

Snap: a Microkernel Approach to Host Networking

Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli*, Michael Dalton*, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat

Google, Inc.

sosp2019-snap@google.com

Abstract—본 논문은 Snap이라는 host networking에 대한 microkernel에서 영감을 받은 접근 방식의 설계와 경험을 제시한다. Snap은 Google의 빠르게 진화하는 요구사항을 지원하는 userspace networking system으로, edge packet switching, cloud platform을 위한 virtualization, traffic shaping policy enforcement, 그리고 고성능 reliable messaging 및 RDMA와 유사한 service를 포함한 다양한 network 기능을 구현하는 유연한 module로 구성된다. Snap은 3년 이상 운영 환경에서 실행되어 있으며, 여러 대규모의 중요한 system의 확장 가능한 통신 요구사항을 지원하고 있다.

Snap은 address space isolation의 이점과 userspace software 개발의 생산성을 활용하고, machine에서 application을 마이그레이션하지 않고도 networking service를 투명하게 업그레이드할 수 있는 지원을 통해 새로운 networking 기능의 빠른 개발과 배포를 가능하게 한다. 동시에 Snap은 최소한의 상태 공유로 원칙적인 동기화를 촉진하는 modular architecture를 통해 강력한 성능을 달성하며, 새로운 kernel/userspace CPU scheduler 공동 설계를 통해 CPU 자원의 동적 확장과 함께 실시간 scheduling을 지원한다. 우리의 평가는 RPC workload에 대해 kernel networking stack과 비교하여 3배 이상의 Gbps/core 개선, 최대 5M IOPS/core의 software 기반 RDMA와 유사한 성능, 그리고 사용자 application에 거의 인지되지 않는 투명한 업그레이드를 보여준다. Snap은 우리 machine fleet의 절반 이상에 배포되어 있으며 수많은 team의 요구사항을 지원한다.

CCS Concepts • Networks → Network design principle; Data center network; • Software and its engineering; • Computer systems organization → Maintainability and maintenance;

Keywords: network stack, datacenter, microkernel, RDMA

1 Introduction

대규모 Internet service provider의 host networking 요구사항은 방대하고 빠르게 진화하고 있다. 지속적인 용량 증가는 edge switching과 bandwidth 관리에 대한 새로운 접근 방식을 요구하고, cloud computing의 부상은 풍부한 virtualization 기능을 필요로 하며, 고성능 분산 system은 지속적으로 더 효율적이고 낮은 지연시간의 통신을 추구한다.

기존의 kernel 기반 networking으로 이러한 요구사항을 실현하려는 우리의 경험은 긴 개발 및 릴리스 주기로 인해 방해받았다. 따라서 몇 년 전 우리는 공동 framework를 통해 networking 기능을 kernel에서 userspace module로 이동시키는 노력을 시작했다. 이 framework인 Snap은 높은 성능과 더불어 높은 개발자 생산성 및 릴리스 속도로 다양한 host networking 요구사항을 지원하는 풍부한 architecture로 발전했다.

Snap 이전에는 여러 측면에서 새로운 network 기능 및 성능 최적화를 개발하고 배포하는 능력이 제한적이었다. 첫째, kernel code를 개발하는 것은 느렸고 더 적은 수의 software

engineer를 활용했다. 둘째, kernel module reload를 통한 기능 릴리스는 기능의 일부만 포함했으며 종종 application의 연결을 끊어야 했고, machine 재부팅이 필요한 더 일반적인 경우에는 실행 중인 application을 machine에서 비워야 했다. 이러한 중단을 완화하기 위해 우리는 kernel 업그레이드 속도를 심각하게 늦춰야 했다. 실제로 kernel 기반 stack의 변경은 배포하는 데 1-2개월이 걸리는 반면, 새로운 Snap 릴리스는 주 단위로 우리 fleet에 배포된다. 마지막으로 Linux의 광범위한 일반성은 최적화를 어렵게 만들었고 upstream 변경으로 쉽게 중단되는 vertical integration 노력을 저해했다.

Snap architecture는 전통적인 operating system 기능이 일반 userspace process로 옮겨지는 microkernel 접근 방식과 유사성을 가진다. 그러나 이전의 microkernel system과 달리 Snap은 multicore

hardware를 활용하여 빠른 IPC를 제공하고, 표준 Linux distribution 및 kernel과 함께 userspace process로 실행되므로 전체 system이 이 접근 방식을 완전히 채택할 필요가 없다. 그럼에도 불구하고 Snap은 자원 관리 및 scheduling을 위한 중앙집중식 OS의 장점을 유지하며, 이는 기능을 조정되지 않은 application library에 넣는 userspace networking의 다른 최근 연구와 우리의 작업을 구별한다 [1], [14], [30], [51]. 우리의 접근 방식은 또한 Snap이 POSIX 및 socket system call과 같은 기존 interface와의 완전한 호환성의 복잡성을 피하고, 대신 우리 application과의 vertical integration 기회를 선호할 수 있게 한다. Snap을 통해 dataplane 상호작용은 lock-free shared memory queue를 통해 통신하는 맞춤형 interface를 통해 발생한다. 또한 기존 kernel 기능과 통합하는 use case의 경우, Snap은 Snap과 kernel 간에 packet을 효율적으로 이동시키기 위한 내부 개발 driver를 지원한다.

networking에서 더 높은 수준의 성능을 실현하는 것은 Moore's Law가 느려지고 더 빠른 storage device가 계속 등장함에 따라 중요하다. Snap을 통해 우리는 맞춤형 reliable transport와 통신 API를 구현하는 Pony Express라는 새로운 통신 stack을 만들었다. Pony Express는 web 검색에서 storage에 이르는 use case를 지원하며 우리 application에 상당한 통신 효율성과 지연시간 이점을 제공한다.

Snap의 architecture는 userspace networking, in-service upgrade, 중앙집중식 자원 accounting, 프로그래밍 가능한 packet 처리, kernel-bypass RDMA 기능, 그리고 transport, congestion control 및 routing의 최적화된 공동 설계에 대한 최근 아이디어의 구성이다. 우리는 Google 전체에 Snap을 구축하고 배포한 경험과, 여러 현대 system 설계 원칙의 통합되고 운영 환경에서 검증된 실현으로서 다음을 제시한다:

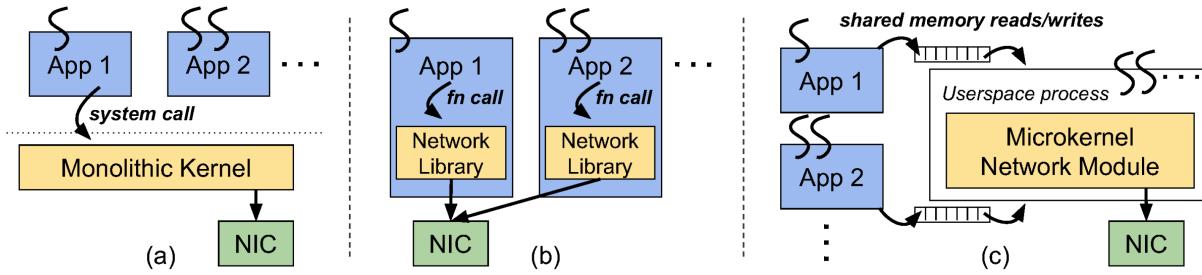


Figure 1: networking 기능을 구성하는 세 가지 다른 접근 방식: (a) application이 system call을 수행하는 전통적인 monolithic kernel을 보여주고, (b) 중앙집중화 없이 application 수준의 thread scheduling을 통한 처리를 수행하는 library OS 접근 방식을 보여주며, (c) 빠른 IPC를 위해 multicore를 활용하는 Snap microkernel과 유사한 접근 방식을 보여준다.

- Snap은 투명한 software 업그레이드를 통해 userspace에서 개발하는 microkernel에서 영감을 받은 접근 방식으로 높은 기능 개발 속도를 가능하게 한다. 이는 monolithic kernel의 중앙집중식 자원 할당 및 관리 기능의 이점을 유지하면서 기존 Linux 기반 system의 accounting 격차를 개선한다.
- operating system architecture의 실용적인 진화는 기존 kernel network 기능 및 application thread scheduler와의 상호운용성을 요구한다. Snap은 맞춤형 kernel packet injection driver와 맞춤형 CPU scheduler를 구현하여 새로운 application runtime의 채택을 요구하지 않고, Snap과 Linux kernel networking stack 모두를 통한 packet 처리를 동시에 요구하는 use case에 걸쳐 높은 성능을 유지하면서 상호운용성을 가능하게 한다.
- Snap은 packet 처리 기능을 “engine”이라고 하는 구성 가능한 단위로 캡슐화하며, 이는 modular CPU scheduling과 업그레이드 중 점진적이고 최소한의 중단을 일으키는 상태 전송을 모두 가능하게 한다.
- Pony Express를 통해 Snap은 RDMA 기능이 있는 “smart” NIC과 유사한 interface를 통해 OSI layer 4 및 5 기능에 대한 지원을 제공한다. 투명한 업그레이드 지원과 함께 이는 server 효율성과 처리량을 더욱 향상시키는 수단으로 emerging hardware NIC의 offload capability를 투명하게 활용할 수 있게 한다.
- I/O overhead를 최소화하는 것은 현대 분산 service를 확장하는 데 중요하다. 우리는 Snap과 Pony Express transport를 성능을 위해 신중하게 조정하여 baseline Linux kernel보다 3배 나은 transport 처리 효율성을 지원하고 5M ops/sec/core의 속도로 RDMA와 유사한 기능을 지원한다.

2 Microkernel Service로서의 Snap

monolithic operating system과 달리, Snap은 host networking 기능을 일반적인 Linux userspace process로 구현한다. 그러나 library OS model을 가정하는 다른 userspace networking 접근 방식과 비교할 때 [14], [30], [31], [51], [59], Snap은 전통적인 OS의 중앙집중식 조정 및 관리 가능성 이점을 유지한다. host는 Snap에 대해 특정 Linux capability를 활성화하여 device 자원에 대한 독점적 접근을 허용하며 (Snap은 root 권한으로 실행되지 않음), application은 shared memory queue를 통한 비동기식(fast path) 또는 Unix domain socket interface

를 통한 동기식(slow path)으로 데이터를 전송하는 library 호출을 통해 Snap과 통신한다.

Figure 1은 (a) 모든 networking 기능을 kernel-space에 배치하는 전통적인 접근 방식, (b) application thread에서 직접 hardware에 접근하는 library OS 접근 방식, 그리고 (c) 빠른 multicore IPC를 활용하여 network 기능을 user-level service로 구현하는 Snap 접근 방식 간의 차이를 보여준다.

Figure 2. Snap은 RPC serving을 중심으로 하는 control plane과 engine을 중심으로 하는 data plane으로 구분된다. Engine은 memory-mapped I/O를 통해 통신하는 packet processing pipeline을 캡슐화한다. Engine group은 engine을 core에 scheduling하는 방법을 결정한다. host kernel 상호작용을 제외하고 모든 code는 userspace에서 실행된다.

접근 방식 (c)는 여러 이점을 제공한다. 첫째, (a)의 중앙집중화 이점과 (b)의 userspace 개발 이점을 결합한다. 둘째, 새로운 networking 기능의 릴리스를 kernel 및 application binary 릴리스 주기 모두로부터 독립적으로 분리한다. 셋째, 낮은 지연시간을 달성하지만 일반적으로 이를 위해 spin-polling application thread에 의존하는 접근 방식 (b)의 library OS 구현과 달리, 접근 방식 (c)는 application threading 및 CPU 할당을 network service로부터 분리한다. 이는 networking 지연시간과 spin-polling capability를 system 수준 자원으로 관리할 수 있게 하며, 이는 일반적으로 수십 개의 독립적인 application을 실행하는 운영 machine에 필수적이다.

수년에 걸쳐 Snap은 host kernel로부터 networking 기능을 점진적으로 흡수해왔으며, memory 관리, thread scheduling, 기존 kernel networking, 그리고 non-networking I/O와 같은 다른 OS 기능은 풍부한 Linux ecosystem을 계속 활용한다. 이러한 측면은 전통적인 microkernel과 다르다. 그러나 Snap은 여전히 networking component와 다른 kernel component 간의 address space isolation, 개발 및 테스트 중 발견된 bug가 machine을 중단시키지 않음으로써 얻는 개발자 및 릴리스 속도, 그리고 전통적인 OS-bypass networking system에서 손실되는 풍부한 scheduling 및 관리 정책을 가능하게 하는 중앙집중화라는 microkernel의 이점을 유지한다.

초기 microkernel 연구는 inter-process communication (IPC)과 address space 변경에 기인한 상당한 성능 overhead를 보였지만 [15], [16], [20], 이러한 overhead는 오늘날 더 중요하다. 80년대와 90년대의 uniprocessor system과 비교할 때, 오늘날의 server는 수십 개의 core를 포함하고 있으며, 이는 microkernel 호출이 application cache locality를 유

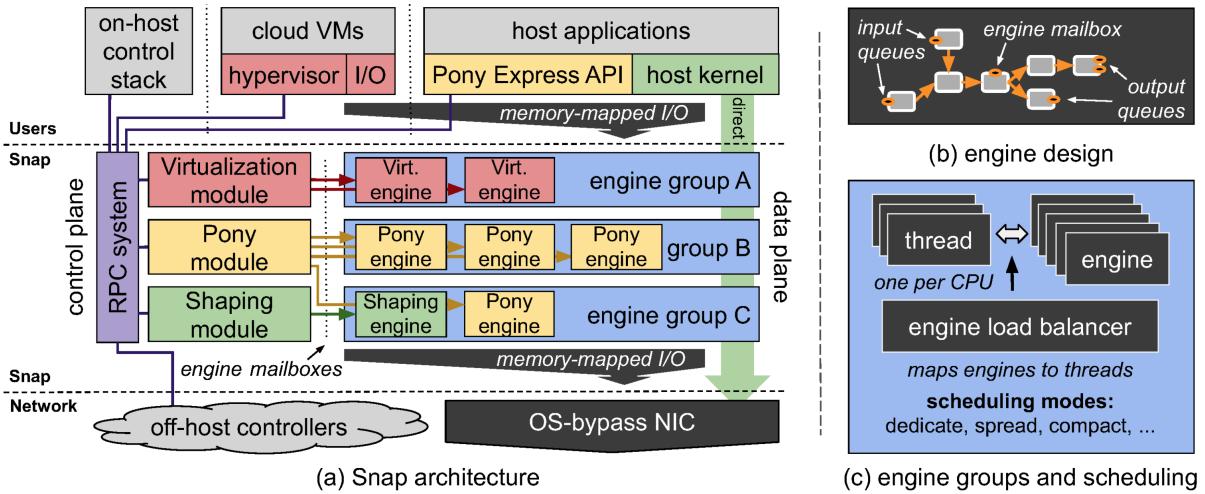


Figure 2: Snap은 RPC serving을 중심으로 하는 control plane과 engine을 중심으로 하는 data plane으로 구분된다. Engine은 memory-mapped I/O를 통해 통신하는 packet processing pipeline을 캡슐화한다. Engine group은 engine을 core에 scheduling하는 방법을 결정한다. host kernel 상호작용을 제외하고 모든 code는 userspace에서 실행된다.

지하면서 inter-core IPC를 활용할 수 있게 한다. 이 접근 방식은 IPC를 통해 통신할 상태가 거의 없을 때(zero-copy networking에서 일반적) 그리고 system call의 ring switch 비용을 포함으로써 전반적인 성능을 향상시킬 수도 있다. 더욱이 Meltdown [43]과 같은 최근의 보안 취약점은 monolithic kernel에서도 kernel/user address space isolation을 강제한다 [25]. 현대 processor의 tagged-TLB 지원, virtualization의 부활로 필요하게 된 간소화된 ring switching hardware, 그리고 L4 microkernel [29], FlexSC [58], SkyBridge [44]에서 탐구된 IPC 최적화 기법과 같은 기술은 현대 microkernel이 IPC를 통한 direct system call과 indirect system call 간의 성능 격차를 본질적으로 좁힐 수 있게 한다.

2.1 Snap Architecture 개요

Figure 2(a)는 Snap의 architecture와 외부 system과의 상호 작용을 보여준다. 이는 Snap이 “control plane”(왼쪽)과 “data plane”(오른쪽) component로 분리되어 있으며, engine이 data plane 작업의 캡슐화 단위임을 보여준다. engine의 예로는 network virtualization을 위한 packet 처리 [19], bandwidth 강제를 위한 pacing 및 rate limiting (“shaping”) [39], 그리고 Pony Express와 같은 stateful network transport (Section 3에서 제시)가 있다.

Figure 2(a)는 또한 서로 다른 component 유형에 걸쳐 사용되는 별도의 통신 패러다임을 보여준다: 왼쪽에서는 control plane component와 외부 system 간의 통신이 RPC를 통해 조율되고, 오른쪽에서는 data plane component로의 통신과 data plane component로부터의 통신이 memory-mapped I/O를 통해 발생하며, 중간에서는 control plane과 data plane component가 engine mailbox라고 하는 특수한 단방향 RPC 메커니즘을 통해 상호작용한다. 마지막으로, 그림은 공통의 scheduling discipline을 공유하는 group으로 구성된 engine 을 보여준다. Engine은 모든 guest VM I/O traffic, 모든 Pony Express traffic, 그리고 Snap에서 구현된 traffic이 필요한 host kernel traffic의 일부를 처리하는 것으로 표시된다.

2.2 Engine 설계 원칙

Engine은 Snap engine scheduling runtime에 의해 scheduling되고 실행되는 stateful, single-threaded task이다. Snap은 engine 개발자에게 OS-bypass networking, rate limiting, ACL enforcement, protocol 처리, 최적화된 data structure 등을 위한 library와, packet processing pipeline을 구성하기 위한 Click-style [47] pluggable “element”的 library를 갖춘 bare-metal programming 환경을 제공한다.

Figure 2(b)는 Snap engine의 구조와 queue 및 mailbox를 통한 외부 system과의 상호작용을 보여준다. engine의 input과 output은 userspace 및 guest application, kernel packet ring, NIC receive/transmit queue, support thread, 그리고 다른 engine을 포함할 수 있다. 이러한 각 경우에 lock-free 통신은 input 또는 output과 공유되는 memory-mapped 영역을 통해 발생한다. 때때로 engine은 inbound 작업을 알리기 위해 interrupt 전달(예: eventfd와 유사한 구조체에 쓰기)을 통해 output과 통신할 수도 있으며, 일부 engine scheduling policy에서는 input에서 interrupt를 수신할 수도 있다(Section 2.4 참조). 그러나 이러한 것들은 engine이 일반적으로 수행하는 유일한 통신 형태이다. 특히 real-time 속성을 유지하기 위해 Snap은 blocking synchronization에 의존하는 통신 형태를 금지한다.

Snap framework의 관점에서 얼마나 많은 engine이 필요한지, 그것들이 어떻게 인스턴스화되는지(정적 또는 동적으로), 그리고 작업이 engine 간에 어떻게 분배되는지(필요에 따라 NIC steering 기능을 활용)를 결정하는 것은 engine 개발자의 역할이다. Section 3.1과 6.4는 이 주제에 대한 일부 논의를 제공하지만, 그렇지 않으면 본 논문의 초점이 아니다.

2.3 Module과 Control-to-Engine 통신

Snap module은 control plane RPC service를 설정하고, engine을 인스턴스화하고, 이를 engine group에 로드하며, 해당 engine에 대한 모든 사용자 설정 상호작용을 proxy하는 역할을 담당한다. 예를 들어, Pony Express의 경우

Figure 2(a)에 표시된 “Pony module”은 사용자를 인증하고 local RPC system을 통해 file descriptor를 교환함으로써 사용자 application과 공유되는 memory 영역을 설정한다. 이는 또한 engine 생성/소멸, 호환성 검사, 그리고 policy 업데이트와 같은 다른 성능에 민감하지 않은 기능도 처리한다.

위의 기능을 제공하는 일부 control 작업, 예를 들어 사용자 command/completion queue 설정, shared memory 등록, encryption key 순환 등은 control component가 engine과 동기화해야 한다. Snap의 real-time, 고성능 요구사항을 지원하기 위해 control component는 *engine mailbox*를 통해 engine과 lock-free로 동기화한다. 이 mailbox는 깊이가 1인 queue로, control component가 engine에 의한 동기 실행을 위해 짧은 작업 섹션을 계시하며, 이는 engine의 thread에서 engine에 대해 non-blocking 방식으로 수행된다.

2.4 Engine Group과 CPU Scheduling

Snap은 engine의 scheduling latency, performance isolation, 그리고 CPU 효율성 간의 균형을 추구한다. 그러나 이 균형은 engine 유형마다 다르다: 일부 engine은 낮은 tail latency를 가장 중요하게 여기는 반면, 다른 engine은 고정된 CPU 예산 내에서 최대 공정성을 추구하고, 또 다른 engine은 무엇보다 효율성을 우선시하는 것을 목표로 한다.

Snap은 scheduling algorithm과 CPU 자원 제약을 지시하는 특정 scheduling mode를 가진 group으로 engine을 묶는 지원을 통해 이러한 각 경우를 수용한다. Figure 2(c)는 engine group의 구성을 보여주고 Figure 3은 Snap이 지원하는 세 가지 광범위한 scheduling mode 범주를 분류한다.

Core 전용화(Dedicating cores): 이 mode에서 engine은 다른 작업이 실행될 수 없는 전용 hyperthread에 고정된다. 이 mode는 CPU 사용률이 부하에 비례하여 확장되는 것을 허용하지 않지만, CPU 허용량이 전체 machine의 고정 예산일 때 적합할 수 있으며, 이 경우 spin polling을 통해 지연시간을 최소화할 수 있고, 선택적으로 userspace mwait [9]를 호출하여 전력을 절약할 수 있다. 그러나 정적 할당으로 인해 이 system은 부하 상태에서 부담을 받거나 과다 할당으로 인한 높은 비용에 직면할 수 있다. CPU가 제약될 때 scheduler는 높은 부하 조건에서 합리적인 performance isolation과 견고성을 제공하기 위해 engine 간에 CPU 시간을 공정하게 공유하려고 시도한다.

Engine 분산(Spread engines): 이 mode는 scheduling tail latency를 최소화하는 데 초점을 맞추면서 부하에 비례하여 CPU 소비를 확장한다. Snap 구현은 각 engine을 활성 상태 일 때만 scheduling하고 유휴 상태일 때는 interrupt 알림을 차단하는 고유한 thread에 바인딩한다. 그런 다음 interrupt는 NIC에서 또는 engine을 scheduling하기 위한 system call을 통해 application에서 트리거된다. schedulable core가 충분한 경우, 이 mode는 두 가지 이유로 최상의 tail latency 속성을 제공할 수 있다. 첫째, 잠재적으로 많은 engine의 작업을 적은 수의 thread 또는 core에 다중화하여 발생하는 scheduling 지연의 영향을 받지 않는다. 둘째, 내부 개발된 real-time kernel scheduling class(Section 2.4.1에서 자세히 설명)를 활용함으로써 engine은 기본 Linux CFS kernel scheduler [5]를 우회하고 interrupt 전달 시 사용 가능한 core에 우선순위를 가지고 빠르게 scheduling된다. 그러나 interrupt를 사용하면 신중하게 관리해야 하는 system 수준의 간접 효과가 있다.

Section 5.3에서는 thread가 저전력 sleep 상태에 있거나 non-preemptible kernel code를 실행하는 중인 core에 scheduling 될 때 발생할 수 있는 schedulability 문제에 대해 자세히 설명한다.

Engine 압축(Compacting engines): 이 mode는 가능한 한 적은 core에 작업을 축소하여 interrupt 기반 실행의 확장 이점과 core 전용화의 cache 효율성 이점을 결합한다. 그러나 위의 “engine 분산” scheduling mode처럼 순간적인 interrupt 신호에 의존하는 대신 부하 불균형을 감지하기 위해 engine queueing delay의 주기적인 polling에 의존한다. 따라서 재균형화 속도는 이러한 queueing delay를 polling하는 지연 시간에 의해 제약되며, 현재 설계에서 이는 non-preemptive이고 engine task가 고정된 latency budget 내에서 scheduler에게 제어를 반환해야 한다. engine을 polling하는 지연시간은 queueing 추정에서 통계적 신뢰도를 위한 지연과 실제로 engine을 다른 core로 이양하는 지연을 모두 포함하여 interrupt 신호의 지연시간보다 높을 수 있다.

구현 측면에서 이 scheduling mode는 Shenango [48]와 유사한 algorithm을 사용하여 측정된 모든 engine의 CPU 병목 queueing delay가 구성된 latency SLO 아래로 유지되는 동안 단일 thread에서 engine을 실행한다. 그런 다음 engine 실행과 교차하여 재균형화 합수가 주기적으로 여러 작업 중 하나를 수행한다. 한 작업은 queue 축적에 대응하여 engine을 다른 thread로 확장하는 것이다(필요한 경우 깨움). 다른 작업은 초과 용량을 감지하고 충분히 낮은 부하를 가진 engine을 다시 마이그레이션하는 것이다. 다른 작업에는 SLO의 제약 내에서 작업을 효과적으로 bin-pack하고 효율성을 극대화하기 위한 engine 압축 및 engine 교환이 포함된다. algorithm은 memory에서 동시에 업데이트되는 shared variable에 직접 접근하여 engine의 queueing 부하를 추정하며, 후속 load balancing 결정은 engine mailbox와 유사하지만 양쪽 모두에서 non-blocking인 message passing 메커니즘을 통해 영향을 받는 thread와 직접 동기화한다. 근본적으로 polling에 의해 구동되지만, 이 mode는 또한 full core 미만으로 축소하기 위해 일정 기간의 유휴 상태 후 interrupt 알림을 차단하는 것을 지원한다.

2.4.1 MicroQuanta Kernel Scheduling Class

CPU 자원을 동적으로 확장하기 위해 Snap은 지연시간에 민감한 Snap engine task와 다른 task 간에 core를 공유하는 유연한 방법을 제공하는 MicroQuanta라는 새로운 경량 kernel scheduling class와 함께 작동하며, 지연시간에 민감한 task의 CPU 점유율을 제한하면서 동시에 낮은 scheduling latency를 유지한다. MicroQuanta thread는 매 period time unit마다 구성 가능한 runtime 동안 실행되며, 나머지 CPU 시간은 고정된 전통적인 time slot이 아닌 높은 우선순위와 낮은 우선순위 task에 대한 fair queuing algorithm의 변형을 사용하여 다른 CFS-scheduled task에 사용 가능하다.

MicroQuanta는 Snap의 CFS task에 의해 실행 가능한 core에서 우선순위를 얻는 견고한 방법으로, 중요한 per-core kernel thread의 starvation을 방지한다 [7]. 공유 패턴은 real-time group scheduling bandwidth control을 갖춘 SCED_DEADLINE 또는 SCED_FIFO와 유사하다. 그러나 MicroQuanta는 훨씬 낮은 지연시간으로 scheduling하고 훨씬 더 많은 core로 확장된다. 다른 Linux real-time

| scheduling mode | CPU resources | scheduling latency | | CPU efficiency | visualization |
|---------------------------|---------------|--------------------|------------|----------------|---------------|
| | | median | tail | | |
| dedicating cores | static | 0-1μs | 0*-100+μs | poor | |
| spreading engines | dynamic | 3-10μs | 10-30**μs | good | |
| compacting engines | dynamic | 0-5μs | 50-100**μs | excellent | |

* 0μs tail scheduling latency under “dedicating cores” possible only when running a single engine per core

** assumes minimal tail latency impact due to low-power sleep states and/or possible preemption failure

Figure 3: Snap은 서로 다른 자원, 지연시간, 그리고 효율성 속성을 가진 세 가지 다른 engine scheduling mode를 지원한다. visualization 열에서 사각형은 두 개의 logical core를 포함하는 physical core이며, Snap이 소비하는 부분은 진한 회색으로, non-Snap은 연한 파란색으로, 유휴 시간은 흰색으로 표시된다.

scheduling class는 bandwidth 제어를 위해 per-CPU tick 기반 및 global high-resolution timer를 모두 사용하는 반면, MicroQuanta는 per-CPU high-resolution timer만 사용한다. 이는 microsecond 단위의 확장 가능한 time slicing을 가능하게 한다.

2.5 CPU 및 Memory Accounting

강력한 CPU 및 memory accountability는 VM과 job이 machine에 집약적이고 동적으로 다중화되기 때문에 datacenter 및 cloud computing 환경에서 중요하다. 예를 들어, 이전 연구 [12], [36]는 kernel networking stack의 soft-interrupt accounting의 단점을 보여주는데, softirq가 해당 application으로 향하는 데이터를 위한 softirq 처리인지 여부와 무관하게 core에서 실행 중인 어떤 application으로부터든 CPU 시간을 훔치기 때문이다. Snap은 내부 개발된 Linux kernel interface를 사용하여 application container에 CPU와 memory를 청구함으로써 application을 대신하여 소비된 CPU와 memory를 모두 해당 application에 정확하게 귀속시켜 강력한 accounting과 isolation을 유지한다. 이를 통해 Snap은 system을 oversubscribing하지 않고 Snap CPU 처리와 per-user memory 소비(per-user data structure에 대해)를 확장할 수 있다.

2.6 Security

Snap engine은 민감한 application 데이터를 처리할 수 있으며, 서로 다른 신뢰 수준을 가진 잠재적으로 여러 application을 대신하여 작업을 동시에 수행한다. 따라서 security와 isolation은 매우 중요하다. monolithic kernel과 달리 Snap은 줄어든 권한을 가진 특별한 non-root 사용자로 실행되지만, packet과 payload가 잘못 전달되지 않도록 주의를 기울여야 한다. Snap과의 상호작용을 설정하는 application은 표준 Linux 메커니즘을 사용하여 그 신원을 인증한다.

Snap userspace 접근 방식의 한 가지 이점은 software 개발이 일반 user-level software의 security와 견고성을 향상시키기 위해 개발된 memory access sanitizer 및 fuzz tester와 같은 광범위한 내부 도구를 활용할 수 있다는 것이다. Section 2.4의 CPU scheduling mode는 또한 특정 application을 위한 engine을 실행하는 core를 다른 application을 위한 engine을 실행하는 core로부터 명확하게 분리함으로써 Spectre-class 취약점 [37]을 완화하는 옵션을 제공한다.

3 Pony Express: Snap Transport

수년에 걸쳐 Snap을 뒷받침하는 architecture는 cloud VM을 위한 network virtualization [19], Internet peering을 위한 packet-processing [62], 확장 가능한 load balancing [22], 그리고 본 논문의 나머지 부분에서 초점을 맞추는 reliable transport 및 통신 stack인 Pony Express를 포함한 여러 networking application의 운영 환경에서 사용되어 왔다. 우리의 datacenter application은 점점 더 CPU 효율적이고 낮은 지연시간의 통신을 추구하며, Pony Express는 이를 제공한다. 이는 reliability, congestion control, 선택적 ordering, flow control, 그리고 remote data access operation의 실행을 구현한다.

TCP/IP를 재구현하거나 기존 transport를 리팩토링하는 대신, 우리는 더 효율적인 interface, architecture, 그리고 protocol을 혁신하기 위해 Pony Express를 처음부터 시작했다. Pony Express에 대한 application interface는 packet 수준 또는 byte-streaming socket interface와는 달리 비동기 operation 수준 command 및 completion을 기반으로 한다. Pony Express는 (two-sided) messaging operation과 RDMA가 한 예인 one-sided operation을 모두 구현한다. one-sided operation은 원격 application thread 상호작용을 포함하지 않으므로 remote data access를 위해 application thread scheduler를 호출하는 것을 피한다.

transport 처리 affinity를 application CPU에 로컬하게 유지하려고 시도하는 Linux TCP stack과 대조적으로 [8], [21], Pony Express는 Snap을 통해 application CPU와 별도의 thread에서 transport 처리를 실행하며, Snap은 engine thread를 PCI에 연결된 NIC의 NUMA node에 affinize한다. Snap을 통해 Pony Express는 주로 application 작업보다는 다른 engine 및 다른 transport 처리 작업과 CPU를 공유한다. 이는 더 나은 batching, code locality, 감소된 context switching, 그리고 spin polling을 통한 지연시간 감소 기회를 가능하게 한다. 포괄적인 설계 원칙은 zero-copy capability가 주어지면 NIC NUMA node locality와 transport layer 내의 locality가 application thread와의 locality보다 함께 더 중요하다는 것이다.

3.1 Architecture 및 구현

Pony Express의 architecture는 Figure 4에 설명되어 있다. Client application은 Pony Express client library API를 통해 잘

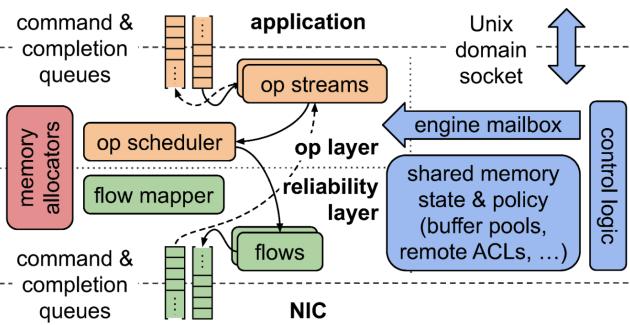


Figure 4: 왼쪽의 Pony Express engine의 architecture와 오른쪽의 관련 control logic.

알려진 주소의 Unix domain socket을 통해 Pony Express에 접속한다. Section 2.3에서 이전에 설명한 바와 같이, 이 socket은 domain socket의 ancillary data 기능을 사용하여 process 간에 tmpfs로 백업된 file descriptor를 전달함으로써 application과 Pony Express 간의 shared memory 영역을 bootstrap한다. 이러한 shared memory 영역 중 하나는 비동기 operation을 위한 command 및 completion queue를 구현한다. application이 operation을 호출하려고 할 때 command queue에 command를 작성한다. 그런 다음 application thread는 completion queue를 spin-poll하거나 completion이 작성될 때 thread 알림을 받도록 요청할 수 있다. 다른 shared memory 영역은 zero-copy request/response payload를 위해 application 영역에 매핑된다.

Pony Express는 stream, operation, flow, packet memory, 그리고 application buffer pool을 포함하는 상태의 동적 생성 및 관리를 최적화하기 위해 맞춤형 memory allocator를 구현한다. 이러한 구조는 Section 2.5에서 논의된 메커니즘을 사용하여 application memory container에 적절하게 청구된다.

Transport 설계: Pony Express는 transport logic을 두 계층으로 분리한다: upper layer는 application 수준 operation을 위한 state machine을 구현하고 lower layer는 reliability와 congestion control을 구현한다. lower layer는 network를 통해 한 쌍의 engine 간의 reliable flow를 구현하고 flow mapper는 application 수준 connection을 flow에 매핑한다. 이 lower layer는 개별 packet을 안정적으로 전달하는 것만 담당하는 반면, upper layer는 특정 operation과 관련된 재정렬, 재조립, 그리고 semantic을 처리한다. Pony Express와 함께 배포하는 congestion control algorithm은 Timely [45]의 변형이며 전용 fabric QoS class에서 실행된다. 새로운 버전의 Pony Express를 빠르게 배포할 수 있는 능력은 congestion control의 개발과 투명에 크게 도움이 되었다.

Pony Express를 사용하여 우리는 fleet에 여러 릴리스 버전이 존재할 수 있는 시간 범위 동안 이전 버전과의 호환성을 유지하면서 내부 wire protocol을 주기적으로 확장하고 변경한다. 주간 릴리스 주기가 이 시간 범위를 작게 만들지만, 사용자를 전환할 때 여전히 상호운용성과 하위 호환성이 필요하다. 현재 우리는 원격 engine에 연결할 때 사용 가능한 wire protocol 버전을 광고하기 위해 out-of-band 메커니즘 (TCP socket)을 사용하고 최소 공통분모를 선택한다. fleet이 전환되면 사용되지 않는 버전에 대한 code를 제거한다.

Engine 작동: Pony Express engine은 들어오는 packet을 처리하고, application과 상호작용하며, messaging 및 one-sided operation을 진행하기 위한 state machine을 실행하고, 나가는 packet을 생성한다. NIC receive queue를 polling할 때 처리되는 packet의 최대 수는 latency vs. bandwidth trade-off를 조정하도록 구성 가능하며, 현재 기본값은 batch당 16개의 packet이다. application command queue polling도 유사하게 구성 가능한 batch 크기를 사용한다. 들어오는 packet 및 application의 command와 NIC transmit descriptor slot의 사용성을 기반으로 engine은 전송을 위한 새 packet을 생성한다. slot 사용성이 기반한 이러한 just-in-time packet 생성은 NIC가 전송할 수 있을 때만 packet을 생성하도록 보장한다 (engine에서 per-packet queueing이 필요하지 않음). 전반적으로 작업 scheduling은 engine 실행 시간에 대한 엄격한 경계를 제공하고, 공정성을 구현하며, 효율성을 위해 기회주의적으로 batching을 활용한다.

Pony Express를 사용하는 application은 자체 *exclusive engine*을 요청하거나 사전 로드된 *shared engine* 세트를 사용할 수 있다. Section 2에서 논의한 바와 같이 engine은 Snap 내에서 scheduling 및 load balancing의 단위이므로, application-exclusive engine을 사용하면 application은 다른 application과 CPU 및 scheduling 결정의 운명을 공유하지 않음으로써 더 강력한 performance isolation을 받지만, 잠재적으로 더 높은 CPU 및 memory 비용이 발생한다. application은 강력한 isolation이 덜 중요할 때 shared engine을 사용한다. 이는 경제적 이점이 있으며 진행 중인 작업 영역인 많은 수의 engine으로 확장하는 것과 관련된 과제도 줄여준다.

3.2 One-Sided Operation

One-sided operation은 목적지에서 어떤 application code도 포함하지 않으며, 대신 Pony Express engine 내에서 완전히 완료될 때까지 실행된다. thread를 dispatch, 알림, 그리고 scheduling하기 위한 application thread scheduler(즉, Linux CFS)의 호출을 피하는 것은 CPU 효율성과 tail latency를 상당히 향상시킨다. RDMA Read는 one-sided operation의 고전적인 예이며, 이 section은 Pony Express가 유사한 이점을 어떻게 가능하게 하는지 다룬다.

임의의 사용자 logic을 구현하는 handler를 application이 작성할 수 있게 하는 RPC system과 달리, Pony Express의 one-sided operation은 Pony Express 릴리스의 일부로 사전 정의되고 사전 설치되어야 한다. 우리는 임의의 application 정의 operation을 허용하지 않고 대신 기능 요청 및 검토 프로세스를 따른다. one-sided logic이 Snap의 address space에서 실행되므로, application의 thread가 logic을 실행하지 않더라도 application은 원격으로 접근 가능한 memory를 명시적으로 공유해야 한다. two-sided messaging과 함께 spin-polling application thread를 사용하여 모든 operation에 대해 application thread scheduler의 호출을 피하는 것이 확실히 가능하지만, 모든 application에 대한 spin-polling thread에는 단점이 있다 (Section 6.1에서 추가 논의).

Pony Express의 software 유연성은 기본 remote memory read 및 write operation을 넘어서는 더 풍부한 operation을 가능하게 한다. 이전 연구 [33], [38]에서 언급된 바와 같이, RDMA의 효율성 이점은 여러 network round-trip을 거쳐야 하는 remote data structure에서 사라질 수 있다. 예를 들어, 우리는 접근할 실제 memory target을 결정하기 위해 application이 채운 indirection table을 참조하는 맞춤형 *indirect read*

operation을 지원한다. 기본 remote read와 비교하여 indirect read는 data structure가 단일 pointer indirection을 필요로 할 때 달성 가능한 operation 속도를 효과적으로 두 배로 만들고 지연시간을 절반으로 줄인다. 또 다른 맞춤형 operation은 *scan and read*로, 작은 application 공유 memory 영역을 스캔하여 인자를 일치시키고 일치와 관련된 pointer에서 데이터를 가져온다. 이러한 operation은 모두 운영 system에서 사용된다.

3.3 Messaging 및 Flow Control

Pony Express는 RPC 및 non-RPC use case를 위한 send/recv messaging operation을 제공한다. HTTP2 및 QUIC [40]와 마찬가지로, 우리는 독립적인 message의 head-of-line blocking을 피하기 위해 message stream을 생성하는 메커니즘을 제공한다. (TCP socket에서 발견되는) per-connection receive window of byte가 아니라, flow control은 receiver 주도 buffer posting과 더 작은 message를 위해 credit을 사용하여 관리되는 shared buffer pool의 혼합을 기반으로 한다.

one-sided operation에 대한 flow control은 더 미묘한데, 예를 들어 initiator가 끝없는 read request stream으로 machine을 공격하는 것을 막을 application에서 볼 수 있는 메커니즘이 없기 때문이다. 즉, byte streaming socket 및 two-sided messaging과 달리, application은 socket에서 읽기를 거부하거나 message를 받기 위해 memory를 post하여 sender를 일시 중지할 수 없다. one-sided client가 connection에서 outstanding operation을 제한하거나 다른 client-side backoff 전략을 사용하여 server overload를 완화할 수 있지만, 우리는 궁극적으로 Section 2에서 논의된 Snap CPU scheduling 및 accounting 메커니즘으로 이 문제를 해결한다. 특히 one-sided operation의 사용자는 공정한 CPU 시간 할당을 받는 Pony Express engine을 인스턴스화한다. client가 one-sided operation으로 이 engine을 압도하면 필연적으로 request packet이 drop되고 congestion control이 back off한다. 따라서 one-sided 사용자는 일반적으로 이러한 시나리오를 피하지만, one-sided operation은 공정한 공유를 위해 상위 수준 flow control 메커니즘보다는 congestion control 및 CPU scheduling 메커니즘에 의존하는 것으로 대체된다.

3.4 Hardware Offload

Moore's Law가 느려짐에 따라 offload 및 기타 accelerator의 추구는 전략적으로 중요하다. Pony Express는 memory copy operation을 offload하기 위한 Intel I/OAT DMA device [2]를 포함한 stateless offload를 활용한다. 우리는 이 capability를 Snap에서 사용 가능하게 하기 위해 kernel module을 개발했으며, DMA 주변의 비동기 상호작용이 Linux의 동기 socket system call과 비교하여 지속적으로 실행되는 packet 처리

pipeline을 가진 Snap에 자연스럽게 적합하다는 것을 발견했다. Section 5.1은 Pony Express에 대한 copy offload로부터의 전반적인 CPU 효율성 향상을 정량화한다. Pony Express는 또한 다른 stateless NIC offload를 활용한다. 한 예는 각 packet에 대한 end-to-end invariant CRC32 계산이다. Section 6.5는 stateful offload의 사용과 이들이 Pony Express와 어떻게 상호작용하는지에 대한 우리의 관점을 논의한다.

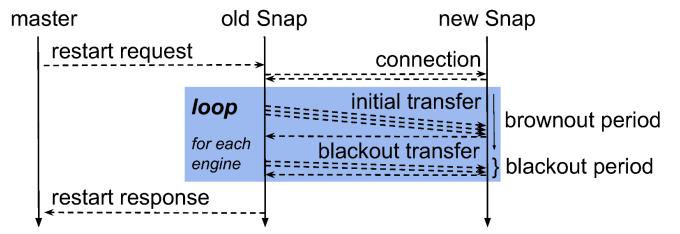


Figure 5: 투명한 업그레이드 sequence diagram.

4 Transparent Upgrades

Snap은 host networking에서 혁신할 수 있는 우리의 능력을 상당히 향상시킨다. 핵심 enabler는 실행 중인 application을 중단하지 않고 Snap의 새 버전을 릴리스할 수 있는 능력이다. 이 section은 우리의 접근 방식을 설명한다.

초기 목표는 개발자에게 릴리스 간에 상당한 변경을 수행할 때 최대한의 유연성을 제공하는 것이었다(API 및 wire format 호환성 문제를 염두에 두고). 이는 code를 변경하는 동안 binary 상태를 memory에 유지하는 scheme을 배제했다. 대신 업그레이드 중에 실행 중인 Snap 버전은 모든 상태를 새 버전과 공유되는 memory에 저장된 중간 형식으로 serialize 한다. virtual machine migration 기법 [18]과 마찬가지로 업그레이드는 network stack을 사용할 수 없는 blackout period를 최소화하기 위해 두 단계로 진행된다. 초기 brownout phase는 성능 영향이 최소화된 준비 background 전송을 수행한다.

두 번째 목표는 업그레이드를 처리하는 code 작성과 연결 손실 측면에서 application에 대한 중단을 최소화하는 것이다. 우리는 200msec 이하의 blackout period를 목표로 하며, 이 목표를 달성하기 위해 Snap은 한 번에 하나씩 engine을 완전히 마이그레이션하면서 점진적으로 업그레이드를 수행한다. 운영 환경에서 실행되는 engine의 수가 증가함에 따라 이 접근 방식은 다른 engine의 전송으로 인해 단일 engine이 장기간 blackout을 경험하는 것을 방지하기 위해 필요하게 되었다. 이 외에도 cluster 전체에 걸친 점진적인 업그레이드 프로세스를 통해 우리의 기존 application은 주당 한 번 발생하는 100밀리초의 통신 끊김을 인지하지 못한다는 것을 발견했다. 이 blackout period 동안 packet 손실이 발생할 수 있지만, end-to-end transport protocol은 이를 congestion으로 인한 packet 손실처럼 허용한다. 중요한 것은 인증된 application connection이 (양쪽 끝에서) 설정된 상태로 유지되고 양방향 message stream과 같은 상태가 그대로 유지된다는 것이다.

Figure 5는 업그레이드 flow를 보여준다. 먼저 Snap “master” daemon이 Snap의 두 번째 instance를 시작한다. 실행 중인 Snap instance는 이에 연결한 다음 한 번에 하나씩 각 engine에 대해 control plane Unix domain socket connection을 일시 중단하고 shared memory file descriptor handle과 함께 background에서 이를 전송한다. 이는 Unix domain socket의 ancillary fd-passing 기능을 사용하여 수행된다. control plane connection이 전송되는 동안 새 Snap은 shared memory mapping을 재설정하고, queue, packet allocator, 그리고 이전 engine이 여전히 작동하는 동안 engine의 새 instance와 관련된 다양한 기타 data structure를 생성한다. 완료되면 이전 engine은 packet 처리를 중단하고, NIC receive filter를 분리하며, 나머지 상태를 tmpfs shared memory volume으로 serialize

하여 blackout period를 시작한다. 그런 다음 새 engine은 동일한 NIC filter를 연결하고 상태를 deserialize한다. 모든 engine이 이런 식으로 전송되면 이전 Snap이 종료된다.

5 Evaluation

이 section은 Pony Express 통신 stack에 초점을 맞춰 Snap의 성능과 동적 확장성을 평가한다. 우리는 Pony Express를 실행하는 Snap (“Snap/Pony”)의 isolation 및 공정성 속성을 측정하고 투명한 업그레이드를 위한 점진적 상태 전송도 시연한다. 아래의 모든 결과는 hyperthreading이 활성화된 상태에서 CPU 사용량을 보고하므로 보고된 단일 “core” 또는 “CPU”는 단일 hardware thread context를 의미한다. Section 5.1–5.3에서는 Linux kernel TCP/IP stack과 비교하는데, 이는 우리 조직의 baseline일 뿐만 아니라 우리가 아는 한 kernel TCP/IP 구현이 datacenter 환경을 위한 유일하게 널리 배포되고 운영 환경에서 검증된 대안이기 때문이다. Section 5.4에서는 우리 조직 내에서 사용 가능한 hardware RDMA 기술과의 정성적 비교도 제공한다.

5.1 Baseline Throughput 및 Latency

Snap/Pony의 baseline 성능을 설정하기 위해 동일한 top-of-rack switch에 연결된 한 쌍의 machine 간 성능 측정으로 시작한다. machine은 Intel Skylake processor와 100Gbps NIC를 갖추고 있다. 우리는 Neper [3] utility (Netperf와 유사하지만 여러 동시 stream을 통해 데이터를 전송하는 지원이 있음)를 사용하여 kernel TCP socket과 비교하는 반면, Snap/Pony의 경우 유사하게 구성된 맞춤형 (non-socket) benchmark를 사용한다. 모든 benchmark는 송수신을 위해 단일 application thread를 사용하며, 다음 section에서 여러 process 및 thread를 사용한 확장 성능을 살펴본다. Pony Express engine은 항상 spin하며 end-to-end latency를 측정하기 위해 application thread가 spin하는지 여부를 변경한다.

Table 1은 Neper를 사용하는 우리 kernel 구성에서 TCP의 baseline single-stream throughput이 22Gbps임을 보여준다. 또한 application과 kernel 모두에 걸친 측정된 평균 core 사용률이 약 1.2 core임을 보여준다. 대조적으로 Snap/Pony는 1.0 Snap core와 0.05 application core를 사용하여 38Gbps를 제공한다. 최근 우리는 또한

switch와 fabric에서 더 큰 MTU 크기를 실험하기 시작했다. 5000B MTU¹이 Snap/Pony의 single-core throughput을 67Gbps 이상으로 증가시키며, I/OAT receive copy offload (zero-copy transmit를 보완하기 위해)를 활성화하면 단일 core를 사용하여 throughput이 80Gbps 이상으로 더욱 증가함을 보여준다. 마지막으로 Table 1은 동시에 활성화된 stream 수가 증가함에 따라 TCP 성능이 상당히 저하되는 반면, Snap/Pony는 이 benchmark에서 과도한 context switching을 피하고 좋은 locality를 유지함으로써 견고한 connection 확장을 보여준다.

Figure 6(a)는 작은 message를 송수신하는 두 application 간의 평균 round-trip latency (two-sided)를 보여준다. 이 구성에서 Neper의 TCP_RR은 TCP가 23μsecs의 baseline latency를 제공함을 보여준다. Snap/Pony를 사용하는 유사한 application은 18μsecs의 latency를 제공한다. Pony Express completion queue에서 spin-poll하도록 application을 구성하면 latency가 10μsecs 미만으로 감소하는 반면, Linux의 busy-polling socket 기능을 사용하면 TCP_RR latency가 18μsecs로

감소한다. 우리는 또한 one-sided access로 달성한 Snap/Pony latency를 보여주며, 이는 latency를 8.8μsecs로 더욱 감소시킨다.

| | # stream | CPU/sec | Gbps |
|------------------------|------------|---------|------|
| Linux TCP | 1 stream | 1.17 | 22.0 |
| | 200 stream | 1.15 | 12.4 |
| Snap/Pony | 1 stream | 1.05 | 38.5 |
| | 200 stream | 1.05 | 39.1 |
| Snap/Pony w/ 5kB MTU | 1 stream | 1.05 | 67.5 |
| | 200 stream | 1.05 | 65.7 |
| Snap/Pony w/ 5kB+I/OAT | 1 stream | 1.05 | 82.2 |
| | 200 stream | 1.05 | 80.5 |

TABLE I: TCP (Neper 사용) 및 Snap/Pony (내부 도구 사용)에 대해 측정된 Throughput. 모든 도구는 부하를 구동하기 위해 단일 application thread를 사용한다.

5.2 확장 및 Performance Isolation

이 section은 단일 core를 넘어 확장할 때 Snap/Pony 성능을 평가한다. 실험은 모두 동일한 switch에 연결된 42개 machine의 rack을 사용한다. 각 machine은 50Gbps NIC와 dual-socket 구성의 Intel Broadwell processor를 갖추고 있다. 각 machine에 10개의 background job을 scheduling하며, 각 job은 Poisson 분포를 따르는 선택된 속도로 RPC를 통해 통신한다. 각 RPC는 420개의 전체 job 중 하나를 무작위로 target으로 선택하고 추가 계산 없이 1MB (cache resident) 응답을 요청한다. 제공된 부하는 각 job의

request 속도를 증가시켜 변경된다. 측정된 CPU 시간은 모든 application, system, 그리고 interrupt 시간을 포함한다. Snap/Pony의 MTU 크기는 5000B이다. TCP의 경우 4096B이며, 이는 우리 조직에서 TCP에 대한 유사한 “large MTU” 설정이다. 안타깝게도 값이 다르지만, 데이터가 여전히 대표적이라고 믿을 만큼 충분히 가깝다. 1MB RPC로 background 부하를 생성하는 10개의 job 외에도, 유사하게 무작위 목적지를 선택하지만 작은 RPC의 latency만 측정하는 단일 latency prober job을 각 machine에 scheduling한다. 우리는 이러한 측정의 99th percentile latency를 보고한다. Snap/Pony의 경우 각 job은 자체 exclusive engine을 요청하며, Section 2.4의 두 가지 scheduling mode인 “spreading engine”과 “compacting engine”을 평가한다.

Figure 6(b)는 전체 machine당 CPU 사용량을 보여주고 Figure 6(c)는 총 background 부하가 machine당 80Gbps로 증가함에 따라 99th percentile prober latency를 보여준다 (50Gbps NIC에서 양방향 traffic 포함). 우리는 먼저 두 Snap engine scheduler가 모두 부하에 비례하여 CPU 소비를 확장하는 데 성공했음을 관찰한다. 다음으로 Snap은 일반적으로 batching 효율성의 결과로 CPU 소비의 sub-linear 증가를 보여준다. 마지막으로 동적 확장 메커니즘이 압박을 받을 수록 Snap과 TCP 간의 상대적 CPU 시간 차이가 증가한다. 8Gbps의 제공 부하에서 TCP와 Snap 모두의 CPU 시간이 비슷하지만, 80Gbps의 제공 부하에서 Snap은 TCP 보다 3x 이상 효율적이다. 우리는 성능 향상이 copy 감소, fine-grained synchronization 회피, 5000B 대 4096B MTU 차이, 그리고 Snap compacting scheduler의 경우 더 끄거운 instruction 및 data cache의 조합에 기인한다고 본다. 특히 이러한 더 큰 1MB RPC의 경우 kernel TCP socket system call

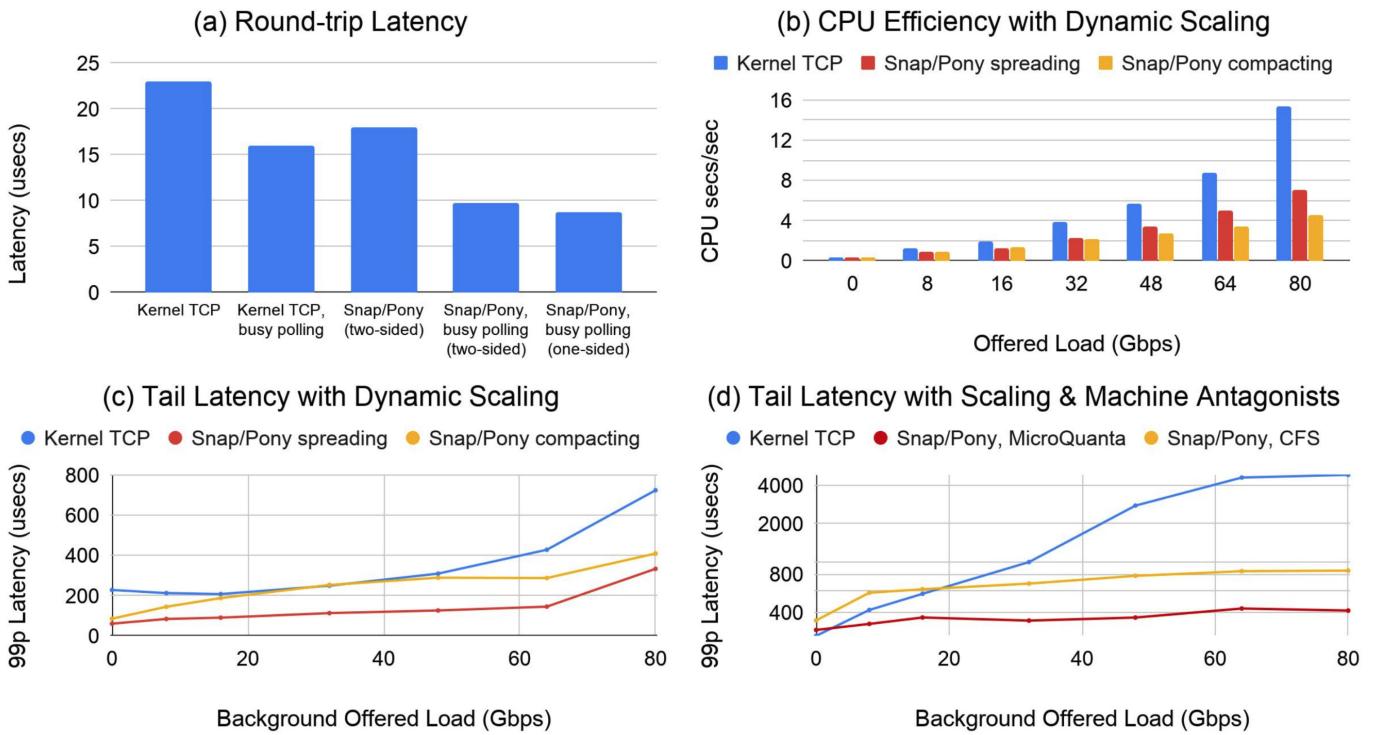


Figure 6: Benchmark 데이터. Graph (a)는 동일한 top-of-rack switch에 연결된 두 machine 간의 평균 latency를 보여준다. Graph (b), (c), 그리고 (d)는 제공된 부하가 증가함에 따라 all-to-all RPC benchmark에 대한 machine당 CPU 시간, prober tail latency, 그리고 background machine antagonist의 latency 영향을 각각 보여준다. Graph (b)와 (c)는 scheduling mode를 비교하는 반면 (d)는 MicroQuanta와 최적화된 Linux CFS를 비교한다. 모든 graph는 Linux TCP와 비교하며, 각 job이 RPC를 요청하고 응답하므로 모든 graph에서 Gbps는 양방향이다.

의 비용이 잘 상각되며 이를 피하는 것이 성능 향상을 보여주지 않는다. Snap/Pony의 Gbps/core는 Table 1에 설명된 82Gbps의 peak single-core throughput보다 적은데, 이는 작업이 여러 engine에 분산되어 batching이 감소하고 이전에 없던 scheduler overhead가 도입되기 때문이다.

Snap compacting scheduler가 최상의 CPU 효율성을 제공하는 반면, Figure 6(c)에 설명된 바와 같이 spreading scheduler가 최상의 tail latency를 가진다. spreading scheduler에서의 낮은 CPU 효율성은 interrupt 및 system context에서 소비된 시간에서 비롯되는 반면, compacting scheduler의 scheduler 시간 대부분은 user context에 있고 전반적으로 더 낮다. 이 실험은 context switching으로 인해 bystander application이 경험하는 성능 손실을 보여주지 않는다. 이 효과는 작업을 가장 적은 core로 통합하고 application 간섭을 최소화하는 compacting scheduler보다 application interrupt에 의존하는 spreading scheduler와 TCP softirq 처리에서 더 높다.

Figure 6(d)는 background antagonist 계산 process로 machine을 로드하고, spreading scheduler를 사용하여 Snap engine을 실행하며, MicroQuanta kernel scheduling class를 nice-ness가 -20인 Linux CFS (가장 공격적인 설정)와 비교하는 영향을 보여준다. background antagonist는 부하 생성 network application job에 비해 감소된 우선순위로 실행되며 MD5 계산을 수행하기 위해 thread를 지속적으로 깨운다. 이들은 hardware (예: DRAM, cache)와 software scheduling system 모두에 엄청난 압력을 가하므로, 특히 non-MicroQuanta 경우

에 antagonist가 없는 Figure 6(c) 실행과 비교하여 전반적인 tail latency를 상당히 증가시킨다.

5.3 System 수준 Interrupt Latency 영향

Figure 6(c)와 (d)가 spreading scheduler에서 유리한 latency 결과를 보여주지만, 이 scheduler는 유후 상태에서 깨어나기 위해 interrupt에 의존하며, 이는 실제로 여러 추가적인 system 수준 latency 기여 요인에 취약하다. 첫째, NIC에서 생성된 interrupt는 deep power-saving C-state에 있는 core를 target으로 할 수 있다. 둘째, 우리의 운영 machine은 MicroQuanta thread의 schedulability에도 영향을 줄 수 있는 복잡한 antagonist를 실행한다. 우리는 Figure 7(a)와 7(b)에서 이러한 효과를 설명하는데, 이전과 동일한 42-machine all-to-all RPC benchmark를 실행하지만 prober job만 실행하고, 1000 QPS (밀리초당 하나의 RPC)에서만 실행하며, application thread wakeup을 transport wakeup으로부터 격리하기 위해 prober application thread를 spin-polling한다.

첫째, Figure 7(a)는 그렇지 않으면 부하가 없는 machine에서 낮은 QPS benchmark를 실행할 때, kernel TCP와 Snap spreading scheduler 모두 C-state interrupt wakeup latency로 인해 6(a)의 이전 two-machine (closed loop) ping-pong latency 결과보다 현저히 나쁜 latency를 보임을 보여준다. Snap compacting scheduler는 가장 압축되고 부하가 가장 적은 상태가 기본적으로 단일 core에서 spin-poll하므로 (필요에 따라 더 많은 core로 확장) 이 wakeup 비용을 피한다.

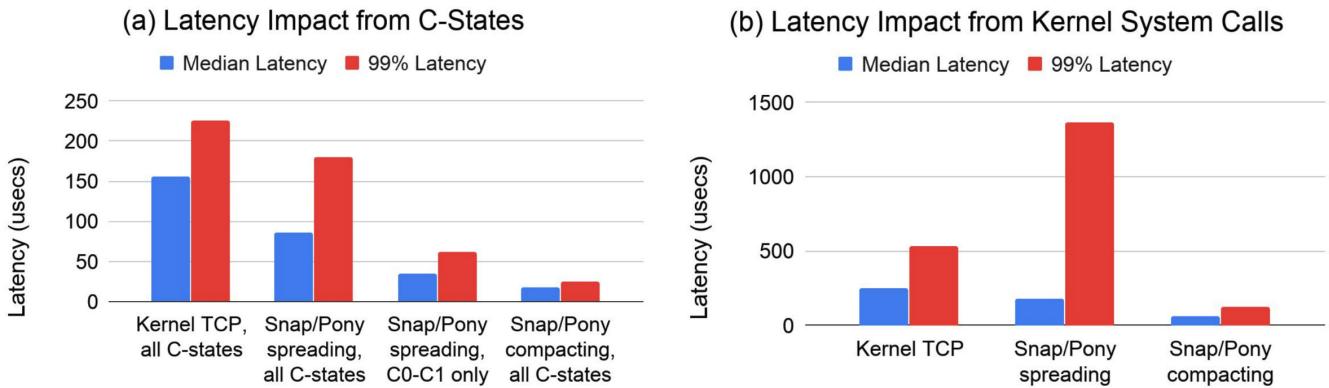


Figure 7: system 수준 효과로 인한 Latency 저하. Graph (a)는 낮은 QPS에서 실행되고 그렇지 않으면 유후 상태인 machine에서 RPC benchmark에 대한 low-power C-state의 latency 영향을 보여주는 반면, graph (b)는 mmap() 및 munmap() system call을 지속적으로 수행하기 위해 thread를 생성하는 가혹한 antagonist의 latency 영향을 보여준다.

둘째, Figure 7(b)는 50MB buffer를 반복적으로 mmap() 및 munmap()하기 위해 thread를 생성하는 가혹한 antagonist를 실행하는 영향을 보여준다. 아마도 흔하지 않고 비현실적인 시나리오이지만, 이는 특정 code 영역이 어떤 userspace process에 의해서도 preempt될 수 없는 많은 Linux kernel에서 발견되는 병리를 보여준다. compacting engine은 이 benchmark에서 engine 작업이 antagonist와 time-share하지 않는 단일 spin-polling core로 압축되기 때문에 최상의 latency를 제공한다. 우리는 Section 6.3에서 CPU 확장에 대한 지속적인 과제를 추가로 논의한다.

5.4 RDMA 및 One-sided Operation

Snap/Pony에서 사용 가능한 one-sided operation을 활용함으로써 application 수준의 상당한 성능 향상을 확인한다. gRPC와 같이 표준 TCP socket에 작성된 기존 RPC stack은 달성 가능한 IOPS/core가 100,000 미만이다 [4]. Figure 8은 one-sided data access를 위해 Snap/Pony를 사용하는 service의 운영 dashboard snapshot을 보여준다. 이 service는 대규모 분산 data analytics system을 지원한다. 이 workload는 단일 Snap/Pony 전용 core로 제공되는 초당 최대 5M의 remote memory access를 요구한다. operation 중 많은 것이 맞춤형 batched indirect read operation을 사용하며, 이에서 Snap/Pony는 해당 indirection이 network를 통해 발생하도록 요구하는 대신 8개의 indirection batch를 로컬에서 수행하기 위해 table을 참조한다.

fabric flow control에 의존하는 hardware RDMA 구현과 비교하여 Snap/Pony로 전환하면 data analytics service의 운영 성능이 두 배가 되었다. 이는 주로 hot-spotting 중 개별 RDMA NIC이 과부하될 때 과도한 fabric back-pressure를 방지하기 위해 이전에 설정된 rate limit의 완화 때문이었다.

Hardware RDMA 구현은 일반적으로 connection 및 RDMA permission 상태의 작은 cache를 구현하며, cache 밖으로 유출되는 access pattern은 상당한 성능 절벽을 초래한다. “thrashing” RDMA NIC은 fabric pause를 방출하며, 이는 다른 switch 및 server로 빠르게 확산될 수 있다. 이로 인해 우리는 machine당 1M RDMA/sec의 상한을 구현했고 credit이 각 client에 정적으로 할당되었다. Snap/Pony로 전환하면 이러한 상한을 제거하고 IOP 속도를 높이며 일시적인 hot-

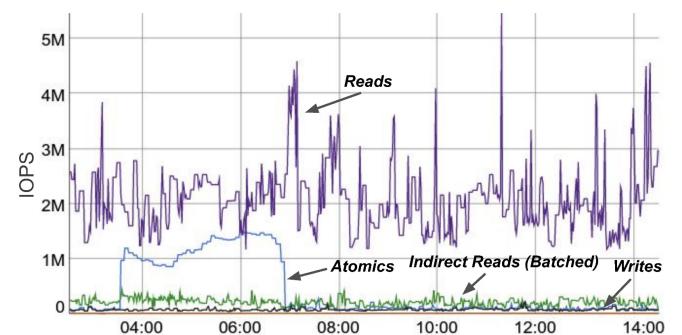


Figure 8: 각 분 간격 동안 가장 뜨거운 machine이 제공한 IOPS 속도를 보여주는 운영 dashboard. 일부 간격은 단일 Snap/Pony engine 및 core가 5M IOPS 이상을 제공함을 보여준다.

spotting을 처리하기 위해 손실이 있는 fabric의 congestion control에 의존할 수 있었다. 그런 다음 통합 batching을 사용하는 맞춤형 indirect read operation으로 전환하면 또 다른 상당한 성능 향상을 얻었다.

5.5 투명한 업그레이드

Figure 9는 2019-01-18에 내부 service를 지원하는 우리 운영 cell 중 하나에서 수행된 투명한 업그레이드에 대한 운영 통계를 보여준다. median blackout duration은 250ms이며, 이는 프로젝트를 시작할 때 설정한 200ms의 목표보다 약간 높다. latency 분포는 heavy-tailed이며 checkpoint된 상태의 양과 강하게 상관관계가 있다. 내부 application 소유자는 주당 한 번 발생하는 끊김을 인지하지 못하지만, 우리는 더 점진적인 전송 단계를 통해 blackout duration을 더욱 줄이는 방법을 탐구하고 있다.

6 경험, 과제, 및 향후 작업

이 section은 Snap 및 관련 networking 기술을 구축하고 배포하는 데 있어 우리의 경험을 논의한다. 또한 진행 중인 작업과 최근 과제를 논의한다.

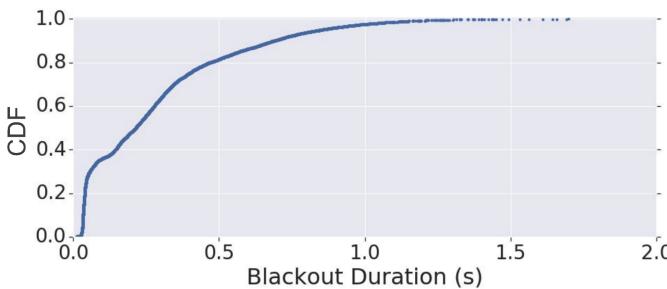


Figure 9: 대규모의 대표적인 운영 cluster로부터의 투명한 업그레이드 blackout duration 데이터.

6.1 In-Application Transport

kernel TCP 또는 Pony Express의 대안은 고성능 use case를 위해 netmap [55] 또는 DPDK [1]와 같은 framework를 사용하거나 QUIC [40]와 같은 낮은 성능 WAN use case를 위해 UDP socket을 사용하여 application process의 address space에서 완전히 작동하는 transport이다. 이러한 in-application transport 설계는 application이 자체 thread context에서 transport code를 실행할 수 있게 하며, 이는 burst에 대응하여 더 빠른 CPU scale-out을 가능하게 하고 system call (kernel TCP) 또는 application thread와 network thread 간의 cross-core hop (Pony Express)을 피함으로써 latency를 향상시킬 수 있다. 실제로 우리는 이 접근 방식을 실험했지만, 이점에도 불구하고 너무 많은 관리 가능성 및 성능 단점을 부과한다는 것을 발견했다.

첫째이자 가장 중요한 것은 Section 4에서 설명한 투명한 업그레이드 접근 방식이 transport를 별도의 process로 실행하는 것에 의존한다는 것이다. 잘 관리되는 Snap의 주간 릴리스를 통해 어디에나 설치된 client library의 모든 wire format을 지원할 필요 없이 Pony Express와 같은 system에 wire protocol 변경을 빠르게 수행할 수 있다. 우리 기법 중 일부는 dynamic linking을 위한 확장이 있는 in-application transport에 적용될 수 있지만, binary 업그레이드는 여전히 많은 개별 service 소유자의 변덕에 달려 있을 것이다.

둘째, datacenter transport는 신중한 CPU 할당 및 scheduling에 민감한 반면, application container에서 실행되는 transport는 복잡한 threading 및 CPU 할당 문제에 직면한다는 것을 발견했다. 모든 application에서 할당된 core를 가진 spin-polling transport thread는 scheduling 예측 불가능성을 완화하는 데 도움이 될 수 있지만, 단일 machine에서 수십 개의 개별 application을 실행하는 것이 일반적이라는 점을 고려하면 이 접근 방식은 일반적으로 실용적이지 않다. 동시에 application transport thread의 interrupt 기반 wakeup에 의존하면 예측 불가능한 scheduling 지연이 발생할 수 있다. 응답하지 않는 transport는 부수적 손상을 일으키는데, 예를 들어 잘 할당된 server가 client에 성공적으로 도달한 packet이 CPU service를 기다리며 멈춰 있기 때문에 과도한 transport timeout 및 불필요한 재전송을 경험할 때 그렇다. interrupt 기반 Snap은 scheduling 지연을 제한하는 데 도움이 되는 MicroQuanta kernel scheduling class를 활용하지만, 이는 특별한 policy 및 중재 없이 임의의 내부 application에서 사용할 수 없는 privileged scheduling class이다.

6.2 Memory Mapping 및 관리

non-kernel networking의 한 가지 중요한 과제는 memory 관리에 있다. 이는 Snap 접근 방식뿐만 아니라 hardware RDMA와 같은 offload stack 및 in-application userspace stack에도 해당된다.

궁극적으로 application은 heap 안팎으로 데이터를 송수신하기를 원한다. Linux kernel은 application이 지정한 virtual memory address로 직접 copy하거나 그로부터 copy할 수 있고, application의 page table을 통해 변환하고 application buffer를 일시적으로 pin하여 zero-copy를 구현할 수도 있지만, Snap은 application page table에 접근할 수 없으며 Linux는 process 간에 임의의 anonymous로 백업된 heap memory를 공유할 수 없다. Linux에서 application과 Snap 간에 공유되는 모든 memory는 tmpfs 또는 memfd로 백업되어야 한다 [6]. 우리는 모든 application heap memory를 tmpfs로 백업하도록 heap 구현을 수정한 다음 Snap과 조정하여 모든 것을 Snap address space (및 page table)에 매핑하는 것을 고려했지만, TLB shootdown에 대한 우려로 인해 이를 구현하지 않았는데, 모든 application address space 수정이 Snap을 실행하는 core를 target으로 하는 inter-processor interrupt를 초래할 수 있기 때문이다. 더욱이 우리의 heap 구현은 phase 동작을 설명하기 위해 memory를 kernel에 자주 반환하며, 이는 빈번한 remapping을 초래할 수 있다.

Hardware RDMA NIC은 OS bypass와 함께 유사한 문제에 직면하는데, 이들도 application page table에 접근할 수 없고 long-lived memory pinning 및 I/O page table에 의존하기 때문이다. 많은 MPI stack이 heap memory를 NIC에 투명하게 등록하는 대체 heap을 제공하지만, 우리 환경의 유동적인 address space 수정은 NIC (또는 IOMMU)과의 과도한 조정을 요구할 수 있다.

Pony Express는 zero-copy transmit를 위해 일부 application 공유 memory를 NIC에 등록한다. 그러나 우리는 이를 선택적으로 수행하며 현재 application 중 일부는 application heap memory에서 Snap과 공유되고 등록된 bounce buffer memory로 copy를 수행한다 (그 반대도 마찬가지). 우리는 예를 들어 memory를 변환하고 일시적으로 pin하는 맞춤형 system call의 비용이 memory copy보다 저렴한 경우 이러한 copy를 피하는 기법을 탐구하고 있다.

6.3 동적 CPU 확장

Snap을 위한 CPU의 scheduling 및 확장은 지속적인 연구 및 개발 영역이다. 이 논문에서 우리는 세 가지 다른 접근 방식을 제시했다: dedicated core, spreading engine, 그리고 compacting engine. 우리의 초기 Snap 배포는 부분적으로 단순성 때문이지만 또한 정적 할당이 알려진 부하 임계값까지 성능 SLO를 충족하기 때문에 dedicated core를 사용했으며, 따라서 많은 배포가 오늘날에도 dedicated core를 계속 사용한다. 그러나 특히 Pony Express를 범용 TCP/IP 대체 품으로 사용하여 Snap의 사용을 확대하려고 함에 따라 host networking CPU 자원의 동적 할당이 가장 중요하게 되었다. 우리는 kernel TCP networking 사용이 매우 burst하며 때때로 짧은 burst 동안 수십 개의 core를 소비한다는 것을 발견한다. 우리의 성능 평가는 compacting engine scheduler가 최상의 전반적인 CPU 효율성을 제공하고 낮은 부하에서 그리고 system call antagonist가 있을 때 가장 낮은 latency를 제공함을 보여주었지만, engine queue 축적에 대응하는 지연

때문에 spreading engine이 더 높은 Snap 부하에서 더 나은 tail latency를 제공할 수 있음도 보여주었다. 우리는 현재 두 유형의 scheduler의 요소를 결합하고 kernel 지원을 계속 개선함으로써 두 가지 장점을 모두 달성하기 위해 노력하고 있다.

6.4 Engine 간 재균형화

Section 2.2에서 논의한 바와 같이 engine은 single-threaded이며, 이는 Snap 개발자를 fine-grained synchronization의 overhead로부터 멀어지게 하고 가능한 한 상태를 격리하도록 유도한다. 예를 들어 Pony Express engine은 현재 engine 간에 flow 상태를 공유하지 않으며, flow도 현재 engine 간에 재균형화되지 않는다. 그러나 결과적으로 engine 내의 flow 간에 flow 처리를 직렬화하면 성능이 제한될 수 있다. 우리의 client library는 engine 간에 flow를 단순히 sharding하는 것에서부터 flow를 복제하고 Pony Express layer 위에서 재균형화를 구현하는 것까지 여러 engine에 걸쳐 부하를 분산시키는 다양한 메커니즘을 구현한다. 그럼에도 불구하고 우리는 work stealing [48], [51]과 같은 메커니즘을 사용하여 engine 간의 fine-grained 재균형화의 필요성을 고려하고 있다.

6.5 Stateful Offload

hardware에서 완전히 구현되고 조정되지 않은 application library에 의해 직접 접근되는 OS-bypass RDMA stack에 대한 우리의 초기 경험은 확장되지 않았고 필요한 반복 속도를 갖지 못했다. 그럼에도 불구하고 Moore's Law가 느려짐에 따라 우리는 datacenter 채택을 위한 장애물을 극복하는 차세대 stateful offload를 계속 탐구한다. 앞으로 우리는 향후 hardware offload에 접근하기 위한 중앙집중식 관리 layer로 Snap을 활용하는 데 여러 이점을 본다. 첫째, 복잡한 stateful NIC에서 자주 발생하는 hardware 및 firmware bug에 대한 workaround를 배포할 수 있게 한다. 둘째, multiplexing 및 관리 정책을 제공하는 OS의 전통적인 역할을 유연하게 구현할 수 있다. 셋째, 복잡한 hardware offload를 위한 견고하고 안전한 virtual function을 제공하는 것은 어려우며 Snap은 필요할 때 투명한 대체 메커니즘을 제공할 수 있다. 마지막으로 application을 특정 hardware로부터 분리하면 기본 hardware와 무관하게 machine 간에 application을 투명하게 마이그레이션할 수 있다.

7 관련 연구

별도의 host service로서의 networking 분야에서 TAS [35], IsoStack [57], 그리고 SoftNIC [26]은 최소화, spin-polling, 그리고 data structure 공유 제거를 통한 성능에 초점을 맞춰 효율성과 낮은 지연시간을 위해 core를 전용화한다. 동시에 library OS 솔루션 분야에서 IX [14], [52], ZygOS [51], 그리고 Shinjuku [31]은 단일 address space에서 단일 application을 처리하는 고성능 system을 개발하며, application 처리 handler를 transport 자체에 통합하고 전용 NIC 지원을 가정한다. ZygOS는 intermediate buffering을 사용한 task stealing을 통합하여 IX와 차별화되고, Shinjuku는 intelligent preemption을 추가한다. 그러나 세 가지 노력 모두 inter-application 공유 및 중재 문제를 회피하며 Dune virtualization [13]을 요구한다. 위의 노력 중 어떤 것도 fine-grained 동적 CPU 확장, 투명한 업그레이드를 통한 효율성과 낮은 지연시간을 다루거나,

Snap이 범용 multi-application 운영 및/또는 cloud 환경에서 제공하는 것의 완전한 일반성을 달성하지 못한다.

다른 최근 접근 방식은 academia [30], [32], [42], [48], [50], [53] 내외 [1], [55]에서 모두 고성능을 달성하기 위해 경량 runtime과 kernel bypass를 활용한다. 아마도 우리의 작업과 가장 유사한 것은 Shenango [48]로, 전체 core를 thread scheduling 및 packet steering 기능에 전용함으로써 우리 작업보다 더 공격적으로 동적 CPU 확장을 다룬다. 그러나 Shenango는 맞춤형 application scheduling runtime을 요구하고 core를 operating system으로부터 분할할 것을 요구한다. Snap은 MicroQuanta kernel scheduling class를 통해 범용 host와 독특하게 통합되며 또한 투명한 업그레이드를 독특하게 다룬다.

transport 측면에서 혼합 onload/offload 접근 방식이 이전에 TCP [54], [56]에 대해 제안되었다. datacenter transport에 대한 풍부한 최근 연구도 있지만 [11], [17], [24], [27], [46], [49], 그 대부분은 이 작업의 범위가 아닌 fabric 수정을 요구한다. RDMA offload 없이 machine에서 실행되는 Pony Express와 마찬가지로 FaSST [34]는 two-sided datagram 교환을 사용하여 RDMA를 구현하며, Pony Express와 마찬가지로 LITE [60]는 software indirection layer를 통해 RDMA를 노출하지만 userspace가 아닌 kernel에서 구현된다.

kernel 설계 측면에서 FlexSC [58]는 현대 multicore hardware를 활용하고 system call을 application thread로부터 별도의 thread에 위임한다는 점에서 Snap과 유사하지만, 이를 수행하는 데 microkernel과 유사한 접근 방식을 제안하지 않는다. 동시에 microkernel은 수십 년의 역사를 가지고 있다 [10], [15], [16], [20], [23], [28], [29], [41], [61]. 가장 관련성이 높은 것은 Thekkath et al. [59]의 작업으로, monolithic kernel에서 network protocol을 구현하는 것과 microkernel 접근 방식 간의 trade-off를 논의한다. 그들은 궁극적으로 domain switching overhead를 언급하며 후자를 기각한다. 그러나 이전에 논의한 바와 같이 이러한 overhead는 multi-core system 및 현대 server architecture의 fine-grained memory 공유 메커니즘으로 상당히 감소했다.

8 결론

우리는 널리 배포된 microkernel에서 영감을 받은 host networking system인 Snap의 설계와 architecture를 제시한다. Snap은 수많은 team이 높은 개발자 및 릴리스 속도로 고성능 packet 처리 system을 개발할 수 있게 한다. 우리는 이 microkernel 접근 방식을 기반으로 한 통신 stack과 실행 중인 machine에서 application을 비우지 않고 통신 stack을 투명하게 업그레이드하는 방법을 자세히 설명한다. 우리의 평가는 Gbps/core 효율성의 최대 3배 향상, sub-10-microsecond latency, 동적 확장 capability, 초당 수백만 개의 one-sided operation, 그리고 효율성과 latency 간의 맞춤형 강조를 가진 CPU scheduling을 보여준다. Snap은 여러 중요한 system의 확장 가능한 통신 요구사항을 지원하며 3년 동안 운영 환경에서 실행되어 왔다. 우리의 경험은 Snap 릴리스 속도가 운영 환경에서 필요한 반복 속도 때문에 이러한 service를 배포하기 위한 본질적인 요구사항이었음을 나타낸다. 성능 향상은 부수적인 이점으로, 그렇지 않으면 불가능했을 방식으로 고급 one-sided 및 low-latency messaging 기능을 활용하는 새로운 application 구조의 공격적인 배포로 가능해졌다.

Acknowledgements

We would like to thank our reviewers, shepherd Irene Zhang, Jeff Mogul, Thomas Wenisch, Aditya Akella, Mark Hill, Neal Cardwell, and Christos Kozyrakis for helpful paper feedback; Luiz Barroso and Sean Quinlan for encouragement throughout; Joel Scherplez and Emily Blem for some of the earliest exploratory work on the project; We also thank Bill Vareka, Larry Greenfield, Matt Reid, Steven Rhodes, Anish Shankar, Alden King, Yaogong Wang, and the large supporting cast at Google who helped realize this project.

References

- [1] Data plane development kit. <http://www.dpdk.org>.
- [2] Fast memcpy with SPDK and intel I/OAT DMA engine. <https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine>.
- [3] Github repository: Neper linux networking performance tool. <https://github.com/google/neper>.
- [4] grpc benchmarking. <https://grpc.io/docs/guides/benchmarking>.
- [5] Linux CFS scheduler. <https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt>.
- [6] memfd manpage. http://man7.org/linux/man-pages/man2/memfd_create.2.
- [7] Nice levels in the linux scheduler. <https://www.kernel.org/doc/Documentation/scheduler/sched-nice-design.txt>.
- [8] Scaling in the linux networking stack. <https://www.kernel.org/doc/Documentation/networking/scaling.txt>.
- [9] Short waits with unwait. <https://lwn.net/Articles/790920/>.
- [10] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In Proceedings of the USENIX Summer Conference, Atlanta, GA, USA, June 1986, pages 93–113, 1986.
- [11] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.
- [12] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.
- [13] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.
- [14] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.
- [15] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. ACM Trans. Comput. Syst., 8(1):37–55, Feb. 1990.
- [16] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, SOSP '93, pages 120–133, New York, NY, USA, 1993. ACM.
- [17] L. Chen, K. Chen, W. Bai, and M. Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, pages 174–187, New York, NY, USA, 2016. ACM.
- [18] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.
- [19] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCaboeter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 373–387, Renton, WA, 2018. USENIX Association.
- [20] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, SOSP '91, pages 122–136, New York, NY, USA, 1991. ACM.
- [21] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation, OSDI '96, pages 261–275, New York, NY, USA, 1996. ACM.
- [22] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation, NSDI'16, pages 523–535, Berkeley, CA, USA, 2016. USENIX Association.
- [23] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.
- [24] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '15, pages 1:1–1:12, New York, NY, USA, 2015. ACM.
- [25] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is dead: Long live KASLR. In Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Proceedings, volume 10379 LNCS of Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 161–176, Italy, 2017. Springer-Verlag Italia.
- [26] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.
- [27] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, pages 29–42, New York, NY, USA, 2017. ACM.
- [28] P. B. Hansen. The nucleus of a multiprogramming system. Commun. ACM, 13(4):238–241, Apr. 1970.
- [29] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ -kernel-based systems. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM.
- [30] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 489–502, Seattle, WA, 2014. USENIX Association.
- [31] K. Kaffles, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 345–360, Boston, MA, 2019. USENIX Association.
- [32] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26–28, 2019., pages 1–16, 2019.
- [33] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.
- [34] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In Proceedings of the 12th USENIX Conference on Operating Systems

- Design and Implementation, OSDI'16, pages 185–201, Berkeley, CA, USA, 2016. USENIX Association.
- [35] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP acceleration as an OS service. In Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19, pages 24:1–24:16, New York, NY, USA, 2019. ACM.
- [36] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella. Iron: Isolating network-based CPU in container environments. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 313–328, Renton, WA, 2018. USENIX Association.
- [37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In 40th IEEE Symposium on Security and Privacy (S&P'19), 2019.
- [38] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 627–643, Carlsbad, CA, Oct. 2018. USENIX Association.
- [39] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, pages 1–14, New York, NY, USA, 2015. ACM.
- [40] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasilev, W.-T. Chang, and Z. Shi. The QUIC transport protocol: Design and internet-scale deployment. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, pages 183–196, New York, NY, USA, 2017. ACM.
- [41] J. Liedtke. Improving IPC by kernel design. In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.
- [42] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association.
- [43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In 27th USENIX Security Symposium (USENIX Security 18), 2018.
- [44] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen. SkyBridge: Fast and secure inter-process communication for microkernels. In Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19, pages 9:1–9:15, New York, NY, USA, 2019. ACM.
- [45] R. Mittal, T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. Timely: RTT-based congestion control for the datacenter. In Sigcomm '15, 2015.
- [46] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. CoRR, abs/1803.09615, 2018.
- [47] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP '99, pages 217–231, New York, NY, USA, 1999. ACM.
- [48] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 361–378, Boston, MA, 2019. USENIX Association.
- [49] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized “zero-queue” datacenter network. In Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, pages 307–318, New York, NY, USA, 2014. ACM.
- [50] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. pages 1–16, 2014.
- [51] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM.
- [52] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27–29, 2015, pages 342–355, 2015.
- [53] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: Core-aware thread management. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, pages 145–160, Berkeley, CA, USA, 2018. USENIX Association.
- [54] G. J. Regnier, S. Makineni, R. Illikkal, R. R. Iyer, D. B. Minturn, R. Huggahalli, D. Newell, L. S. Cline, and A. P. Foong. TCP offloading for data center servers. IEEE Computer, 37(11):48–58, 2004.
- [55] L. Rizzo. Netmap: A novel framework for fast packet I/O. In Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.
- [56] L. Shalev, V. Makhervaks, Z. Machulsky, G. Biran, J. Satran, M. Ben-Yehuda, and I. Shimony. Loosely coupled TCP acceleration architecture. In Hot Interconnects, pages 3–8. IEEE Computer Society, 2006.
- [57] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: Highly efficient network processing on dedicated cores. In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.
- [58] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, pages 33–46, Berkeley, CA, USA, 2010. USENIX Association.
- [59] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. IEEE/ACM Trans. Netw., 1(5):554–565, Oct. 1993.
- [60] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 306–324, New York, NY, USA, 2017. ACM.
- [61] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. Commun. ACM, 17(6):337–345, June 1974.
- [62] K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. 2017.