터에 클라이언트의 주소를 기록하는 단계;

서버가 클라이언트의 연결 요청 대기열에 생성된 큐의 물리 주소와 크기를 기록하는 단계;

서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트의 주소를 확인하여 클라이언트-서버 연결을 수행하는 단계; 및

각 클라이언트-서버 쌍은 네트워크 스택 및 전역 네트워크 큐를 우회하여 개별적으로 할당된 메모리 공간을 통신용 큐로 사용하는 공유 메모리 큐를 생성하는 단계

를 포함하는 바이너리 호환 가상 머신 간 통신 방법.

청구항 2

제1항에 있어서,

프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹(Function Hooking)을 사용하여 서버가 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하는 단계는,

함수 후킹을 통해 바이너리 프로그램이 호출하는 함수를 후킹하여 다른 동적 라이브러리의 함수를 호출하여 유저모드에서 원하는 함수를 실행시키고, TCP/IP를 동적 라이브러리로 사용하는 모든 프로그램에 적용하고, 불필요한 기존 함수의 실행을 방지하기 위해 가상 머신 간 공유 메모리를 사용하는 Stub 함수로 TCP/IP 함수 호출을 링크하는

바이너리 호환 가상 머신 간 통신 방법.

청구항 3

제2항에 있어서,

서버는 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하고, 엔트리에 연결 요청 대기열을 생성하며 TCP/IP 함수 호출과 같이 이루어져 가상 네트워크를 통해 외부 네트워크에서 오는 연결 요청도 수신하는

바이너리 호환 가상 머신 간 통신 방법.

청구항 4

제1항에 있어서,

가상 머신 간 통신에서 발생하는 커널모드 스위칭 오버헤드를 제거하기 위해 유저모드 폴링 기법을 이용하여 커널모드 스위칭 없이, 공유 메모리에 존재하는 메타데이터 영역을 유저모드에서 주기적으로 참조하여 데이터가 도달하였는지 확인하는

바이너리 호환 가상 머신 간 통신 방법.

청구항 5

제1항에 있어서,

서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트의 주소를 확인하여 클라이언트-서버 연결을 수행하는 단계는,

서버 또는 클라이언트 중 어느 한쪽에서 연결을 종료할 경우 큐 메타 데이터에 연결이 종료되었음을 표기하고, 종료를 확인한 반대쪽이 가상 머신 간 주소 테이블 엔트리 및 공유 메모리 큐를 소거하는

바이너리 호환 가상 머신 간 통신 방법.

청구항 6

프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹(Function Hooking)을 사용하여 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하는 서버;

가상 머신 간 주소 테이블에서 서버 주소를 찾고, 서버 주소가 존재하는 경우 통신에 쓸 공유 메모리 큐를 생성하고 큐 메타데이터에 클라이언트의 주소를 기록하는 클라이언트

를 포함하고,

서버는,

클라이언트의 연결 요청 대기열에 생성된 큐의 물리 주소와 크기를 기록하고, 서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트의 주소를 확인하여 클라이언트-서버 연결을 수행하며,

각 클라이언트-서버 쌍은 네트워크 스택 및 전역 네트워크 큐를 우회하여 개별적으로 할당된 메모리 공간을 통신용 큐로 사용하는 공유 메모리 큐를 생성하는

바이너리 호환 가상 머신 간 통신 시스템.

발명의 설명

기 술 분 야

[0001] 본 발명은 동일 물리 머신에서 공유메모리를 이용한 가상 머신 간 고성능 통신 방법 및 시스템에 관한 것이다.

배 경 기 술

[0002] 현대의 컴퓨팅 환경에서 물리적으로 분산된 시스템에서 수행되고 있는 응용 프로그램 프로세스 간의 정보 교환을 위해 네트워크 통신이 널리 사용되고 있다. 원격지에 정보를 저장하고 불러오게 해주는 파일 전송 프로토콜(File Transfer Protocol; FTP), 처리를 요청할 수 있는 원격 프로시저 호출(Remote Procedure Call; RPC) 등이 대표적인 예다. 이러한 네트워크 통신을 위해서 TCP/IP 프로토콜이 널리 사용된다. 더 나아가 가상 머신 간의 통신에서는 같은 물리 머신 내에서도 네트 워크 통신이 이루어지기도 한다. 가상 머신은 메모리 공간 및 프로세스 문맥 등이 호스트 및 다른 가상 머신으로부터 분리되어 있다. 이때 서로 분리된 호스트 및 가상 머신 간 데이터 교환에서 보편적으로 네트워크 프로토콜을 사용한다. 이를 위해 하이퍼바이저는 가상 네트워크를 제공하며 결과적으로 기존 네트워크 소프트웨어와의 호환성을 유지할 수 있다. 그러나 물리적으로 분리된 환경을 고려한 기존의 TCP/IP 기반 네트워킹은 가상 머신 간 통신에 있어 일정 수준의 비효율성을 피할 수 없다. 이를 해결하기 위해 효율적인 가상 머신 간 정보 교환을 위한 프로토콜 및 시스템 소프트웨어들이 등장하였다. 그러나 응용 프로그램의 재프로그램 및 재컴파일 이 필요하거나, 기존 네트워크 구조를 일부 유지함으로 써 커널모드 스위칭으로 오버헤드를 줄이기 어려운 문제가 여전히 남아있다[1].

[0003] 하이퍼바이저는 가상 네트워크 환경을 제공하여 외부 네트워크뿐만 아니라 호스트 및 타 가상 머신으로의 연결을 지원한다. 가상 머신은 가상 네트워크 인터페이스(Virtual Network Interface Controller; vNIC)를 통해 호스트 및 타 가상 머신과 통신할 수 있다. 이때 반가상화(Paravirtualization) 방식으로 성능을 개선하기도 한다[2].

[0004] 그러나 물리적 네트워크를 모방하여 기존 네트워크 스택을 사용함에 따라, 계층 간 메모리 복사 및 메시지 생성, 라우팅, 네트워크 큐로 인한 오버헤드가 존재한다[3,4]. 또한, 네트워크 큐에 읽기나 쓰기가 발생할 때 커널모드 스위칭이 자주 발생함에 따라 오버헤드가 발생한다. 특히 최근 Meltdown과 Spectre와 같은 부채널 공

격들에 대한 대응을 위한 오버헤드가 더욱 증가하였다[5-7].

[0005] 가상 머신 간 통신을 위한 다른 방식으로는 가상 머신 간 공유 메모리(Inter-VM Shared Memory)가 있다. Linux의 가상 PCIe(Peripheral Component Interconnect Express)인 IVSHMEM은 호스트에서 생성한 공유 메모리 영역을 가상 하드웨어의 I/O 메모리 영역으로 제공한다[8]. 가상 머신 내 개별 프로세스의 페이지 테이블에 해당 영역을 맵핑하여 직접 접근이 가능하다. 그러나 이 방식은 API의 변경이 필요하며, 이미 작성된 기존 TCP/IP 기반 응용 프로그램 및 라이브러리들을 재프로그램해야 하는 단점이 있다.

발명의 내용

해결하려는 과제

[0006] 본 발명이 이루고자 하는 기술적 과제는 가상 머신 간 공유 메모리 기반 바이너리 호환 통신 방법들의 오버헤드를 제거하기 위해 네트워크 스택을 TCP/IP 라이브러리의 함수 후킹으로 우회하고, 연결 별 전송 큐 및 유저모드 폴링 기법의 도입으로 커널모드 스위칭 오버헤드를 제거하는 방법 및 시스템을 제공하는데 있다.

과제의 해결 수단

[0007] 일 측면에 있어서, 본 발명에서 제안하는 공유 메모리를 이용한 유저모드 폴링 기반 바이너리 호환 가상 머신 간 통신 방법은 프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹(Function Hooking)을 사용하여 서버가 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하는 단계, 클라이언트가 가상 머신 간 주소 테이블에서 서버 주소를 찾고, 서버 주소가 존재하는 경우 통신에 쓸 공유 메모리 큐를 생성하고 큐 메타데이터에 클라이언트의 주소를 기록하는 단계, 서버가 클라이언트의 연결 요청 대기열에 생성된 큐의 물리 주소와 크기를 기록하는 단계, 서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트의 주소를 확인하여 클라이언트-서버 연결을 수행하는 단계 및 각 클라이언트-서버 쌍은 네트워크 스택 및 전역 네트워크 큐를 우회하여 개별적으로 할당된 메모리 공간을 통신용 큐로 사용하는 공유 메모리 큐를 생성하는 단계를 포함한다.

[0008] 프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹(Function Hooking)을 사용하여 서버가 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하는 단계는 함수 후킹을 통해 바이너리 프로그램이 호출하는 함수를 후킹하여 다른 동적 라이브러리의 함수를 호출하여 유저모드에서 원하는 함수를 실행시키고, TCP/IP를 동적 라이브러리로 사용하는 모든 프로그램에 적용하고, 불필요한 기존 함수의 실행을 방지하기 위해 가상 머신 간 공유 메모리를 사용하는 Stub 함수로 TCP/IP 함수 호출을 링크한다.

[0009] 본 발명의 실시예에 따른 서버는 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하고, 엔트리에 연결 요청 대기열을 생성하며 TCP/IP 함수 호출과 같이 이루어져 가상 네트워크를 통해 외부 네트워크에서 오는 연결 요청도 수신할 수 있다.

[0010] 본 발명의 실시예에 따르면, 가상 머신 간 통신에서 발생하는 커널모드 스위칭 오버헤드를 제거하기 위해 유저모드 폴링 기법을 이용하여 커널모드 스위칭 없이, 공유 메모리에 존재하는 메타데이터 영역을 유저모드에서 주기적으로 참조하여 데이터가 도달하였는지 확인할 수 있다.

[0011] 서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트의 주소를 확인하여 클라이언트-서버 연결을 수행하는 단계는 서버 또는 클라이언트 중 어느 한쪽에서 연결을 종료할 경우 큐 메타 데이터에 연결이 종료되었음을 표기하고, 종료를 확인한 반대쪽이 가상 머신 간 주소 테이블 엔트리 및 공유 메모리 큐를 소거한다.

[0012] 또 다른 일 측면에 있어서, 본 발명에서 제안하는 공유 메모리를 이용한 유저모드 폴링 기반 바이너리 호환 가상 머신 간 통신 시스템은 프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹(Function Hooking)을 사용하여 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하는 서버, 가상 머신 간 주소 테이블에서 서버 주소를 찾고, 서버 주소가 존재하는 경우 통신에 쓸 공유 메모리 큐를 생성하고 큐 메타데이터에 클라이언트의 주소를 기록하는 클라이언트를 포함한다.

[0013] 본 발명의 실시예에 따른 서버는 클라이언트의 연결 요청 대기열에 생성된 큐의 물리 주소와 크기를 기록하고, 서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에

맵핑하고 클라이언트의 주소를 확인하여 클라이언트-서버 연결을 수행하며, 각 클라이언트-서버 쌍은 네트워크 스택 및 전역 네트워크 큐를 우회하여 개별적으로 할당된 메모리 공간을 통신용 큐로 사용하는 공유 메모리 큐를 생성한다.

발명의 효과

[0014] 본 발명의 실시예들에 따르면 가상 머신 간 공유 메모리 기반 바이너리 호환 통신 방법들의 오버헤드를 제거하기 위해 네트워크 스택을 TCP/IP 라이브러리의 함수 후킹으로 우회하고, 연결 별 전송 큐 및 유저모드 폴링 기법의 도입으로 커널모드 스위칭 오버헤드를 제거할 수 있다. 본 발명의 실시예들에 따르면 기존 가상 네트워크 방식에 비해 평균 레이턴시는 감소시키고, 평균 처리량은 증가시킬 수 있다.

도면의 간단한 설명

[0015] 도 1은 본 발명의 일 실시예에 따른 공유 메모리를 이용한 유저모드 폴링 기반 바이너리 호환 가상 머신 간 통신 방법을 설명하기 위한 흐름도이다.

도 2는 본 발명의 일 실시예에 따른 SocketIV를 이용한 통신 과정을 설명하기 위한 도면이다.

도 3은 본 발명의 일 실시예에 따른 가상 머신 간 주소 테이블 및 공유 메모리 큐를 설명하기 위한 도면이다.

도 4는 본 발명의 일 실시예에 따른 최소, 최대, 평균 레이턴시를 나타내는 그래프이다.

도 5는 본 발명의 일 실시예에 따른 평균 처리 시간을 나타내는 그래프이다.

발명을 실시하기 위한 구체적인 내용

[0016] 가상 머신의 호스트 및 타 가상 머신 간 데이터 교환에서 호환성을 위해 보편적으로 TCP/IP 프로토콜을 사용하나 성능상의 비효율성이 발생한다. 가상 머신 간 공유 메모리를 이용하여 통신할 경우 더 효율적인 데이터 전송이 가능하지만, 기존 TCP/IP 기반 프로그램을 재프로그램 혹은 재컴파일해야 하는 단점이 있다. 이를 해결하기 위한 가상 머신 간 공유 메모리 기반 바이너리 호환 통신 방법들이 연구되었으나 제거되지 못한 오버헤드가 존재한다.

[0017] 가상 머신 간 연결에서 TCP/IP 기반 네트워킹의 오버헤드를 줄이기 위한 연구들이 다수 진행되었다. 최근 효율성을 높이기 위해 같은 CPU/코어를 공유하는 가상 머신의 수가 늘어나면서 가상 머신 간의 스케줄링 오버 헤드 역시 점차 증가하였다. 증가한 스케줄링 오버헤드는 패킷 전송에도 영향을 미쳐 전체적인 네트워크 성능과 해당 데이터에 의존성이 있는 연산 처리량도 떨어뜨린다. vSnoop과 vFlood는 스케줄링에서 우선순위가 낮은 가상 머신들 대신 호스트 도메인에서 패킷 처리를 대신 하는 방식으로 성능을 개선하였다[9,10]. 하지만 이러한 종래기술들은 가상 머신 간 연결에서 기존 TCP/IP 스택을 그대로 유지한다는 점에서 본 발명과는 차이가 있다.

[0018] XenLoop은 가상 머신 간 통신 시 네트워크 드라이버 레벨에서 TCP/IP 패킷을 가로채고, 공유 메모리를 사용한 데이터 전송 성능 향상을 이루는 방안을 제시하였다 [11]. 바이너리 호환성을 제시하고 공유 메모리를 사용 한다는 점에서 본 발명과 일부 유사하다. 하지만 본 발명의 경우, 기존 네트워크 스택을 전혀 사용하지 않음으로써 그로 인한 오버헤드를 없애 더욱 큰 성능 향상을 도모한다는 점에서 차이가 있다.

[0019] XenVMC, XWAY, Socket-Outsourcing의 경우는 커널에서 시스템 콜을 가로채는 방식을 사용하였다 [12-14]. 이러한 종래기술도 바이너리 호환성을 유지하면서 네트워크 스택을 우회하였다는 점은 본 발명과 유사하다. 그러나 읽기 및 쓰기 과정에서 빈번하게 발생하는 시스템 콜로 인한 커널모드 스위칭 및 커널 영역의 전역 큐의 공유로 인한 오버헤드가 남아있다.

[0020] 마지막으로 Fido[15]는 전송자 가상 머신의 커널 메모리 영역을 수신자 가상 머신에게 읽기 전용으로 노출하는 방법을 사용한다. 이 방식은 데이터 복사 비용을 줄여준다. 그러나 네트워크 스택 자체를 우회하지는 않아, 전송 데이터가 네트워크 스택을 지난 다음 커널 내 전역 네트워크 큐에 들어간 이후 공유가 되므로 일정 부분의 오버헤드가 남아있다. 또한, 이 방식은 가상 머신 간의 분리를 해칠 수 있는 보안상의 문제점이 존재 한다. 이하, 본 발명의 실시 예를 첨부된 도면을 참조하여 상세하게 설명한다.

[0022] 본 발명에서는 기존 네트워크 스택 및 커널모드 스위칭으로 인해 발생하는 오버헤드를 제거하기 위한 바이너리 호환 가상 머신 간 통신 기법을 제안한다. 기존 네트워크 스택을 TCP/IP 라이브러리의 함수 후킹으로 우회하여

응용 프로그램의 변경이나 재컴파일이 필요 없으며, 유저모드 폴링을 통해 커널모드 스위칭 오버헤드를 제거할 수 있다.

[0024] 도 1은 본 발명의 일 실시예에 따른 공유 메모리를 이용한 유저모드 폴링 기반 바이너리 호환 가상 머신 간 통신 방법을 설명하기 위한 흐름도이다.

[0025] 제안하는 공유 메모리를 이용한 유저모드 폴링 기반 바이너리 호환 가상 머신 간 통신 방법은 프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹(Function Hooking)을 사용하여 서버가 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하는 단계(110), 클라이언트가 가상 머신 간 주소 테이블에서 서버 주소를 찾고, 서버 주소가 존재하는 경우 통신에 쓸 공유 메모리 큐를 생성하고 큐 메타데이터에 클라이언트의 주소를 기록하는 단계(120), 서버가 클라이언트의 연결 요청 대기열에 생성된 큐의 물리 주소와 크기를 기록하는 단계(130), 서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트의 주소를 확인하여 클라이언트-서버 연결을 수행하는 단계(140) 및 각 클라이언트-서버 쌍은 네트워크 스택 및 전역 네트워크 큐를 우회하여 개별적으로 할당된 메모리 공간을 통신용 큐로 사용하는 공유 메모리 큐를 생성하는 단계(150)를 포함한다.

[0026] 단계(110)에서, 프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹(Function Hooking)을 사용하여 서버가 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가한다.

[0027] 함수 후킹을 통해 바이너리 프로그램이 호출하는 함수를 후킹하여 다른 동적 라이브러리의 함수를 호출하여 유저모드에서 원하는 함수를 실행시키고, TCP/IP를 동적 라이브러리로 사용하는 모든 프로그램에 적용한다. 본 발명의 실시예에 따르면, 불필요한 기존 함수의 실행을 방지하기 위해 가상 머신 간 공유 메모리를 사용하는 Stub 함수로 TCP/IP 함수 호출을 링크한다.

[0028] 서버는 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하고, 엔트리에 연결 요청 대기열을 생성한다. 이러한 과정은 TCP/IP 함수 호출과 같이 이루어져 가상 네트워크를 통해 외부 네트워크에서 오는 연결 요청도 수신할 수 있다.

[0029] 단계(120)에서, 클라이언트가 가상 머신 간 주소 테이블에서 서버 주소를 찾고, 서버 주소가 존재하는 경우 통신에 쓸 공유 메모리 큐를 생성하고 큐 메타데이터에 클라이언트의 주소를 기록한다.

[0030] 단계(130)에서, 서버가 클라이언트의 연결 요청 대기열에 생성된 큐의 물리 주소와 크기를 기록한다.

[0031] 단계(140)에서, 서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트의 주소를 확인하여 클라이언트-서버 연결을 수행한다.

[0032] 서버 또는 클라이언트 중 어느 한쪽에서 연결을 종료할 경우 큐 메타 데이터에 연결이 종료되었음을 표기하고, 종료를 확인한 반대쪽이 가상 머신 간 주소 테이블 엔트리 및 공유 메모리 큐의 소거를 한다.

[0033] 단계(150)에서, 각 클라이언트-서버 쌍은 네트워크 스택 및 전역 네트워크 큐를 우회하여 개별적으로 할당된 메모리 공간을 통신용 큐로 사용하는 공유 메모리 큐를 생성한다.

[0034] 본 발명의 실시예에 따르면, 가상 머신 간 통신에서 발생하는 커널모드 스위칭 오버헤드를 제거하기 위해 유저모드 폴링 기법을 이용하여 커널모드 스위칭 없이, 공유 메모리에 존재하는 메타데이터 영역을 유저모드에서 주기적으로 참조하여 데이터가 도달하였는지 확인할 수 있다. 도 2를 참조하여 본 발명의 일 실시예에 따른 SocketIV를 이용한 통신 과정을 더욱 상세히 설명한다.

[0036] 도 2는 본 발명의 일 실시예에 따른 SocketIV를 이용한 통신 과정을 설명하기 위한 도면이다.

[0037] 본 발명에서는 TCP/IP 기반 응용 프로그램 및 네트워크 스택의 변경 없이 가상 머신 간의 통신 성능을 개선하기 위한 방법 및 시스템을 제안한다. 본 발명의 일 실시예에 따르면, 프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함 수 후킹(Function Hooking)을 사용하였다. 함수 후킹이 란 바이너리 프로그램이 호출하는 함수를 가로채어 다른 동적 라이브러리의 함수를 호출하는 기술이다. 이 방법을 사용하면 유저모드에서 바로 원하는 함수를 실행시킬 수 있어 커널모드 스위칭 비용을 절감할 수 있다.

[0038] 본 발명의 실시예에서는 링커 ld의 LD_PRELOAD 기능을 사용하여, TCP/IP 함수 호출을 가상 머신 간 공유 메모리를 사용하는 Stub 함수로 링크하였다[18]. 이와 같은 방법을 통해, TCP/IP를 동적 라이브러리로 사용하는 모든 프로그램에 적용이 가능하고, 불필요한 기존 함수의 실행을 방지할 수 있다. 함수 후킹용 라이브러리, SocketIV(Socket for InterVM)는 공유 메모리 백엔드로 IVSHMEM을 사용한다.

[0039] 도 2는 이러한 SocketIV 라이브러리를 적용하였을 때의 통신 과정을 나타내는 도면이다. 먼저 서버는 가상 머신 간 주소 테이블(Inter-VM Address Table; IVAT)에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가한다. 그리고 자신의 엔트리에 연결 요청 대기열을 생성한다. 이 과정은 연관된 기존 TCP/IP 함수 호출과 같이 이루어져 가상 네트워크를 통해 외부 네트워크에서 오는 연결 요청도 받을 수 있다. 클라이언트는 테이블에서 서버 주소를 찾고, 존재하면 통신에 쓸 공유 메모리 큐를 생성하고 큐 메타데이터에 자신의 주소를 기록한다. 이후 연결 요청 대기열에 생성된 큐의 물리 주소와 크기를 기록하며, 이때 상호 배제를 위하여 CPU가 지원하는 Atomic Swap 명령어를 이용한다[19]. 이후 서버는 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 자신의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트의 주소를 확인한다.

[0041] 도 3은 본 발명의 일 실시예에 따른 가상 머신 간 주소 테이블 및 공유 메모리 큐를 설명하기 위한 도면이다.

[0042] 도 3을 참조하면, 앞에서 설명된 과정을 거쳐 구성된 가상 머신 간 주소 테이블 및 공유 메모리 큐 구조를 나타내었다. 각 클라이언트-서버 쌍은 개별적으로 할당된 메모리 공간을 통신용 큐로 사용한다. 클라이언트나 서버는 공유 메모리 큐 영역에 데이터를 기록하면 자신의 쓰기 포인터를 업데이트한다. 이러한 방식을 통해 기존 네트워크 스택 및 전역 네트워크 큐를 우회하여 고속의 데이터 전송이 가 능하다. 이후 어느 한쪽에서 연결을 종료할 경우 큐 메타 데이터에 연결이 종료되었음을 표기하고, 이를 확인한 반대쪽이 가상 머신 간 주소 테이블 엔트리 및 공유 메모리 큐의 소거를 한다. 일반적인 네트워크 소켓 통신에서는 서로의 전송이 끝났을 때를 감지하기 위한 기법이 필요하다. 이를 인터럽트로 처리할 경우엔, 인터럽트 핸들러 호출로 인한 커널모드 스위칭 오버헤드가 발생한다. 가상 머신 간 통신은 실제 분산 네트워크 환경 대비 전송속도가 매우 빠르므로 커널모드 스위칭이 더욱 자주 발생하게 된다. 본 발명에서는 이러한 문제를 해결하기 위해 유저모드 폴링 기법을 도입하였다. 유저모드 폴링 기법을 통해 커널모드 스위칭 없이, 공유 메모리에 존재하는 메타데이터 영역을 유저모드에서 주기적으로 참조하여 데이터가 도달하였는지 확인할 수 있다.

[0043] 종래기술과 비교하여 본 발명은 기존의 자체 벤치마크가 아닌 표준적인 TCP/IP 벤치마크 프로그램으로 사용되는 netperf[16]를 통해 실험하였다. 그리고 이에 대한 성능 향상 측정을 기존처럼 대역폭뿐만이 아닌 레이턴시를 포함하여 보였다. 마지막으로, 해당 프로그램을 기존 방식과 본 발명의 방식으로 수행하였을 때의 콜 트레이스를 uftrace[17]로 분석하였다.

[0044] 본 발명의 일 실시예에 따른 실험은 다음과 같은 환경에서 진행하였으며, 네트워크 대역폭 실험에 사용하는 소프트웨어인 netperf 2.7.0 버전을 사용하여 대역폭 및 레이턴시를 측정하였다. netperf 는 전송량을 2.00GB로 하고, 나머지 설정은 2.7.0 버전 기준 기본 설정을 사용하였다.

[0045] CPU: Intel® CoreTM i7-7700@3.60GHz

[0046] 메모리: 32GB(8GB * 4EA) DDR4 2400MHz

[0047] 운영체제(호스트): Ubuntu 19.10 64-bit

[0048] 운영체제(가상 머신): Ubuntu 18.04.4 LTS 64-bit

[0049] 가상 머신당 vCPU 개수: 4(Pinning: 2C/4T)

[0050] 가상 머신당 메모리 용량: 8GB(페이지 크기: 1GB)

[0052] 도 4는 본 발명의 일 실시예에 따른 최소, 최대, 평균 레이턴시를 나타내는 그래프이다.

[0053] 도 5는 본 발명의 일 실시예에 따른 평균 처리 시간을 나타내는 그래프이다.

[0054] 도4를 참조하면, netperf를 기존 방식과 본 발명에서 제안하는 방식으로 각각 100회 씩 수행하였을 때의 최소, 최대, 그리고 평균 레이턴시를 나타내었다. 또한, 도 5는 위와 같은 방식으로 수 행하였을 때의 평균 처리 시간이다. 평균 처리량으로 환산할 경우 기존 방식은 2.98GB/s, 본 발명에서 제안하는 방식은 9.59GB/s이다. 평균

레이턴시의 경우 96.06% 감소하였으며, 평균 처리량은 222.24%가 증가하였다. uftrace 0.8.2 버전을 이용하여 netperf 실행 시, 기존 가상 네트워크 방식과 본 발명에서 제안하는 방식의 유저모드 및 커널모드 콜 트레이스를 비교하였다. 비교 대상은 실질적으로 netperf 안에서 읽기와 쓰기 작업을 담당하는 최상위 함수인 recv_data()와 send_data()로 하였다.

[0055] 표 1 은 가상 네트워크 방식으로 구동하였을 때의 콜 트레이스 결과이며, 표 2는 본 발명에서 제안하는 방식으로 구동하였을 때의 콜 트레이스 결과이다.

[0056] <표 1>

[Server] recv_data() (Elapsed Time: 18.455s)		**[Client]** send_data() (Elapsed Time: 14.268s)	
Function	%	Function	%
sk_wait_data()	14.19	tcp_sendmsg_locked()	34.48
tcp_cleanup_rbuf()	13.54	release_sock()	20.63
skb_copy_datagram_iter()	9.59	lock_sock_nested()	1.09
__kfree_skb()	6.43	security_socket_sendmsg()	1.06
release_sock()	4.70	sockfd_lookup_light()	0.48

[0057]

[0058] <표 2>

[Server] recv_data() (Elapsed Time: 544.406ms)		**[Client]** send_data() (Elapsed Time: 270.827ms)	
Function	%	Function	%
usleep()	78.05	memcpy()	70.58
memcpy()	18.87	do_async_page_fault()	5.67
do_async_page_fault()	0.92	do_syscall_64()	1.97
do_syscall_64()	0.44	smp_apic_time_interrupt()	0.37
smp_apic_time_interrupt()	0.02	smp_irq_work_interrupt()	0.04

[0059]

[0060] 각 표는 소요 시간을 기준으로 5개 함수를 표시하였다. 서버와 클라이언트의 실행 시작 시간은 0.25s의 오차로 동기화 되었다. 표 1과 표 2를 비교해보면, 서버의 경우 sk_wait_data() 와 usleep()은 데이터 전송을 기다리는 부분이다. 또한, 기존 방식의 경우 tcp_cleanup_rbuf(), _kfree_skb(), release_ sock() 같은 커널 내 전역 메모리 큐의 불필요한 동적 조절이 계속 발생하는 것을 확인할 수 있다. 본 발명에서 제안하는 방식의 경우, usleep()을 제외하고는 메모리 복사가 대 부분을 차지하고 있는데, usleep()을 오래하는 부분은 정적인 폴링 주기의 영향을 받은 것으로 보인다. 추후 처리량의 변화에 따른 가변 폴링이나 선택적 인터럽트를 도입하는 것으로 완화할 수 있을 것으로 예측된다. 또한, 클라이언트에서는 기존 가상 네트워크 방식의 경우 tcp_sendmsg_locked() 및 lock_sock_nested() 등 전역 메모리 큐에 쓰기 위하여 락을 사용한다. 본 발명에서 제안하는 방식은 연결 별 별도의 큐를 사용하므로 이 부분의 오버헤드가 제거된 상태이다. 페이지 폴트를 제외한 시간 대부분을 memcpy()에 사용되고, 이는 대역폭의 향상으로 이어진다

[0061] 실험 결과에서 나타난 성능향상을 고려하면 본 발명에서 제안하는 방식의 장점은 매우 명확하다. 하지만 본 발명에서 제안하는 방식은 기존의 가상 머신에서의 TCP/IP 통신 방식에 비해 크게 두 가지의 단점이 있다. 첫째는 개별 프로그램들이 백엔드로 사용하는 IVSHMEM의 I/O 메모리 영역에 접근하기 위해 관리자 권한이 있어야 하는 점이다. 따라서 해당 프로세스가 메모리를 임의로 접근할 수 있는 문제점도 존재한다. 그리고 둘째는 IVSHMEM의 I/O 메모리 공간으로 큰 공간을 정적으로 할당해줘 메모리 공간의 낭비가 발생하는 점이다. 클라이언트와 서버 간의 연결이 최초로 이루어질 때, 커널에서 공유 가능한 버퍼 공간을 각 연결에 독립적으로 할당해주는 방식으로 보안 문제점을 해결할 수 있다. 이러한 문제를 해결하기 위해 리눅스 커널 내부에서 상술된 역할을 담당하는 커널 함수를 작성하고 이에 접근하기 위해 연결당 1회 사용되는 시스템 콜을 추가할 수 있다. 현재는 IVSHMEM 백엔드의 I/O 메모리 영역 크기 설정 및 사용 가상 머신 수를 동적으로 조정하는 것이 불가능하다. 이로 인해 IVSHMEM을 위한 큰 메모리 공간을 정적으로 할당해야 하는 문제가 있다. 이러한 문제를 해결하기 위해 IVSHMEM의 I/O 메모리 영역을 동적으로 조절할 수 있게 하고, 더 나아가 PCIe Hot-Plug 기능을 구현하여 장치를 동적으

로 추가 및 제거할 수 있다.

[0062] 앞서 설명된 바와 같이, 본 발명은 현재의 가상 머신 간 공유 메모리 기반 바이너리 호환 통신 방법들의 오버헤드를 제거하기 위한 기법을 제안한다. 제안하는 기법에서는 기존 네트워크 스택을 TCP/IP 라이브러리의 함수 후킹으로 우회하였으며, 연결 별 전송 큐 및 유저모드 폴링 기법의 도입으로 커널모드 스위칭 오버헤드를 제거하였다. 결과에 따르면, 제안하는 기법을 사용할 경우 기존 가상 네트워크 방식 대비 평균 레이턴시는 96.06% 감소, 평균 처리량은 222.24%가 증가한 것을 확인 할 수 있었다.

[0064] 이상에서 설명된 장치는 하드웨어 구성요소, 소프트웨어 구성요소, 및/또는 하드웨어 구성요소 및 소프트웨어 구성요소의 조합으로 구현될 수 있다. 예를 들어, 실시예들에서 설명된 장치 및 구성요소는, 예를 들어, 프로세서, 콘트롤러, ALU(arithmetic logic unit), 디지털 신호 프로세서(digital signal processor), 마이크로컴퓨터, FPA(field programmable array), PLU(programmable logic unit), 마이크로프로세서, 또는 명령(instruction)을 실행하고 응답할 수 있는 다른 어떠한 장치와 같이, 하나 이상의 범용 컴퓨터 또는 특수 목적 컴퓨터를 이용하여 구현될 수 있다. 처리 장치는 운영 체제(OS) 및 상기 운영 체제 상에서 수행되는 하나 이상의 소프트웨어 애플리케이션을 수행할 수 있다. 또한, 처리 장치는 소프트웨어의 실행에 응답하여, 데이터를 접근, 저장, 조작, 처리 및 생성할 수도 있다. 이해의 편의를 위하여, 처리 장치는 하나가 사용되는 것으로 설명된 경우도 있지만, 해당 기술분야에서 통상의 지식을 가진 자는, 처리 장치가 복수 개의 처리 요소(processing element) 및/또는 복수 유형의 처리 요소를 포함할 수 있음을 알 수 있다. 예를 들어, 처리 장치는 복수 개의 프로세서 또는 하나의 프로세서 및 하나의 콘트롤러를 포함할 수 있다. 또한, 병렬 프로세서(parallel processor)와 같은, 다른 처리 구성(processing configuration)도 가능하다.

[0065] 소프트웨어는 컴퓨터 프로그램(computer program), 코드(code), 명령(instruction), 또는 이들 중 하나 이상의 조합을 포함할 수 있으며, 원하는 대로 동작하도록 처리 장치를 구성하거나 독립적으로 또는 결합적으로(collectively) 처리 장치를 명령할 수 있다. 소프트웨어 및/또는 데이터는, 처리 장치에 의하여 해석되거나 처리 장치에 명령 또는 데이터를 제공하기 위하여, 어떤 유형의 기계, 구성요소(component), 물리적 장치, 가상 장치(virtual equipment), 컴퓨터 저장 매체 또는 장치에 구체화(embody)될 수 있다. 소프트웨어는 네트워크로 연결된 컴퓨터 시스템 상에 분산되어서, 분산된 방법으로 저장되거나 실행될 수도 있다. 소프트웨어 및 데이터는 하나 이상의 컴퓨터 판독 가능 기록 매체에 저장될 수 있다.

[0066] 실시예에 따른 방법은 다양한 컴퓨터 수단을 통하여 수행될 수 있는 프로그램 명령 형태로 구현되어 컴퓨터 판독 가능 매체에 기록될 수 있다. 상기 컴퓨터 판독 가능 매체는 프로그램 명령, 데이터 파일, 데이터 구조 등을 단독으로 또는 조합하여 포함할 수 있다. 상기 매체에 기록되는 프로그램 명령은 실시예를 위하여 특별히 설계되고 구성된 것들이거나 컴퓨터 소프트웨어 당업자에게 공지되어 사용 가능한 것일 수도 있다. 컴퓨터 판독 가능 기록 매체의 예에는 하드 디스크, 플로피 디스크 및 자기 테이프와 같은 자기 매체(magnetic media), CD-ROM, DVD와 같은 광기록 매체(optical media), 플롭티컬 디스크(floptical disk)와 같은 자기-광 매체(magneto-optical media), 및 롬(ROM), 램(RAM), 플래시 메모리 등과 같은 프로그램 명령을 저장하고 수행하도록 특별히 구성된 하드웨어 장치가 포함된다. 프로그램 명령의 예에는 컴파일러에 의해 만들어지는 것과 같은 기계어 코드뿐만 아니라 인터프리터 등을 사용해서 컴퓨터에 의해서 실행될 수 있는 고급 언어 코드를 포함한다.

[0067] 이상과 같이 실시예들이 비록 한정된 실시예와 도면에 의해 설명되었으나, 해당 기술분야에서 통상의 지식을 가진 자라면 상기의 기재로부터 다양한 수정 및 변형이 가능하다. 예를 들어, 설명된 기술들이 설명된 방법과 다른 순서로 수행되거나, 및/또는 설명된 시스템, 구조, 장치, 회로 등의 구성요소들이 설명된 방법과 다른 형태로 결합 또는 조합되거나, 다른 구성요소 또는 균등물에 의하여 대치되거나 치환되더라도 적절한 결과가 달성될 수 있다.

[0068] 그러므로, 다른 구현들, 다른 실시예들 및 특허청구범위와 균등한 것들도 후술하는 특허청구범위의 범위에 속한다.

[0070] <참고자료>

[0071] [1] Y. Ren et al., "Shared-Memory Optimizations for Inter-Virtual-Machine Communication," ACM

Computing Surveys, Vol. 48, Issue 4, pp. 1-42, Feb. 2016.

[0072] [2] R. Russell, "virtio: Towards a De-Facto Standard for Virtual I/O Devices," ACM SIGOPS Operating Systems Review, Vol. 42, Issue 5, pp. 95-103, Jul. 2008.

[0073] [3] E. Martin. (2019, Sep. 12). Deep Dive into Virtio- Net working and vhost-net [Online]. Available: http:// www.redhat.com (downloaded 2020. Jun. 16)

[0074] [4] M. Jones. (2010, Jan. 29) Virtio: An I/O Virtualization Framework for Linux [Online]. Available: http://deve loper.ibm.com (downloaded 2020, Mar. 1)

[0075] [5] M. Lipp et al., "Meltdown: Reading Kernel Memory from User Space," Proc. of 27th USENIX Conference on Security Symposium, pp. 973-990, Aug. 2018.

[0076] [6] P. Kocher et al., "Spectre Attacks: Exploiting Speculative Execution," 40th IEEE Symposium on Security and Privacy, pp. 1-19, May, 2019.

[0077] [7] M. Larabel. (2019, Mar. 11) Spectre/Meltdown Perform ance Impact Across Eight Linux Distributions [Online]. Available: http://www.phoronix.com (downloaded 2020. Jun. 16)

[0078] [8] C. Macdonell, "Shared-Memory Optimizations for Virtual Machines," Doctoral Thesis, University of Alberta, 2011.

[0079] [9] A. Kangarlou et al., "vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload," Proc. of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11, Nov. 2010.

[0080] [10] S. Gamage et al., "Opportunistic flooding to improve TCP transmit performance in virtualized clouds," Proc. of 2nd ACM Symposium on Cloud Computing Article 24, pp. 1-14, Oct. 2011.

[0081] [11] W. Jian et al., "XenLoop: a transparent high performance inter-VM network loopback," Cluster Computing, Vol. 12, Issue 2, pp. 141-152, Jun. 2009.

[0082] [12] Y. Ren et al., "A Fast and Transparent Communication Protocol for Co-Resident Virtual Machines," 8th International Conference on Collaborative Computing, pp. 70-79, Oct. 2012.

[0083] [13] K. Kim et al., "Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen," Proc. of 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 11-20, Mar. 2008.

[0084] [14] H. Eiraku et al., "Fast Networking with SocketOutsourcing in Hosted Virtual Machine Environments," Proc. of 2009 ACM Symposium on Applied Computing, pp. 310-317, 2009.

[0085] [15] A. Burtsev et al., "Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances," Proc. of 2009 Conference on USENIX Annual Technical Conference, Jun. 2009.

[0086] [16] R. Jones et al. (2015, Jul. 21) Netperf [Online]. Available: http://github.com/HewlettPackard/netperf/tree/netperf -2.7.0 (downloaded 2020. Jun. 16)

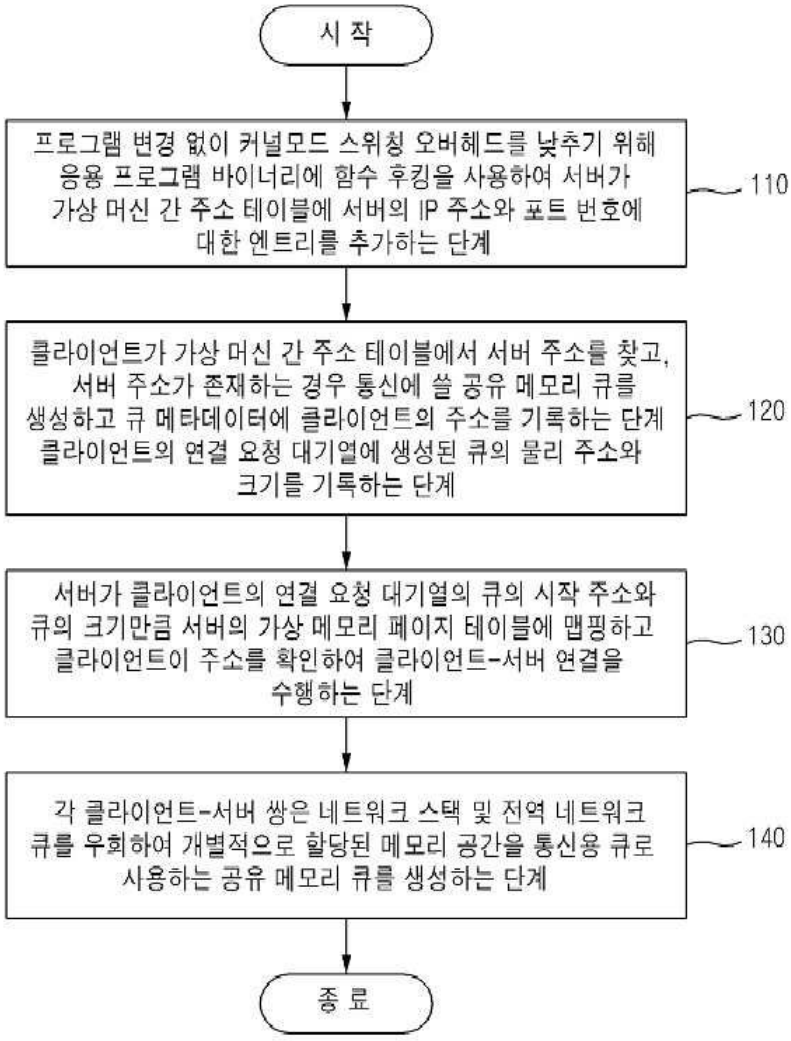
[0087] [17] K. Namhyung. (2017, Dec. 5) uftrace: Function Graph Tracer for C/C++/Rust [Online]. Available: http://git hub.com/namhyung/uftrace/tree/v0.8.2 (downloaded 2020. Jun. 16)

[0088] [18] (2019, Aug. 26) Linux Programmer's Manual [Online]. Available: https://man7.org/linux/man-pages/man8/ld. so.8.html (downloaded 2020, Mar. 1)

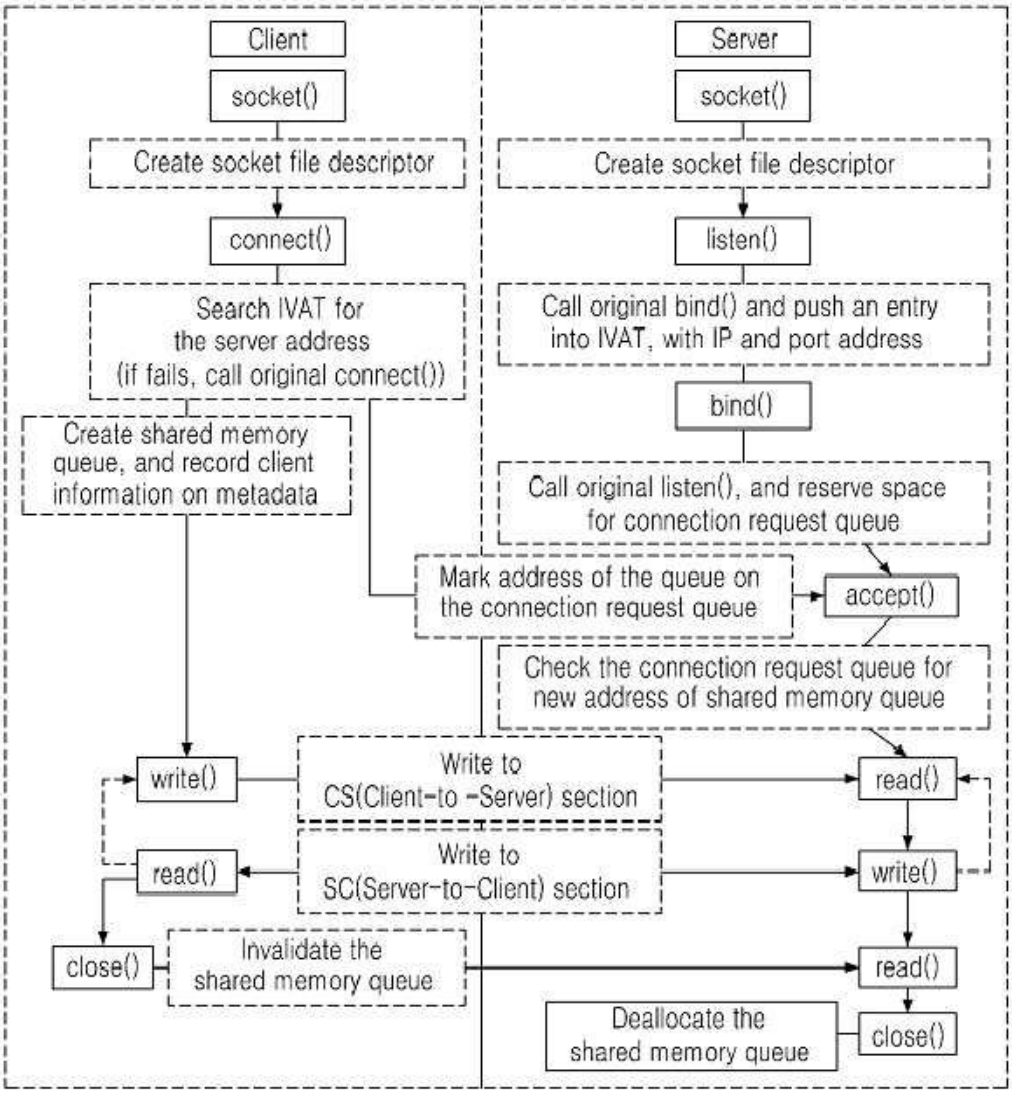
[0089] [19] "Exchange Instructions," Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 1, Chap. 7.3.1.2, Oct. 2019.

도면

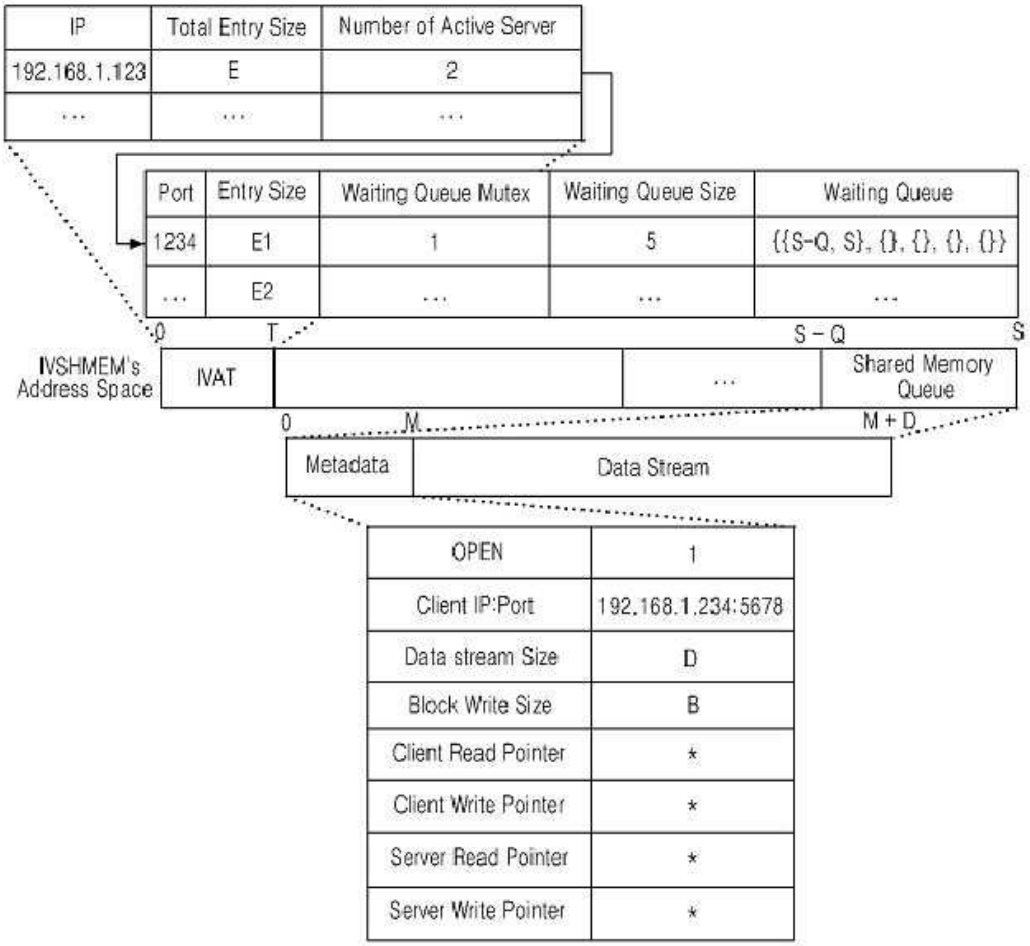
도면1


시 작
프로그램 변경 없이 커널모드 스위칭 오버헤드를 낮추기 위해 응용 프로그램 바이너리에 함수 후킹을 사용하여 서버가 가상 머신 간 주소 테이블에 서버의 IP 주소와 포트 번호에 대한 엔트리를 추가하는 단계
110
클라이언트가 가상 머신 간 주소 테이블에서 서버 주소를 찾고, 서버 주소가 존재하는 경우 통신에 쓸 공유 메모리 큐를 생성하고 큐 메타데이터에 클라이언트의 주소를 기록하는 단계 클라이언트의 연결 요청 대기열에 생성된 큐의 물리 주소와 크기를 기록하는 단계
120
서버가 클라이언트의 연결 요청 대기열의 큐의 시작 주소와 큐의 크기만큼 서버의 가상 메모리 페이지 테이블에 맵핑하고 클라이언트이 주소를 확인하여 클라이언트-서버 연결을 수행하는 단계
130
각 클라이언트-서버 쌍은 네트워크 스택 및 전역 네트워크 큐를 우회하여 개별적으로 할당된 메모리 공간을 통신용 큐로 사용하는 공유 메모리 큐를 생성하는 단계
140
종 료

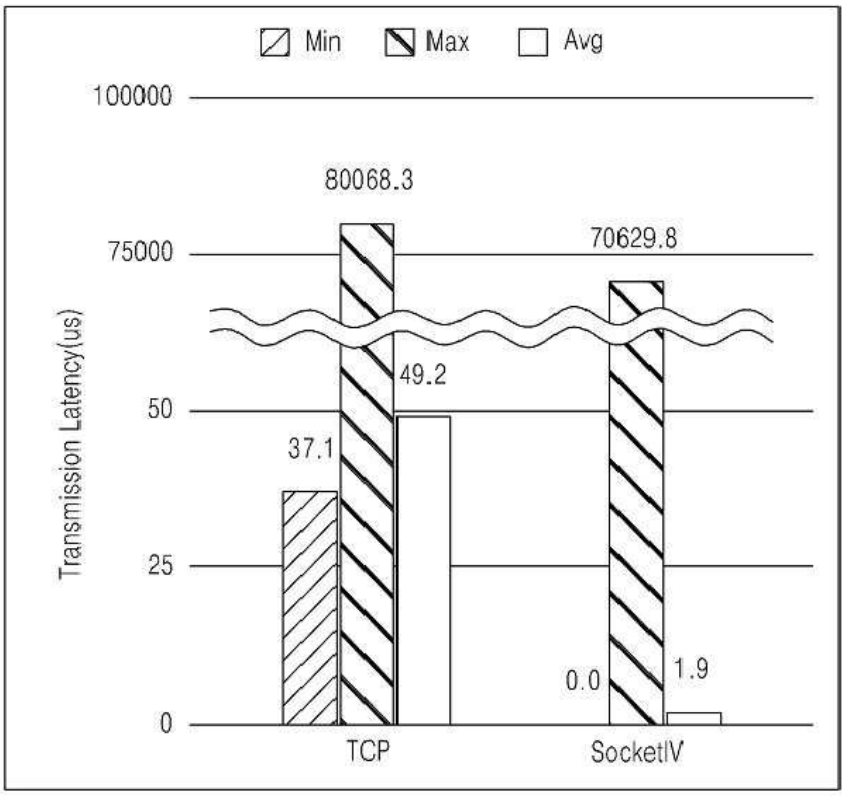
도면2


Client
socket()
Create socket file descriptor
connect()
Search IVAT for the server address (if fails, call original connect())
Create shared memory queue, and record client information on metadata
Server
socket()
Create socket file descriptor
listen()
Call original bind() and push an entry into IVAT, with IP and port address
bind()
Call original listen(), and reserve space for connection request queue
Mark address of the queue on the connection request queue
accept()
Check the connection request queue for new address of shared memory queue
write()
Write to CS(Client-to -Server) section
read()
read()
Write to SC(Server-to-Client) section
write()
close()
Invalidate the shared memory queue
read()
Deallocate the shared memory queue
close()

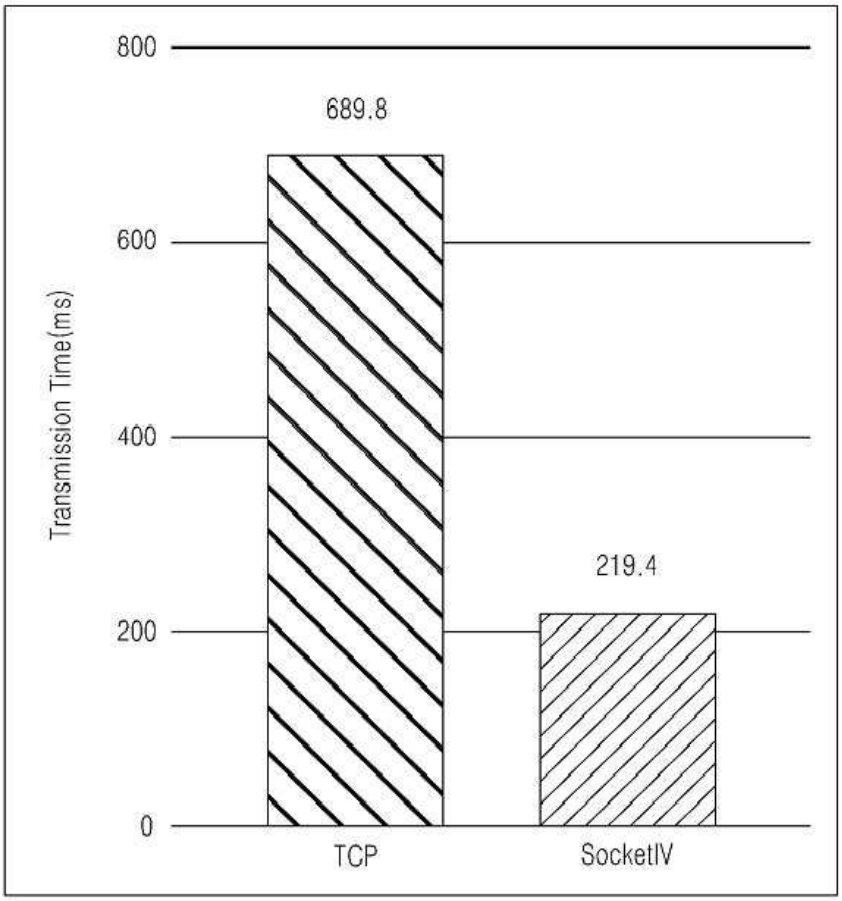
도면3


IP
Total Entry Size
Number of Active Server
192.168.1.123
E
2
Port
Entry Size
Waiting Queue Mutex
Waiting Queue Size
Waiting Queue
1234
E1
1
5
{{S-Q, S}, {}, {}, {}, {}}
E2
0
T
S - Q
S
IVSHMEM's Address Space
IVAT
Shared Memory Queue
0
M
M + D
Metadata
Data Stream
OPEN
1
Client IP:Port
192.168.1.234:5678
Data stream Size
D
Block Write Size
B
Client Read Pointer
*
Client Write Pointer
*
Server Read Pointer
*
Server Write Pointer
*

도면4


Min
Max
Avg
100000
75000
50
25
0
Transmission Latency(us)
80068.3
70629.8
49.2
37.1
0.0
1.9
TCP
SocketIV

도면5


800
689.8
600
Transmission Time(ms)
400
219.4
200
0
TCP
SocketIV