UNIVERSIDADE FEDERAL DO PAMPA

Ígor Ferrazza Capeletti

Análise de Desempenho de Aplicações eBPF/XDP em Planos de Dados Programáveis

Alegrete
2022

Ígor Ferrazza Capeletti

Análise de Desempenho de Aplicações eBPF/XDP em Planos de Dados Programáveis

Alegrete
2022

RESUMO

O Extended Berkeley Packet Filter (eBPF) é um subsistema do Núcleo do Sistema Operacional (kernel) que filtra os pacotes de rede nos dispositivos do plano de dados com o auxílio do *Hook* Caminho de Dados Expresso (XDP). O *Hook* XDP permite que programas eBPF realizem o processamento dos pacotes de rede no espaço do usuário, no espaço do kernel, no driver das placas e também no hardware das placas de rede. Apesar das iniciativas de avaliar o desempenho de programas eBPF, as análises existentes ainda não avaliam a execução de diversos programas eBPF processando diferentes tamanhos de pacotes para as todas as abordagens existentes do *Hook* XDP. Este trabalho tem o propósito de analisar as capacidades e limitações que os programas eBPF atingem processando pacotes em diferentes abordagens do *Hook* XDP com SmartNICs, além de servir como objeto de estudo para a área. Portanto realizamos avaliações de desempenho como Latência, Taxa de Transferência e uso de CPU para as execuções de diversos programas eBPF em SmartNICs com suporte à todos os *Hook* XDP. Em nosso trabalho, estudamos o paradigma Rede Definida por Software (SDN) e o plano de dados programável com eBPF/XDP. Em seguida, realizamos o levantamento dos principais trabalhos relacionados com eBPF/XDP e apresentamos a metodologia utilizada para realização dos experimentos e avaliações. Nossos resultados mostraram que todos os modos XDP avaliados conseguem excelentes taxas de transferência ao processar grandes pacotes de rede. Para a métrica de Latência nossos resultados mostraram que em todos os modos XDP e para a maioria dos tamanhos de pacotes, ao aumentarmos a quantidade de acessos à memória, o tempo de Latência também aumenta. Quanto ao uso de CPU, todos os modos XDP tiveram baixas taxas de uso dos núcleos ao utilizar mais filas TX/RX para processar tráfegos com grandes pacotes de rede. Nas avaliações de número de instruções, número de branches, número de load hits e load misses, os resultados mostraram que não ocorreram diferenças de comportamento entre os diferentes experimentos.

Palavras-chave: Processamento de Pacotes com *eBPF/XDP*, Hook *XDP*, *Software-Defined Network(SDN)*, Plano de Dados Programável

LISTA DE FIGURAS

LISTA DE SÍMBOLOS

API Interface de Programação de Aplicativos

cBPF Berkeley Packet Filter

DDoS Negação de Serviço Distribuída

eBPF Extended Berkeley Packet Filter

FPGA Field Programmable Gate Array

IoT Internet das Coisas

kernel Núcleo do Sistema Operacional

NFV Virtualização de Funções de Rede

NIC Placa de Interface de Rede

NOS Sistemas Operacionais de Rede

P4 Programming Protocol-independent Packet Processors

SDN Rede Definida por Software

VM Máquina Virtual

XDP Caminho de Dados Expresso

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto e Motivação

A programabilidade do plano de dados (PDP) tem redesenhado o domínio de aplicações na área de Redes de Computadores. Essa programabilidade permite (re)definir o comportamento dos dispositivos de rede que processam pacotes de forma simples através de linguagens de domínio-específico (por exemplo, P4(BOSSHART et al., 2014a)). Isto permite desenvolver mecanismos especializados de processamento de pacotes para equipamentos com suporte a tal programabilidade, tais como roteadores programáveis, SmartNICs e para o próprio kernel. Nos últimos anos, a programabilidade do plano de dados tem permitido realizar a migração de aplicações tipicamente executadas no plano de controle para serem executadas no plano de dados. Exemplos incluem algoritmos de aprendizado de máquina (XIONG; ZILBERMAN, 2019), roteamento (Pizzutti; Schaeffer-Filho, 2019) ou monitoramento de rede (Castro et al., 2021). Ao deslocar a operação dessas aplicações para o plano de dados, tem-se o benefício direto de se processar cada pacote que transita na infraestrutura e de reagir às condições da rede na ordem de nanossegundos, com intervenção mínima no plano de controle. Apesar desses benefícios, a operação de aplicações mais complexas diretamente no plano de dados pode afetar diretamente o desempenho dos dispositivos de encaminhamento – isto é *menor* Taxa de Transferência e *maior* Latência.

Aplicações mais complexas executadas diretamente no plano de dados (por exemplo, de aprendizado de máquina) tendem a executar um maior número de instruções a cada pacote processado. Similarmente, o número de acesso às memórias disponíveis em dispositivos programáveis também pode aumentar (ROSSI et al., 2021). Para além dessas operações elementares, aplicações desenvolvidas para o plano de dados podem aplicar operações específicas deste domínio. Um exemplo é a recirculação de pacotes. Esta primitiva permite que o plano de dados processe o mesmo pacote inúmeras vezes ao reenviar o mesmo pacote para uma determinada fila de entrada do dispositivo. Este tipo de procedimento é utilizado em aplicações convencionais para duplicar pacotes, por exemplo. Em aplicações modernas, esse tipo de primitiva pode ser utilizada para imitar um mecanismo baseado em *loop*, já que no geral tais dispositivos não possuem (ou não permitem) estruturas iterativas. Esses são exemplos diretos de operações simples comumente usadas por aplicações complexas como, por exemplo, aplicações de agrupamento/*clustering* de dados executados diretamente na rede (XIONG; ZILBERMAN, 2019).

Para além dos dispositivos de rede programáveis comumente utilizados para executar aplicações de rede (isto é, roteadores programáveis e SmartNICs), há uma crescente adoção da comunidade científica e da indústria de tecnologias em *software* que permitam executar aplicações de rede com alto desempenho (isto é, em *line-rate*[1]). Exemplos de

[1] O termo *line-rate* se refere ao processamento de pacotes na taxa máxima de transferência do enlace.

tecnologias que amadureceram ao longo dos últimos anos e permitem o processamento rápido de pacotes pelo kernel incluem os mecanismos de filtragem e processamento de pacotes eBPF (*Extended Berkeley Packet Filter*) e XDP (*eXpress Data Path*).

O primeiro mecanismo de filtragem de pacotes em sistemas *Unix* foi implementado no espaço do usuário. Os pacotes eram transferidos do espaço do kernel para o espaço do usuário para serem filtrados e processados. Após esta abordagem de filtro de pacotes se mostrar ineficiente em termos de Vazão e Latência, o subsistema Berkeley Packet Filter (cBPF) foi introduzido no espaço do kernel como uma solução para filtrar os pacotes diretamente no kernel o mais cedo possível e evitar cópias desnecessárias para o espaço do usuário. O cBPF consiste de uma linguagem *assembly* usada para executar filtragem de pacotes em um máquina virtual no caminho de dados da pilha de protocolos TCP/IP dentro do próprio kernel (MCCANNE; JACOBSON, 1993). As aplicações BPF possuem um conjunto de instruções flexíveis e independentes de protocolos e, desta forma, permitem aos programadores desenvolverem e a aplicarem em diversos casos de uso. Além disso, as aplicações desenvolvidas não precisam que os módulos do kernel sejam reescritos ou recompilados já que as instruções BPF são executadas diretamente em uma máquina virtual minimalista dentro do próprio kernel.

Para aumentar os casos de uso e atualizar a arquitetura de acordo com os avanços nos processadores modernos, o *Extended Berkeley Packet Filter* (eBPF) (MCCANNE; JACOBSON, 1993) foi proposto como uma evolução do cBPF. No eBPF, a máquina virtual BPF foi reescrita com adição de diferentes componentes, instruções e funcionalidades. O eBPF consiste em um *assembly* mais abrangente e permite que os programas carregados façam uso de memória (ou mapas) para armazenar pares de chave/valor, além de uma pilha e registradores de 64 bits. Para obter maior desempenho, os pacotes de rede precisam ser filtrados nas camadas mais inferiores do kernel – isto é, mais próximas do *driver* de rede. Desta forma, surgiu recentemente o *eXpress Data Path* (XDP), uma extensão do eBPF que permite o processamento de pacotes logo que o pacote é recebido pela interface de rede. Isto permite que os pacotes de rede sejam processados pelo CPU na camada mais inferior do kernel ou também descarregados (*offloading*) para a uma interface de rede com suporte a tais instruções (por exemplo, para uma SmartNIC).

1.2 Problema de Pesquisa

Embora existam estudos recentes que tenham realizado a avaliação do impacto de diferentes aplicações executadas no plano de dados em SmartNICs (HARKOUS et al., 2019; ROSSI et al., 2021), a análise de desempenho e eficiência de programas eBPF e XDP ainda é um tópico de pesquisa a ser investigado. A análise de programas eBPF e XDP permitirá determinar os limites reais de desempenho (por exemplo, Vazão e Latência)

Por exemplo, em um enlace de 10Gbit/s, atinge-se o line-rate quando é possível transferir pacotes de qualquer tamanho nesta taxa.

e fornecer informações sobre o processamento de pacotes em software. O entendimento dos limites alcançáveis por aplicações de rede executadas em software é importante para (i) auxiliar na portabilidade de aplicações de rede (isto é, determinar que aplicações são adequadas para serem executadas em software) e (ii) auxiliar na divisão de aplicações monolíticas de rede para serem executadas de maneira distribuída em múltiplas plataformas/arquiteturas.

1.3 Objetivos e Contribuições

Neste trabalho, objetiva-se *entender e quantificar o desempenho de aplicações genéricas de rede baseadas em eBPF e XDP quanto a métricas de desempenho de rede.* Desta forma, este objetivo se desdobra em:

1. Identificar os principais blocos construtores de uma aplicação eBPF e XDP de modo a reproduzir tais comportamentos.
2. Automatizar a geração de códigos de aplicações genéricas baseadas em eBPF e XDP.
3. Avaliar o desempenho das aplicações geradas quanto a Taxa de Transferência, Latência e uso do CPU.

1.4 Estrutura

No Capítulo 2, apresenta-se uma visão geral da literatura atual sobre Redes Definidas por *Software*, assim como sobre o processamento e encaminhamento de pacotes no plano de dados programável com eBPF e XDP. Posteriormente, no mesmo capítulo, discute-se os esforços recentes de pesquisa para o processamento de pacotes utilizando eBPF e XDP. No Capítulo 3, apresenta-se a metodologia experimental utilizada para a condução deste trabalho, enquanto que no Capítulo 4 discute-se os resultados obtidos. Por fim, no Capítulo 5, apresentam-se as considerações finais e perspectivas de trabalhos futuros.

2 FUNDAMENTAÇÃO E TRABALHOS RELACIONADOS

Neste capítulo, primeiramente apresenta-se uma revisão acerca dos recentes avanços no tratamento de pacotes de aplicações de rede, abordando aspectos do paradigma de Redes Definidas por *Software* (*Software-Defined Networking* – SDN) com ênfase na programabilidade do encaminhamento de pacotes no plano de dados com eBPF e XDP. Em seguida, discutimos os principais trabalhos de pesquisa na área.

2.1 Redes Definidas por Software

As redes IP tradicionais são amplamente adotadas e cada vez mais serviços estão sendo oferecidos pelas operadoras de rede. Desta forma, as infraestruturas de redes IP estão se tornando cada vez mais complexas e difíceis de serem gerenciadas. Como consequência, configurações de rede incorretas e erros humanos são muito comuns. Por exemplo, um equipamento da infraestrutura mal configurado pode gerar uma série de problemas na correta operação da rede (por exemplo, indisponibilidade de serviços). Conforme (KREUTZ et al., 2015), os ambientes de rede precisam suportar (re)configuração, ou seja, adaptar-se às mudanças de carga de trabalho e tratar falhas dinamicamente. Uma limitação atual dos dispositivos tradicionais é a integração entre os planos de controle e de dados. O plano de controle é responsável por decidir as ações de encaminhamento do tráfego de rede, enquanto que o plano de dados é responsável por apenas encaminhar o tráfego a partir das decisões tomadas pelo plano de controle. Essa integração vertical reduz a flexibilidade e torna cada vez mais difícil o desenvolvimento e implantação de novos recursos de forma independente.

Para adicionar novos recursos, funcionalidades e serviços nas infraestruturas de rede, além da necessidade de comprar hardwares dedicados, especializados e com custo elevado, como por exemplo *Middleboxes*, as operadoras precisam configurar os dispositivos individualmente através de linhas de comando (isto é, CLI) específicas, além de integrar uma série de *softwares* de gerenciamento de cada fornecedor. Além disso, os *Middleboxes* precisam ser implantados em locais estratégicos na rede, o que torna ainda mais difícil modificações na topologia, configurações e funcionalidades. Os custos para implementação e manutenção das redes tradicionais são altos e o retorno financeiro é no geral lento, dificultando a inovação e evolução dos serviços.

O conceito de SDN surgiu como um paradigma de rede emergente com potencial para solucionar os problemas acima mencionados e as limitações das infra-estruturas de rede existentes. De uma maneira geral, SDN é baseado nos princípios de redes ativas/programáveis e da própria proposição da separação entre os planos de controle e de dados. Para tornar as redes programáveis, o embasamento se deu através de redes ATM programáveis (LAZAR; LIM; MARCONCINI, 1996), (LAZAR, 1997) e redes ativas (TENNENHOUSE et al., 1997). Para separar os planos de controle e dados, os estudos partiram da plataforma de controle de roteamento (BCP) (CAESAR et al., 2005)

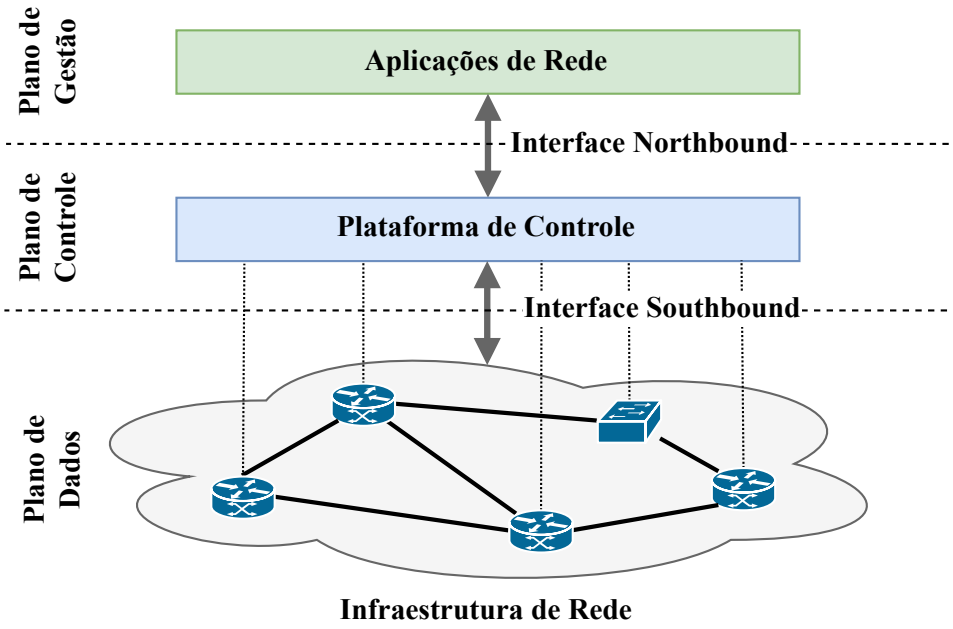

Figura 1 – Visão geral da arquitetura SDN.

e ponto de controle de rede (NCP) (SHEINBEIN; WEBER, 1982). Em (FEAMSTER; REXFORD; ZEGURA, 2014) são abordadas informações adicionais sobre o histórico de SDN e redes programáveis.

O princípio básico de SDN é a separação entre os planos de controle e dados – conforme ilustrado na Figura 1. Desta forma, é possível construir aplicações no plano de controle independentes que possam aplicar políticas de encaminhamento, de (re)configuração, ou de migração de serviços de forma independente. Em infraestruturas baseadas em SDN, os dispositivos de encaminhamento (isto é, o plano de dados) apenas realizam o encaminhamentos. A lógica de controle, isto é, implementada no plano de controle, é executada em controlador logicamente centralizado, mas fisicamente distribuído. Essa separação dos planos de controle e dados somente é viável através de uma Interface de Programação de Aplicativos (API) bem definida entre o controlador SDN e os dispositivos de encaminhamento.

Por meio desta API, o controlador consegue gerenciar diretamente os estados dos elementos no plano de dados. Uma API relevante é o protocolo OpenFlow (MCKEOWN et al., 2008). O protocolo OpenFlow permite que aplicações no plano de controle gerenciem e controlem o funcionamento de dispositivos de encaminhamento que suportam tal protocolo. Um dispositivo de encaminhamento com suporte ao protocolo OpenFlow possui tabelas de encaminhamento com regras para o tratamento de pacotes (tabela de fluxo). Cada uma dessas regras executa determinadas ações sobre um subconjunto dos pacotes do tráfego de rede. Quando instruído por um controlador, um dispositivo de encaminhamento OpenFlow pode comportar-se como um *Firewall*, um *Switch*, um roteador ou até mesmo outras funções mais específicas determinadas pelas aplicação situada no

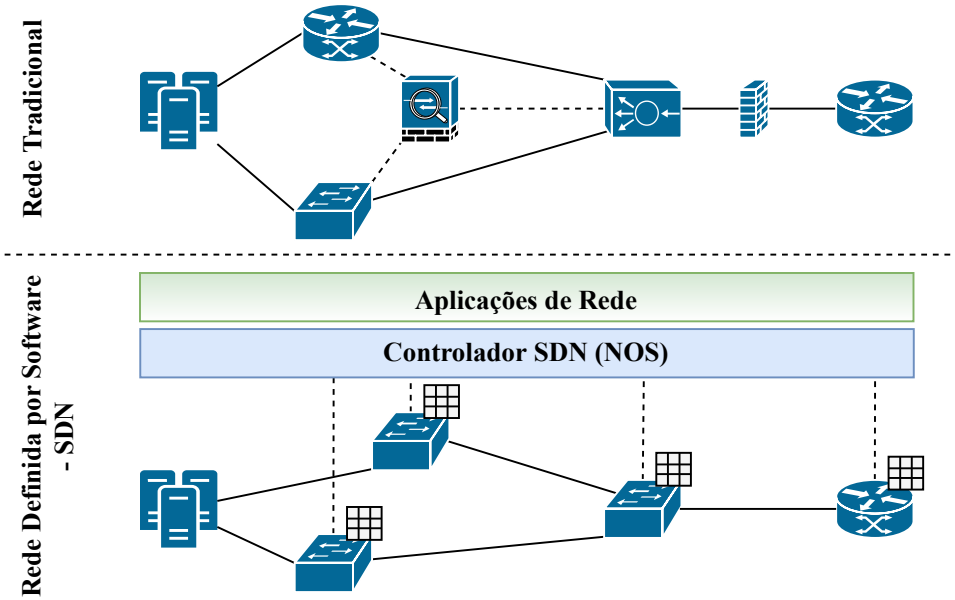

Figura 2 – Rede Tradicional versus Rede Definida por Software (SDN).

controlador. OpenFlow e SDN surgiram como experimentos acadêmicos (MCKEOWN et al., 2008) e atualmente a maioria dos fornecedores de dispositivos de encaminhamento comerciais oferecem suporte para o protocolo.

SDN é definido em (KREUTZ et al., 2015) como uma arquitetura de rede de quatro pilares. No primeiro pilar, os planos de controle e dados são desacoplados, ou seja, as funcionalidades de controle dos dispositivos de rede serão removidas, tornando-os apenas elementos de encaminhamento. Como segundo pilar, as decisões dos encaminhamentos não são mais baseadas no destino, mas sim no fluxo para permitir maior flexibilidade no encaminhamento. Como o plano de controle é desacoplado dos dispositivos de rede (de acordo com o primeiro pilar), no terceiro pilar, o plano de controle torna-se uma entidade externa como o controlador SDN (Sistemas Operacionais de Rede (NOS)). Esse desacoplamento possibilita várias vantagens como: (i) fornece recursos e abstrações essenciais para facilitar a programação dos dispositivos de encaminhamento, (ii) permite executar ações de forma flexibilizada em qualquer lugar da infraestrutura de rede, (iii) integra diferentes aplicações de forma simplificada e, (iv) fornece uma visão global das informações da rede para tomar decisões mais eficazes e consistentes. No quarto pilar, sendo o principal princípio do SDN, a rede é programável por softwares executados no controlador SDN NOS que interagem com os dispositivos do plano de dados subjacentes. A Figura 2 ilustra a diferença entre redes IP tradicionais e redes baseadas em SDN, além das interações entre o plano de controle e o plano de dados. Com o desacoplamento dos planos de controle e dados, percebe-se que SDN possibilita maior programabilidade na rede, tornando o gerenciamento mais simples e os *Middleboxes* podem ser desenvolvidos como aplicações no plano de controle SDN.

Para além dos benefícios já citados, a centralização lógica das decisões de controle

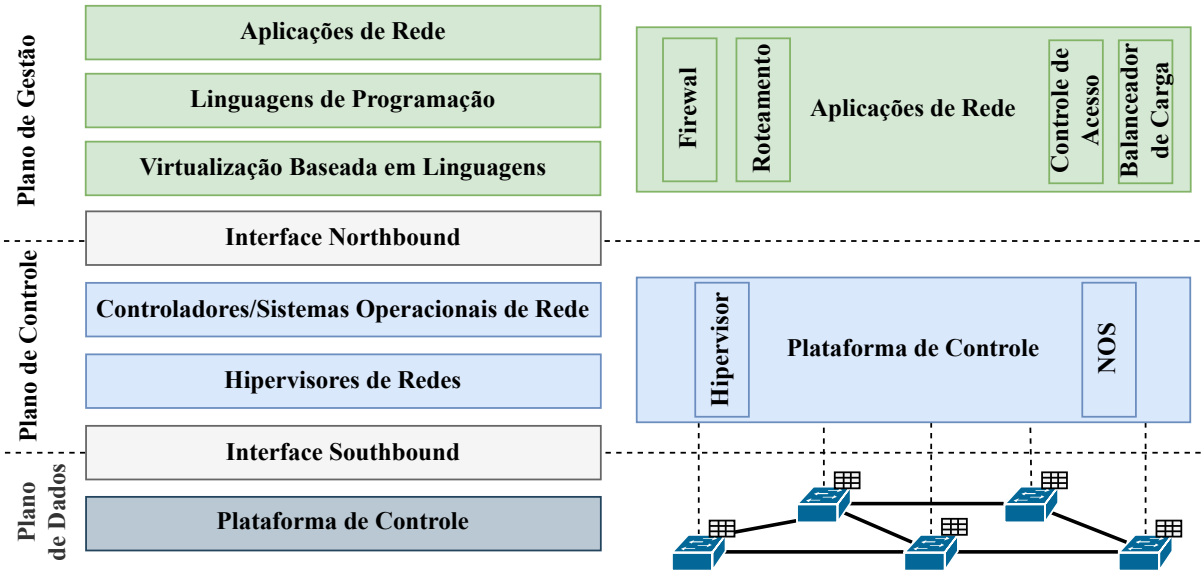

Figura 3 – Visão geral das Camadas SDN.

em um controlador com conhecimento global possibilita: (i) diminuição de erros que possam modificar as políticas de rede por meio de softwares de alto nível, (ii) simplicidade no desenvolvimento de funções, serviços e aplicativos de rede, e (iii) para preservar as políticas de alto nível, o programa de controle pode automaticamente reagir as mudanças fraudulentas no estado da rede. De acordo com (SCHENKER, 2011), uma rede SDN ainda pode ser definida por três principais abstrações: encaminhamento, distribuição e especificação. A abstração de encaminhamento, enquanto oculta detalhes de hardwares subjacentes, precisa permitir qualquer comportamento de encaminhamento desejado pelos aplicativos de rede. Como um exemplo desta abstração, temos o próprio protocolo OpenFlow.

Na abstração por distribuição, as aplicações SDN devem ser protegidos do estado distribuído para que o problema de controle seja logicamente centralizado. Localizando-se dentro do controlador SDN NOS, essa abstração requer uma camada de distribuição comum, responsável por instalar os comandos de controle nos dispositivos de encaminhamento e também coletar informações sobre a camada de encaminhamento, oferecendo uma visão global da rede para os aplicativos de controle. A partir de linguagens de programação de rede e soluções de virtualização, a abstração de especificação permite que os aplicativos de rede expressem seus comportamentos desejados sem a necessidade de implementá-los.

2.1.1 As Camadas da Arquitetura SDN

Conforme ilustrado na Figura 3, SDN pode ser composta por oito camadas diferentes, cada uma possuindo ações e funcionalidades específicas. A seguir, discute-se as funcionalidades de cada camada.

2.1.1.1 Camada de Infraestrutura

A Camada de Infraestrutura possui o mesmo conjunto de dispositivos de rede tradicionais (roteadores, *switches* e *middleboxes*). Como em redes SDN a inteligência dos dispositivos (plano de controle) é movido para um sistema de controle NOS, esses dispositivos realizam apenas o encaminhamento de pacotes sem nenhum software com decisão autônoma. As redes também são construídas em interfaces abertas (por exemplo OpenFlow) permitindo a programabilidade dinâmica desses dispositivos de encaminhamento heterogêneos (por exemplo, de diferentes fabricantes). Portanto, na arquitetura SDN/OpenFlow, um dispositivo do plano de dados é um software ou hardware especializado no encaminhamento de pacotes. Quando habilitados com OpenFlow, esses dispositivos são baseados em pipeline com tabelas de fluxo. Cada entrada dessas tabelas refere-se a um determinado conjunto de ações para executar nos pacotes correspondentes e contadores que mantêm estatísticas desses pacotes. Este modelo é baseado no OpenFlow e atualmente segue como referência para dispositivos SDN no plano de dados.

2.1.1.2 Camada de Interface *Southbound*

A camada de Interface *Southbound* é responsável pelas pontes de conexão entre os dispositivos de controle e dispositivos de encaminhamento de pacotes. Essas interfaces são fundamentais em SDN para realizar a separação necessária entre o plano de controle e o plano de dados. OpenFlow é a interface padrão *Southbound* mais aceita e implementado pela indústria, fornecendo especificações comuns para dispositivos de encaminhamento e também para o canal de comunicação entre os dados e elementos do plano de controle. Entretanto, outras interfaces podem ser implementadas como a P4Runtime.

Dentro do protocolo OpenFlow existem três informações principais para os controladores NOS: (i) quando ocorrer uma alteração de link ou uma porta for ligada, os dispositivos de encaminhamento enviam as mensagens baseadas em eventos para o controlador; (ii) o controlador recebe continuamente as estatísticas de dados gerados pelos dispositivos de encaminhamento; (iii) quando dispositivos de encaminhamento recebem um novo fluxo de entrada de pacotes e não existam tratamentos para esses pacotes, ou exista uma ação explícita (enviar para controlador), os pacotes são sempre encaminhados para o controlador. Portanto, as interfaces *Southbound* são canais essenciais para realizar a troca de informação entre o nível de fluxo e o controlador NOS.

2.1.1.3 Camada de Hipervisores de Redes

Os hipervisores permitem que máquinas virtuais heterogêneas e distintas possam compartilhar os mesmos recursos de hardware, desde armazenamento até a computação. A virtualização tornou-se uma das principais plataformas de computação, permitindo que máquinas virtuais sejam criadas e/ou destruídas sob demanda, como também migradas

entre servidores físicos. Com a virtualização, os provedores gerenciam de maneira fácil e eficiente o uso da capacidade de suas infraestruturas, assim como, ao aumentar fluxos de demanda, geram aumento da receita com minimização dos custos.

Similarmente ao que já é consolidado na área de virtualização de servidores, o conceito de virtualização de redes permite que múltiplas redes coexistam sob a mesma infraestrutura de rede. O hipervisor de rede é um software baseado nos conceitos de SDN que permite a separação lógica das regras de encaminhamentos para fluxos de redes virtuais distintas. Para se implementar a virtualização de redes de forma completa, a infraestrutura das redes deve permitir o suporte para topologias de rede arbitrárias e abstração dos esquemas de endereçamento. Atualmente existem propostas de hipervisores de rede aproveitando os avanços de SDN, mas algumas questões ainda precisam de avanços, como suporte para virtualização aninhada (CASADO; FOSTER; GUHA, 2014) e técnicas de mapeamento virtual para físico (GHORBANI; GODFREY, 2014).

2.1.1.4 Camada de Sistemas Operacionais de Rede

Atualmente os sistemas operacionais tradicionais fornecem segurança e permitem abstrações de acesso aos recursos subjacentes dos dispositivos de nível inferior. A ideia de sistemas operacionais mantém-se praticamente ausente nas redes, devido a atual limitação das redes serem configuradas e gerenciadas por NOS de fornecedores variados contendo instruções e dispositivos de rede específicos.

Similar a um sistema operacional, o NOS oferece um controle logicamente centralizado, abstrai os detalhes do nível inferior de conexão com os dispositivos de encaminhamento e possibilita a criação de novos ambientes. O NOS aumenta o ritmo de inovação e minimiza a complexidade para criação de novos protocolos ou aplicativos de rede. Portanto, SDN possibilita a facilidade no gerenciamento de redes, abstrações de problemas, APIs comuns para desenvolvedores e serviços essenciais para rede.

2.1.1.5 Camada de Interfaces *Northbound*

As interfaces *northbound* são abstrações importantes em SDN caracterizadas por um conjunto de softwares essenciais para permitir a portabilidade e interoperabilidade de aplicativos com diferentes controladores NOS. Portanto, interfaces *northbound* precisam garantir independência entre as linguagens de programação e o controlador NOS, semelhante ao sistema POSIX (GROUP, 2014) em sistemas operacionais. No contexto atual de redes, interfaces *northbound* continuam sendo um problema em aberto, mas existem discussões sobre o assunto em (DIX, 2013), (GUIS, 2012), (FERRO, 2012), (CASEMORE, 2012) e (PEPELNJAK, 2012).

2.1.1.6 Camada de Virtualização Baseada em Linguagem

Soluções de virtualização expressam modularidade e permitem diferentes níveis de abstração. Uma técnica de virtualização pode representar a abstração de um "grande *switch*" virtual combinando diversos equipamentos de encaminhamento. Essa abstração facilita o desenvolvimento e implantação de aplicações de rede complexas. Na técnica de fatiamento, a rede é dividida por um compilador que se baseia em definições da camada de aplicação. Este fatiamento estático permite implantações com requisitos específicos como isolamentos e desempenho.

2.1.1.7 Camada de Linguagens de Programação

Com os avanços em códigos mais portáteis e reutilizáveis, a programabilidade em redes está começando a migrar das linguagens de programação de baixo nível para linguagens de programação de alto nível. As linguagens de programação de alto nível podem ser utilizadas para implementar e abstrair muitas propriedades e funcionalidades importantes em SDN. Com elas, também é possível desenvolver e implementar topologias de redes virtuais. Semelhante à programação orientada a objetos, essas linguagens de alto nível abstraem dados e funções para desenvolvedores, visando facilitar a solução de problemas específicos.

2.1.1.8 Camada de Aplicações de Rede

As aplicações de rede representam a inteligência da infraestrutura, uma vez que implementam as lógicas de controle que serão traduzidas em comandos e encaminhadas através das camadas inferiores da rede para serem instaladas nos dispositivos de encaminhamento do plano de dados. SDN pode ser implementada em qualquer ambiente de rede tradicional, possibilitando que as aplicações de rede SDN consigam executar qualquer funcionalidade dos dispositivos tradicionais.

Além das funcionalidades como roteamento, balanceamento de carga, firewall e aplicação de políticas de rede, as aplicações de SDN também realizam novas abordagens como virtualização de rede, redução do consumo de energia, QoS de ponta a ponta, *failover* e confiabilidade para o plano de dados e muitas outras. A maioria dos aplicativos de SDN podem ser agrupadas em (i) medição e monitoramento, (ii) mobilidade e wireless, (iii) engenharia de tráfego, (iv) rede de data centers e (v) segurança e confiabilidade.

Para informações mais detalhadas sobre as camadas, o leitor pode ser direcionado para (KREUTZ et al., 2015), em que abordam de maneira específica as principais propriedades e conceitos das camadas, baseados em diferentes tecnologias e soluções.

2.2 Programabilidade do Plano de Dados

Apesar de SDN ser um conceito genérico sobre a programabilidade de infraestruturas de rede, muito dos avanços recentes na área se deram para permitir maior programabilidade do plano de controle. O plano de dados, entretanto, permaneceu composto por dispositivos simples e com pouca capacidade de processamento. Ainda, as ações e primitivas implementadas pelo plano de dados dos dispositivos de rede passou a ser determinada pelas especificações do próprio protocolo OpenFlow. Dessa forma, mesmo as aplicações residentes no plano de controle ficaram limitadas a um conjunto pré-determinado de ações implementadas no plano de dados. Por exemplo, suponha que um operador queira que o plano de dados faça amostragem de pacotes de acordo com uma métrica nova proposta por ele. Neste caso, havia a necessidade de se projetar equipamentos de encaminhamentos com suporte a tal funcionalidade, além de estender a própria definição do protocolo OpenFlow.

Para permitir a programabilidade do plano de dados e quebrar a dependência entre as funcionalidades disponíveis, nos últimos anos surgiram vários esforços para prover abstrações para permitir a rápida programação e prototipação de planos de dados. Um desses esforços recentes que tem recebido atenção da indústria e da academia é a linguagem Programming Protocol-independent Packet Processors (P4) (BOSSHART et al., 2014b). P4 (*Programming Protocol-independent Packet Processors*) é uma linguagem de programação utilizada para programar o plano de dados de dispositivos de encaminhamento (CONSORTIUM, 2020) como, por exemplo, SmartNICs, *Field Programmable Gate Array* (Field Programmable Gate Array (FPGA)), e roteadores. Em um dispositivo de encaminhamento tradicional, o plano de dados é definido de acordo com as premissas propostas pelo fabricante. Já o plano de controle controla o plano de dados gerenciando as entradas nas tabelas de encaminhamento. Por outro lado, dispositivos programáveis possuem duas características básicas: (i) o plano de dados é definido no momento em que o código P4 compilado é carregado no dispositivo (por exemplo, por meio de um *firmware*); e (ii) o plano de controle se comunica com o plano de dados utilizando o mesmo canal, porém os programas P4 podem ser modificados a qualquer momento pelo controlador.

Para além dos dispositivos de rede programáveis comumente utilizados no plano de dados (isto é, roteadores programáveis e SmartNICs), há uma crescente adoção da comunidade científica e da indústria com tecnologias em *softwares* que permitam executar aplicações de rede com alto desempenho em planos de dados completamente em *software*. A seguir, descreve-se o as tecnologias eBPF e XDP, as quais são alvo de estudo nesta monografia.

2.3 Processamento de Pacotes com eBPF e XDP

2.3.0.1 Máquina BPF Clássica

Berkeley Packet Filter (BPF) é um filtro de pacotes que foi desenvolvido por (MCCANNE; JACOBSON, 1993) para realizar a filtragem de pacotes de rede no kernel de sistemas Unix BSD. O design consiste em um conjunto de instruções de 32 bits e também de uma Máquina Virtual (VM) para execução dos programas escritos na linguagem BPF. Desta forma, os programas fornecidos pelo usuário podem ser executados de maneira segura no kernel. Para isso, o código de Bytes (*bytecode*) do programa é transferido do espaço de usuário para o kernel e em seguida ocorre a verificação para prevenir problemas no kernel (por exemplo, acessos indevidos na memória ou *loops* sem condição de parada). Ao terminar o procedimento de verificação, o programa é compilado pela máquina de compilação JIT(Just-In-Time) para BPF no kernel. Posteriormente, este código compilado é adicionado a um soquete e a cada pacote de rede recebido, o código BPF é executado para processar o pacote. Com a máquina de compilação JIT no kernel e instruções de 32 bits simples e bem definidas, a ferramenta possui fatores fundamentais para um bom desempenho. De forma geral, a arquitetura do BPF é composta de código de Bytes, um modelo de melhoria baseado em pacotes, registradores, um armazenamento de memória temporário e um contador de programas. A arquitetura é ilustrada na Figura 4.

Desde a versão 2.5, o kernel do Linux possui suporte para BPF. No ano de 2011, o interpretador BPF foi alterado para ser um tradutor dinâmico de programas, ou seja, o kernel do Linux traduzia programas BPF para serem executados em arquiteturas específicas como x86, ARM, MIPS, etc.

2.3.0.2 Máquina BPF extendida – eBPF

Com a evolução das arquiteturas de computadores, muitos processadores modernos começaram a utilizar registradores de 64 bits e a implementar novas instruções para trabalhar com multiprocessadores. Na medida que os processadores modernos evoluíram, o design inicial do BPF com instruções de 32 bits começou a ficar defasado. Uma nova versão do BPF, o eBPF foi criado para aproveitar os recursos existentes dos novos processadores e utilizar instruções de 64 bits, sendo introduzido da versão 3.5 do kernel.

A Figura 4 ilustra as arquiteturas do cBPF e do eBPF. No lado esquerdo da figura, pode-se visualizar a primeira versão do BPF – a cBPF –, enquanto que à direita temos a arquitetura eBPF. Na nova arquitetura, a largura dos registradores evoluiu para 64 bits e o número de registradores passou de 2 para 11 (10 podem ser escritos). O registrador r0 armazena o valor de retorno de funções e da saída do programa eBPF, caracterizando a ação que será feita no encaminhamento do pacote. De r1 a r5 são armazenadas as passagens de argumentos das funções. Entre r6 e r9, são armazenados os valores em chamadas de função. No último registrador, o r10 é exclusivamente de leitura

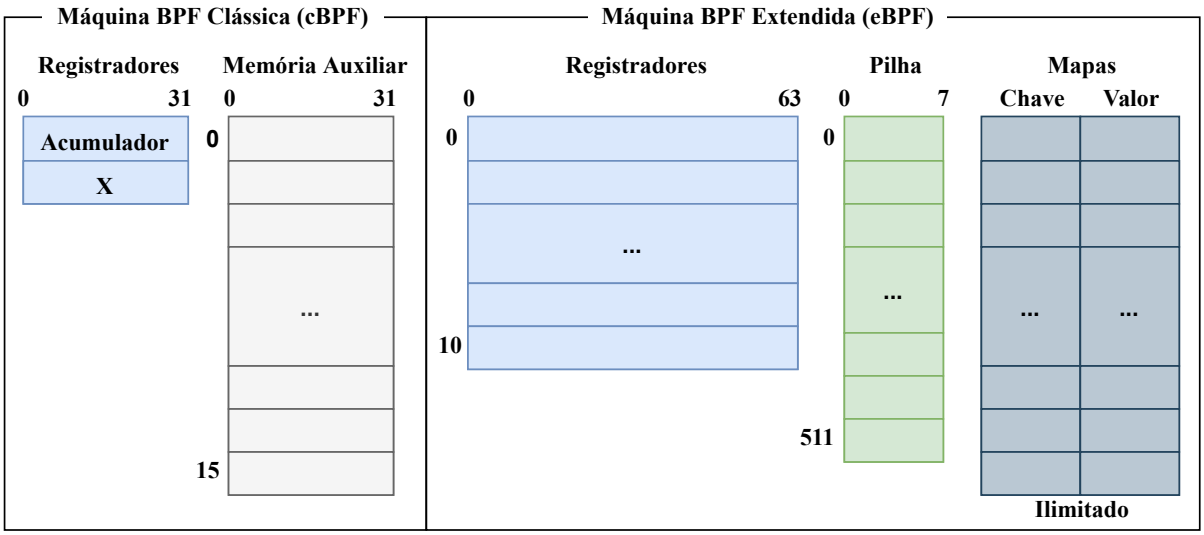

Figura 4 – Arquitetura cBPF e ebpf.

para armazenar o ponteiro do quadro para acesso à pilha. Foram adicionados também na arquitetura, uma pilha de 512 Bytes, os *mapas* (armazenamentos de dados globais ilimitados) e as funções auxiliares (opção de chamadas de funções que são executadas dentro do kernel).

Os programas eBPF possuem limitação de 4096 Bytes, por isto, foram implementadas as chamadas de cauda (*tail-call*s) que são capazes de passar o controle de execução para um novo programa. Outra mudança ocorreu nos desvios, sendo que na cBPF precisavam ser definidos os desvios para os valores verdadeiros e falsos. Já na eBPF só precisam ser definidos os desvios verdadeiros, os falsos seguem a sequência de execução do programa. O kernel gerencia o ciclo de vida dos programas e a máquina virtual eBPF consegue carregar e recarregar programas dinamicamente. Sendo assim, é possível adicionar ou remover partes dos programas e carregá-los novamente, para estender ou limitar a quantidade de processamento necessário.

Mapas eBPF são estruturas que armazenam diferentes tipos de dados contendo pares de chave e valor (similar a uma estrutura de dados baseado em dicionários). Esses tipos são *blobs* binários, definidos pelo usuário durante a criação do mapa. Os tipos também determinam em qual contexto podem ser usados. A criação de *mapas* pode ser feita diretamente por programas eBPF carregados no kernel ou através de programas no espaço de usuário utilizando a chamada de sistema *bpf*. Vários *mapas* podem ser criados por um processo no espaço de usuário. Para um *mapa* ser destruído, deve ser fechado o descritor de arquivo que está associado a ele. Como descrito em (VIEIRA et al., 2019), *mapas* podem ser acessados paralelamente por outros programas eBPF, permitindo o compartilhamento de dados entre aplicações no kernel e entre aplicações do kernel com aplicações no espaço do usuário.

Ainda, funções auxiliares podem ser consideradas para manipular os dados armazenados nos *mapas* e realizar interações com o kernel. Como diferentes programas eBPF

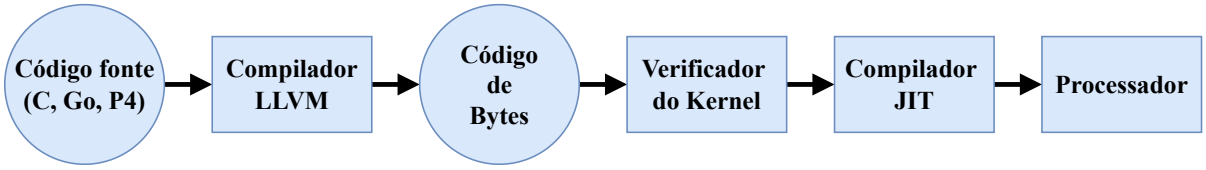

Figura 5 – Fluxo do sistema eBPF.

são executados em contextos diferentes, cada programa consegue chamar apenas um conjunto dessas funções. Novas funções auxiliares podem ser adicionadas ou estendidas por meio de extensões do kernel e não por meio da adição de módulos. Funções auxiliares oferecem um bom desempenho e não adicionam *overhead* pois ao serem adicionadas pelos programas BPF, elas são chamadas diretamente e compiladas em tempo de execução.

2.3.0.3 Sistema eBPF

O sistema eBPF tem seu fluxo de execução ilustrado na Figura 5. Inicialmente, os programas são escritos em linguagem de alto nível (C, Go e P4) e compilados para código de Bytes (*bytecode* de arquivos objeto/ELF) através do LLVM. Este *bytecode* é então analisado pelo carregador BPF ELF do espaço do usuário, e por meio da chamada do sistema BPF, é inserido no kernel. O kernel verifica essas instruções eBPF e faz a tradução dinâmica (JIT), gerando um novo *bytecode* que pode ser transferido para execução em um hardware específico ou pode ser executado pelo próprio processador.

Para garantir a validade, integridade, segurança e desempenho dos programas eBPF que são carregados para o sistema, o kernel do Linux faz uso de um verificador, localizado no arquivo `kernel/bpf/verifier.c`. Este analisa estaticamente se um programa termina, qual a maior profundidade do caminho de execução e se os acessos de memória estão em intervalos permitidos. Uma descrição geral do verificador é apresentada em (T. et al., 2018), o qual mostram que o verificador é executado depois da compilação do código e também antes de ser carregado para o plano de dados.

2.3.0.4 Programas eBPF

O eBPF oferece vantagens importantes, como por exemplo possibilitar um ambiente de programação flexível e segura, programação para diferentes contextos no kernel e também possui suporte para diferentes tipos de aplicações, como filtragem de pacotes, classificação de tráfego, medições de desempenho, etc. Para tornar programas mais rápidos, o eBPF faz uso da compilação JIT, onde os *bytecodes* são compilados para instruções nativas da máquina.

O tipo de programa eBPF precisa definir qual é a entrada (contexto), onde será carregado no kernel e também quais funções auxiliares podem ser utilizadas. Os tipos dos programas eBPF suportados são definidos através da *enum* `bpf_prog_type` em `bpf.h`. Exemplos de programas eBPF podem ser encontrados direto no código fonte do Linux,

nos diretórios `samples/bpf` e `tools/testing/selftests/bpf`. Arquivos terminados em *user.c* são utilizados para o espaço de usuário e terminados em *kern.c* podem ser carregados no kernel.

2.3.0.5 Instruções eBPF

O eBPF adicionou uma instrução para chamadas de função no kernel de forma barata e novas instruções lógicas e aritméticas para seus registradores. Assim como na linguagem C, os parâmetros são passados para as funções através dos registradores da máquina virtual. Como o eBPF possui funções auxiliares, elas permitem que programas realizem chamadas de sistema para interagir diretamente com o kernel durante o processamento.

Os programas eBPF podem ser escritos a partir de quatro categorias principais de instruções: carregamentos (*loads*), armazenamentos (*stores*), desvios (*jumps*) e operações ULA (lógicas e aritméticas). Nas classes de instruções que realizam operações de carregamento (*loads*), tem-se *loads* imediato (`BPF_LD`) e *loads* estendido para 64 bits (`BPF_LDX`). Essas duas classes de instruções são utilizadas para carregar 8 bits (Byte (B)), 16 bits (meia palavra half word(H)), 32 bits (palavra word(w)) e 64 bits (palavra dupla double word (DW)). Nas instruções (`BPF_LD`) são realizados carregamentos dentro da memória, ao contrário das (`BPF_LDX`) que realizam carregamentos externos, como na memória de pilha, dados de valor de *mapa* e dados de pacote.

`BPF_ST` e `BPF_STX` compõem as classes de instruções para operações de armazenamento (*stores*). As instruções do tipo `BPF_ST` são caracterizadas por armazenar dados na memória, mas apenas quando o operando de origem for um valor imediato. A classe `BPF_STX` possui instruções que realizam o armazenamento dos dados de um registrador externo (pilha, valor de *mapa* ou dados de pacote) para a memória. Existem também instruções especiais caracterizadas por executar palavras (32 bits) e palavras duplas (64 bits) baseados em operações de adição atômica que geralmente são usadas em contadores.

Na classe `BPF_JMP` de instruções de desvios (*jumps*), ou seja, saltos nas execuções de instruções, temos os saltos condicionais e incondicionais. Caracterizadas pela simplicidade, as instruções de salto incondicional movem o contador de programa (PC) para frente ou para trás. Sendo assim, a próxima instrução a ser executada de acordo com o deslocamento da atual, será *deslocamento* + 1 ou *deslocamento* - 1, dependendo do sinal (*signed*). Nos saltos condicionais, os desvios acontecem de acordo com os resultados das condições dos operandos baseados em registros. Quando o resultado das condições for verdadeiro, é realizado um salto (desvio + 1). Como eBPF não possui tratamento para condições que resultam em falso, o desvio é (0 + 1), ou seja, é executada a próxima instrução abaixo da atual. Para esta classe de instruções de desvio existem as condições que não avaliam o sinal, como igualdade (`jeq` ==), diferença (`jne` ! =), maior (`jgt` >), maior ou igual (`jge` >=), menor (`jlt` <), menor ou igual (`jle` <=) e também condições

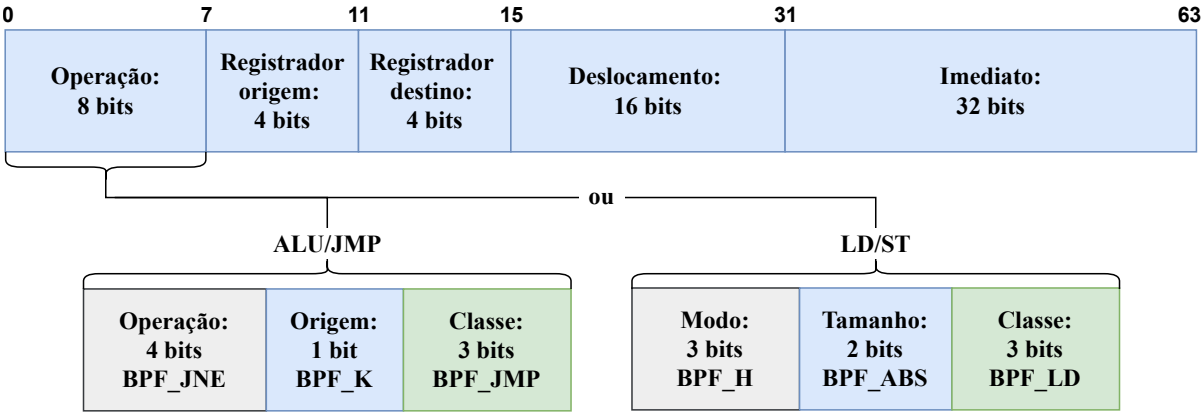

Figura 6 – Código de Bytes eBPF.

que avaliam o sinal, como maior com sinal (`jsgt` >), maior ou igual com sinal (`jsge` >=), menor com sinal (`jslt` <), menor ou igual com sinal (`jsle` <=) e `jset` (salto se origem e destino). Existem ainda 3 instruções de salto especiais, a instrução *tail* que salta para outro programa eBPF, a instrução *call* que realiza uma chamada para as funções auxiliares e a instrução *exit* que finaliza o atual programa eBPF, retornando o valor atual para o registrador r0.

`BPF_ALU` e `BPF_ALU64` fazem parte da classe de instruções que realizam operações na ULA (lógica e aritmética), sendo que `BPF_ALU` trabalha com instruções de 32 bits e `BPF_ALU64` com instruções de 64 bits. As duas instruções realizam operações contendo um operando de origem baseado em registro e outro baseado em um imediato. As duas instruções contém suporte para operações de soma, subtração, multiplicação, divisão, resto, negação, *e* lógico, *ou* lógico, *ou* exclusivo, deslocamento à esquerda (<<), deslocamento à direita (>>) e *mov*. Além destas operações, `BPF_ALU64` suporta também deslocamento a direita e `BPF_ALU` realiza instruções de conversão de *endianness* para 16 bits (meia palavra), 32 bits (palavra) e 64 bits (palavra dupla).

A Figura 6 ilustra as representações em código de Bytes (*bytecode*) das instruções eBPF. Como o processador eBPF trabalha com instruções de 64 bits, visualizando da esquerda para a direita, os primeiros 8 bits representam o código da operação *opcode*, seguido por 4 bits representando o endereço do registrador de origem, 4 bits para o endereço do registrador de destino, 16 bits para o deslocamento (*offset*) e por fim, 32 bits representando um número imediato (*imm*). Dentro do código da operação *opcode* temos ainda 2 subdivisões. Na primeira, as instruções que utilizam a ULA e instruções de desvio (JMP), os primeiros 4 bits especificam a operação de comparação, o próximo bit especifica operação com um operando de origem ou valor imediato e, os últimos 3 bits referenciam a classe de operação. Para a segunda subdivisão, os 3 primeiros bits representam o modo de acesso à memória, seguido por 2 bits para tamanho da palavra e os últimos 3 bits representando a classe de operação.

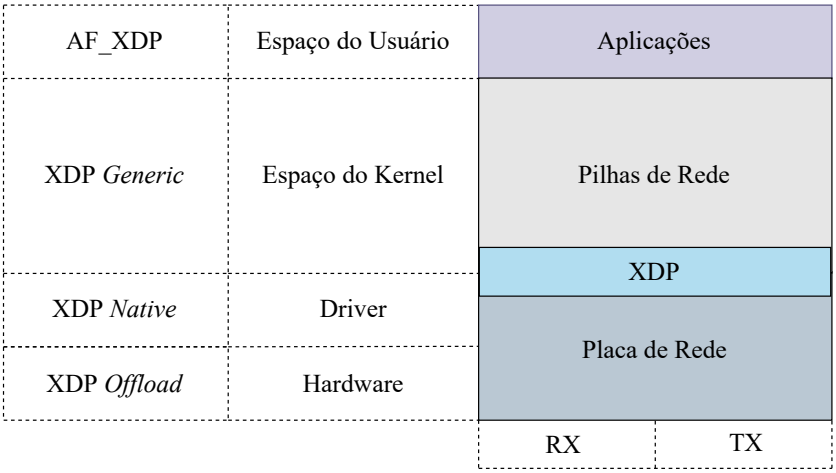

Figura 7 – Camada para Hooks XDP.

2.3.0.6 Hook eBPF/XDP

Hooks eBPF/XDP são interfaces que permitem uma programação customizada ser inserida diretamente no kernel, modificando o comportamento do sistema operacional e aplicações. Os *Hooks* realizam a interceptação de pacotes de rede antes da chamada ou durante a execução no sistema operacional. Como o processamento na pilha de rede do sistema operacional pode ser custoso devido a cópia de memória intensiva, os *Hooks* diminuem esse custo adicional, permitindo que pacotes possam ser processados antecipadamente às aplicações do espaço do usuário.

A Figura 7 ilustra as camadas existentes para o posicionamento de *Hooks* e como os pacotes de entrada são transmitidos entre a interface de rede, passando pelos componentes da pilha TCP/IP do kernel, até serem eventualmente entregues a uma aplicação no espaço de usuário. Os pacotes de rede podem ser destinados para aplicações no espaço do usuário, assim como, programas eBPF podem conectar-se com vários locais do kernel Linux para filtrar pacotes.

2.3.0.6.1 Caminho de Dados Expresso - eXpress Data Path (XDP)

Para obter maior desempenho, os pacotes de rede precisam ser filtrados/processados nas camadas mais inferiores do kernel. Com isso surgiu o *Hook* Caminho de Dados Expresso (XDP) que ocorre dentro do driver, no dispositivo de rede. Ele permite que os pacotes de rede sejam processados pela CPU na camada mais inferior do kernel ou também, transferindo a computação direto para a placa de rede (quando esta suporta tal funcionalidade).

O conjunto de ações XDP é composto por Valores e Ações. Como o valor de retorno de um programa eBPF é armazenado no registrador r0, este valor determina qual ação o XDP irá tomar para o encaminhamento de pacotes. O valor *0* indica que um erro aconteceu e o pacote é descartado, executando a ação `XDP_ABORTED`. Este tipo de

erro pode acontecer, por exemplo, quando um pacote mal formado é recebido pelo kernel. Para valor *1*, o pacote é descartado pela ação `XDP_DROP`. Neste caso, o descarte é intencional e realizado a partir de alguma política de descarte (por exemplo, em um *firewall*). No caso do valor *2* a ação `XDP_PASS` permite que o pacote continue o processamento até o kernel ou camada de aplicação. Se valor for *3*, a ação `XDP_TX` retransmite imediatamente o pacote de rede pela mesma interface de chegada. No último caso onde o valor é *4*, a ação `XDP_REDIRECT` redireciona o pacote para um alvo específico (por exemplo, uma outra porta física da interface), mas antes do programa sair, ele necessita de um parâmetro adicional definido por uma função auxiliar (por exemplo, o *id* da porta física). Com esta ação de redirecionamento, as regras podem ser alteradas dinamicamente sem modificar o programa. Através dela o pacote pode ser transmitido para uma interface de rede diferente, para uma CPU diferente (processamento adicional) ou para um soquete (*AF_XDP*) no espaço do usuário.

Para conseguir alto desempenho, os programas eBPF carregados para o *Hook* XDP, modificam o processamento de pacotes em nível da interface de rede, mas necessitam suporte por parte do *driver*. Esses *drivers* utilizam o modo XDP *Native* e podem ser visualizados na lista do projeto BBC (BCC, 2019). Para dispositivos que não possuem *Hook* XDP em nível de *driver*, temos o modo XDP *Generic*, onde o kernel do Linux oferece suporte, mas com desempenho inferior devido a alocação de *buffers* internos. Esses *buffers* são alocados quando os pacotes são recebidos pelo sistema operacional. Através desse *Hook*, qualquer dispositivo que não possui suporte ao XDP *Native* consegue executar programas eBPF.

Possuindo desempenho superior aos modos XDP *Generic* e *Native*, o XDP *Offload* descarrega e executa os programas eBPF direto no hardware da interface de rede. Para isto, é necessário um dispositivo específico que consiga executar eBPF no hardware e também é necessário indicar explicitamente o modo *Offload* durante o carregamento do programa. Nos modos XDP *Generic* e *Native* o sistema escolhe automaticamente durante o carregamento do programa eBPF, mas também podem ser carregados de maneira manual, escolhendo qual modo será utilizado.

2.3.0.6.2 Soquete AF_XDP

O *AF_XDP* é uma nova família de endereços otimizada para realizar o processamento de pacotes com alto desempenho em aplicações no espaço do usuário. Os soquetes *AF_XDP* permitem que pacotes de rede sejam redirecionados de programas XDP para um *buffer* de memória da aplicação no espaço do usuário. Desta forma, um programa XDP (*Generic*, *Native* e *Offload*) realiza o encaminhamento dos pacotes para o *AF_XDP* pela função `bpf_redirect_map()`. Através do *AF_XDP* os pacotes podem ser processados com o programa XDP que realiza o redirecionamento e também com a aplicação no espaço do usuário.

Um soquete *AF_XDP* (XSK) é criado com a chamada de sistema *syscall socket()*. Este soquete XSK possui o anel RX que recebe pacotes e o anel TX que envia pacotes. Esses anéis são registrados e dimensionados com os *setsockopts* `XDP_RX_RING` e `XDP_TX_RING`. Os anéis descritores RX ou TX apontam para um *buffer* de dados da memória chamada UMEM. Os anéis RX e TX acessam o mesmo UMEM para que um pacote não precise ser copiado entre RX e TX.

O UMEM é uma região de memória virtual contígua, dividida em quadros de tamanhos iguais e alocado pela aplicação do espaço do usuário. Ele está associado a uma interface de rede e a um identificador de fila específico dessa interface. Ele é criado e configurado usando a chamada de sistema `XDP_UMEM_REG` *setsockopt*. Um UMEM pode ser acessado por vários soquetes *AF_XDP*, mas cada soquete *AF_XDP* é vinculado a somente um único UMEM.

O UMEM possui os anéis FILL e COMPLETION. O anel FILL é usado pela aplicação para enviar o endereço de memória para o kernel preencher com dados do pacote RX, assim que cada pacote for recebido. O anel COMPLETION contém o endereço de memória com os dados do pacote que o kernel transmitiu anteriormente usando o anel TX. Portanto, os anéis RX e FILL são usados para o caminho RX e os anéis TX e COMPLETION são usados para o caminho TX.

Existe um mapa BPF chamado XSKMAP (`BPF_MAP_TYPE_XSKMAP`) onde cada aplicação do espaço do usuário pode colocar um soquete *AF_XDP* XSK neste mapa. O programa XDP, ao redirecionar cada pacote para um índice específico do mapa, também precisa validar se o XSK daquele mapa estava de fato vinculado ao dispositivo e número de anel. Caso contrário, o pacote é descartado. Atualmente é obrigatório ter um programa XDP carregado com um XSK no XSKMAP para poder obter qualquer tráfego para o espaço do usuário através do XSK.

O*AF_XDP* pode operar atualmente nos modos XDP_SKB e XDP_DRV. O modo XDP_SKB utiliza SKBs do XDP *Generic* para copiar os pacotes para o espaço do usuário. Já o modo XDP_DRV utiliza SKBs do XDP Native para copiar os pacotes para o espaço do usuário, mas somente se o *driver* da placa de rede permitir tal suporte para XDP.

Na Figura 8 são ilustrados os fluxos de tratamento dos pacotes para os diferentes *Hooks* XDP. Primeiro, o pacote de rede chega na entrada RX da SmartNIC e é encaminhado diretamente para o XDP que está ativo na interface. Esse pacote é processado pelo programa eBPF que está carregado no XDP e em seguida é encaminhado para a próxima camada de processamento, ou é encaminhado para a saída TX da SmartNIC. Caso o pacote não foi encaminhado para a saída TX da interface, este vai ser direcionado para as pilhas de rede do kernel e posteriormente encaminhado para as aplicações no espaço do usuário. Se o modo *AF_XDP* estiver ativo, o programa XDP encaminha o pacote direto para o espaço do usuário, sem passar pelas pilhas de rede do kernel.

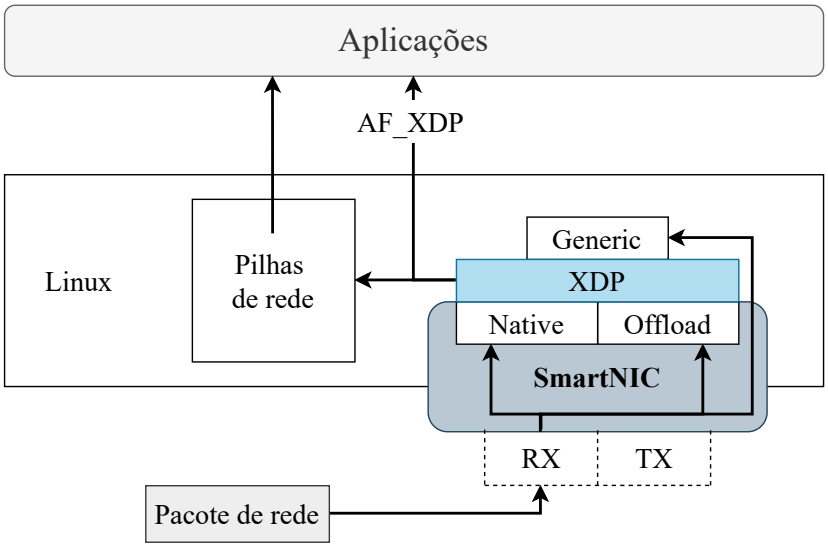

Figura 8 – Fluxo dos Hooks XDP.

2.4 Trabalhos Relacionados

Como mencionado anteriormente, eBPF e XDP tem atraído a atenção tanto da indústria quanto da academia. O motivo é relacionado com a alta flexibilidade e desempenho para tornar o plano de dados do sistema operacional programável. Nesta seção são analisados trabalhos promissores relacionados com eBPF e XDP que abordam de maneira geral a aplicação dessas tecnologias para resolver problemas variados no domínio de redes.

No trabalho (CARRASCAL et al., 2020) foram realizadas diversas análises quantitativas e qualitativas entre as tecnologias P4 e XDP para compará-las e auxiliar em futuros trabalhos de pesquisa. Como as tecnologias P4 e XDP são muito importantes e recentes para SDN, foram implementados diversos casos de uso para medir o desempenho de CPU, Latência, consumo de memória e número de linhas de código. Seus resultados mostraram que XDP é atualmente a melhor opção para dispositivos Internet das Coisas (IoT) restritos, apresentando metade do uso da CPU, Latência mais baixa e consumo de memória reduzido na comparação com P4.

Em (JOLY, 2020), os autores comparam o desempenho de chamadas de cauda entre programas eBPF antes e depois das otimizações introduzidas para mitigar falhas de espectro nas versões 5.4 e 5.5 do kernel. A versão 5.4 inclui mitigação de *retpoline* e a 5.5 introduz uma otimização de desempenho que remove a sobrecarga de *retpoline*, sempre que possível, tentando substituí-los por chamadas diretas. Os resultados mostraram que o custo por *tail-call* é de cerca de 5ns e cada chamada de cauda é 80% mais rápida na versão 5.5 do kernel.

Uma análise de várias abordagens que podem ser adotadas para introduzir SmartNICs no processamento de plano de dados é realizada em (MIANO et al., 2019b). É apresentado ainda, uma solução que combina SmartNICs com outras tecnologias recentes como eBPF/XDP para construir um pipeline de processamento mais eficiente e que per-

mita gerenciar grandes quantidades de tráfego como, por exemplo, oriunda de ataques/ invasões. A solução aprimora os recursos de mitigação dos servidores de borda, transferindo de forma transparente uma parte das regras de mitigação de Negação de Serviço Distribuída (DDoS) para o SmartNIC. A solução proposta obteve melhor desempenho, resultando em maior eficiência de taxa de queda e uso da CPU, devido à combinação de filtragem de hardware no SmartNIC e filtragem de software XDP no host.

Uma avaliação é realizada em (ROSSI et al., 2021) para quantificar as limitações de desempenho existentes em hardwares SmartNICs. A partir do estudo, medição de desempenho são obtidas em termos de Latência e Taxa de Transferência, de diferentes programas P4, para uma infinidade de cenários de uso intensivo de memória de pacotes. Os resultados mostraram que a Taxa de Transferência pode diminuir em até 8 vezes, mas a Latência pode aumentar em até 80 vezes ao executar operações com uso intenso de memória no plano de dados.

No trabalho (KICINSKI; VILJOEN, 2016), apresenta-se um método usado para descarregar programas eBPF/XDP para SmartNICs e permitir a aceleração geral de qualquer programa eBPF. Já em (KRUDE et al., 2021), os autores implementam uma metodologia que determina a Taxa de Transferência alcançável de um programa executado em uma SmartNIC em termos de taxa de pacotes alcançável e taxa de bits. Assim, um desenvolvedor de programa ou operador de rede pode determinar se um programa executado em uma SmartNIC sempre atingirá a Taxa de Transferência necessária. Essa abordagem combina a busca incremental do caminho mais longo com verificações SMT para estabelecer um limite inferior para o caminho mais lento do programa satisfazível. Ao analisar apenas os caminhos de programa mais lentos, a abordagem estima os limites de Taxa de Transferência em alguns segundos.

A partir de (PAROLA; PROCOPIO; RISSO, 2021), apresentam-se experimentos preliminares entre as tecnologias XDP e *AF_XDP*, de modo a melhorar a Taxa de Transferência geral de servidores. Em (HOHLFELD et al., 2019) são discutidos os benefícios e deficiências do descarregamento genérico do kernel *AF_XDP*, o descarregamento do *driver* do dispositivo XDP e até o descarregamento de programas XDP para uma SmartNIC. Os resultados indicam que a SmartNIC fica facilmente sobrecarregada com tarefas muito pesadas, mas se destaca com Latência ultra baixa no processamento de pequenas tarefas.

Em (BRUNELLA et al., 2020), os autores apresentam o projeto e implementação do hXDP, um sistema para executar programas XDP em placas de rede FPGA, usando apenas uma fração dos recursos de hardware disponíveis e combinando o desempenho de CPUs de ponta. A implementação tem clock de 156,25 MHz e usa cerca de 15% dos recursos do FPGA. Seus resultados mostraram que a solução atinge a Taxa de Transferência de processamento de pacotes de um núcleo de CPU de ponta e fornece uma Latência de encaminhamento de pacotes 10 vezes menor.

Outra análise mais abrangente ocorre em (COSTA et al., 2021), onde os autores

avaliam o desempenho e maturidade de onze switches com suporte ao protocolo OpenFlow de software e hardware que implementam o protocolo nas versões 1.0 e 1.3. Os resultados mostraram que implementações de software otimizadas para SO, suportam um número maior de regras de fluxo, têm um desempenho comparável aos switches de hardware e suportam mais recursos do protocolo OpenFlow do que os switches de hardware avaliados.

Como eBPF e XDP analisam constantemente e com alto desempenho os pacotes da rede, muitos pesquisadores utilizaram essa premissa para inovar seus estudos na área de monitoramento de redes. Uma estrutura prática de monitoramento de rede baseada em software é apresentada em (ABRANCHES et al., 2021). A solução pode ser parcialmente descarregada para um SmartNIC, que aciona aplicações em nível de usuário somente quando necessário, com base em métricas de tráfego de alto nível para evitar cálculos desnecessários e redundantes. INTCollector é apresentado em (TU et al., 2018) sendo um coletor de alto desempenho para telemetria de rede *in-band* que extrai informações importantes da rede e realiza chamadas de eventos dos dados brutos do INT. O mecanismo filtra os eventos da rede, reduzindo o uso da CPU, o custo de armazenamento e a quantidade de dados necessários para serem armazenados.

Em (SOMMERS; DURAIRAJAN, 2021) foi desenvolvida a ferramenta ELF para monitoramento de fluxo *in-band* que envia sondas limitadas por saltos dentro de um fluxo existente. Nos experimentos, 90% dos roteadores percorridos pelas sondas responderam positivamente. Já em (DERI; SABELLA; MAINARDI, 2019), uma nova ferramenta combina visibilidade de sistema e rede, para detecção de falhas de segurança e aplicação de políticas de segurança. A ferramenta explora sondas e pontos de rastreamento do kernel do Linux para interceptar comunicações. Isso permite que a ferramenta tenha visibilidade das comunicações de rede com metadados em nível de sistema, incluindo processos de origem e destino, usuários e contêineres.

O trabalho (SANTOS et al., 2019) propõe a implementação de mecanismos utilizando o BPFabric para aplicações de monitoramento de tráfego. Com base em amostragem e também utilizando estruturas de dados probabilísticos, as aplicações foram implementadas para realizar a contagem do número de fluxos ativos ao longo do tempo e o número de pacotes por protocolo IP que trafegam em uma rede.

Para o monitoramento de desempenho de redes SRv6, o trabalho (LORETI et al., 2020) desenvolveu a solução SRv6-PM que consegue rastrear eventos de perda de um único pacote em tempo quase real. O SRv6-PM, apresenta uma arquitetura nativa em nuvem para o controle baseado em SDN de roteadores Linux e para ingestão, processamento, armazenamento e visualização de dados PM.

Implementado em (WENG et al., 2021), o Kmon é um sistema de monitoramento transparente in-kernel para sistemas de microsserviços. A solução captura vários tipos de indicadores e os organiza em três níveis para fornecer várias informações de tempo de execução de microsserviços, como Latência, topologia e métricas de desempenho com

baixa sobrecarga. O ViperProbe, apresentado em (LEVIN, 2020), é outra ferramenta de monitoramento de alto desempenho para microsserviços, mas é voltado especificamente para servicemensh.

Em (LIU et al., 2020), um sistema não intrusivo para observabilidade de rede de contêineres é desenvolvido para coletar informações da interação de aplicativos do usuário em protocolos de rede de contêineres em ambiente nativo de nuvem. O sistema faz uso de um método de aprendizagem de máquina para analisar e diagnosticar o desempenho e os problemas na rede de aplicativos. No trabalho (NAM; KIM, 2017) é utilizado o rastreamento de pacotes baseados em eBPF para monitorar conexões entre caixas Linux.

Completando os trabalhos que abordam monitoramento, temos ainda (RIVERA et al., 2020) e (BHAT et al., 2021). Em (RIVERA et al., 2020), os autores aproveitam eBPF e XDP para construir ROS-FM, um sistema de monitoramento para sistemas robóticos construídos em cima do Robot Operating System (ROS). Já em (BHAT et al., 2021), foram utilizadas inovações de monitoramento e verificação de rede para o desenvolvimento de uma técnica de controle de congestionamento em tempo real para redes sem fio. A técnica integra o eBPF com o INT para coletar informações detalhadas em tempo real de telemetria de rede *in-band* e modifica os algoritmos de controle de congestionamento em tempo real para ajustar a transferência de dados no transmissor.

Em virtude da segurança, programabilidade e do alto desempenho proporcionado com o uso de eBPF e XDP para realizar o processamento de pacotes em placas de rede e no kernel, muitos pesquisadores estão estudando a filtragem e tratamento de pacotes com eBPF e XDP para a criação de firewalls. Seguindo nesta linha, (DEEPAK et al., 2020) apresenta uma solução de implementação de filtro de pacotes baseado em XDP e também com o firewall baseado em iptables. A proposta de traduzir as regras do iptables para uma solução baseada em BPF, possibilita uma solução melhor, sem a necessidade de implementar ou aprender novas formas de configurar firewalls.

A solução bpf-iptables, apresentada em (MIANO et al., 2019a), caracteriza-se por um firewall Linux que implementa iptables usando *Hooks* eBPF e XDP no kernel. A solução traduz as regras iptables em programas eBPF, preserva a filtragem semântica e melhora a velocidade e escalabilidade de iptables. Bpf-iptables aproveita as características de eBPF, como a compilação dinâmica e injeção dos programas eBPF no kernel em tempo de execução para cooperar com o restante das funções do kernel e outras ferramentas do ecossistema Linux. Os resultados mostram que a solução melhora significativamente o rendimento em comparação com as alternativas iptables e nftables, com maiores vantagens para um número maior de regras de filtragem.

Na prevenção de ataques de sondagem baseados em sinalizadores TCP (Null, FIN, XMAS), (BERTOLI et al., 2020) apresenta uma solução com o uso efetivo de filtros de pacotes de rede através de LKM e eBPF/XDP. A abordagem é implementada em sistemas operacionais Linux por filtragem de baixo nível por meio do Linux kernel Module

(LKM) e do Netfilter para operar diretamente na pilha de rede. Para amenizar os problemas causados por ataques DDoS, (BERTIN, 2017) apresente o GateBot, um sistema baseado em eBPF e XDP, totalmente automatizado e que foi desenvolvido pela Cloudflare para mitigação de ataques DDoS. A arquitetura do sistema passa primeiro por uma amostragem do tráfego, seguida pela agregação e análise. Por fim, o sistema reage aos ataques e posteriormente mitiga os mesmos. Em (DIMOLIANIS; PAVLIDIS; MAGLARIS, 2021a) é proposto um esquema para filtragem e classificação do tráfico DDoS. Com o uso de aprendizagem de máquina obtém-se algumas regras de filtragem, para que os pacotes sejam classificados entre maliciosos ou não. São dois os meios de aplicação, o primeiro através de Deep Packet Inspectors programáveis ou firewalls flexíveis que por sua vez bloqueiam os pacotes maliciosos.

Outra solução para ataques DDoS, temos o trabalho (DIMOLIANIS; PAVLIDIS; MAGLARIS, 2021b), que implementa um esquema de detecção e mitigação para ataques DDoS TCP SYN Flood por meio de firewalls com XDP. O esquema coleta e analisa dados de pacotes apropriados, formando assinaturas que são usadas como entrada para modelos de aprendizado de máquina supervisionados. Esses modelos detectam ataques SYN, identificam vítimas e isolam assinaturas maliciosas. Todo o tráfego encaminhado para vítima é redirecionado para firewalls com XDP, que atenuam ataques, filtrando pacotes SYN maliciosos com base nas assinaturas calculadas. Após esta atenuação, os pacotes são encaminhados para o mecanismo SYN *cookies* que processa e manipula adequadamente os pacotes.

Uma ferramenta é desenvolvida em (SCHOLZ et al., 2018) para instalar configurações de filtragem de pacotes específicos no nível do soquete. A ferramenta permite que desenvolvedores de aplicativos definem as regras de firewall para limitar a exposição de rede do aplicativo. Em seus resultados, o XDP em hardware pode render quatro vezes o desempenho em comparação com a execução de uma tarefa semelhante no kernel. Em (DUARTE et al., 2021), um SDI Serverless é implementado com a utilização de filtros escritos em eBPF para realizar análise e filtragem de pacotes trafegados na rede. Quando esses filtros são descarregados no hardware, são capazes de analisar o tráfego e definir o destino dos pacotes de modo programável, podendo ser alterados em tempo de execução sem gerar sobrecarga no SDI. O SDI proposto utiliza a SmartNIC Netronome para acelerar o processamento de pacotes e reduzir custos operacionais. Os resultados mostraram que ataques são detectados em taxa de linha.

Seguindo o tema de firewalls com descarregamento em hardware, temos o trabalho (KOSTOPOULOS; KALOGERAS; MAGLARIS, 2020), onde o XDP é utilizado como Deep Packet Inspection (DPI) para mitigar ataques de Water Torture no nível do driver Placa de Interface de Rede (NIC) de servidores DNS autoritativos. Na abordagem apresentada, o XDP intercepta os pacotes com solicitação de DNS, extrai os nomes da carga útil da mensagem e classifica-os em categorias, de acordo com a validade dos resultados

apresentados por Filtros Bloom. Após esse processo, os nomes inválidos são descartados pelo kernel do Linux e os nomes válidos são encaminhados para o espaço do usuário, onde serão resolvidas as solicitações.

Outras duas soluções para firewalls, mas com foco em DNS, são apresentadas em (KOSTOPOULOS et al., 2021) e (STEADMAN; SCOTT-HAYWARD, 2021). No trabalho (KOSTOPOULOS et al., 2021), classificadores Naive Bayes baseados em XDP e eBPF, são implementados no plano de dados para mitigar ataques de Water Torture nos resolvedores de data center e assim, diferenciar as solicitações DNS válidas e inválidas. As solicitações consideradas inválidas são descartadas antes que qualquer recurso seja alocado a elas no kernel. Já as solicitações válidas são encaminhadas ao espaço do usuário para serem resolvidas. A arquitetura DNSxP apresentada em (STEADMAN; SCOTT-HAYWARD, 2021), detecta e mitiga ataques maliciosos de exfiltração de dados que exploram o protocolo DNS. A solução realiza a filtragem e análise de pacotes de granulação grossa no plano de dados, identifica rapidamente o tráfego suspeito e o envia para ser classificado em controles de segurança adicionais no controlador SDN.

Apesar do alto desempenho de eBPF e XDP para processamento de pacotes, novos programas eBPF surgem com funcionalidades mais complexas, sendo assim, torna-se necessário otimizações para melhorar o desempenho. A partir desta questão, alguns estudos abordam soluções de otimização de eBPF e XDP, como por exemplo, em (MIANO et al., 2021b), é implementado o Morpheus, um sistema que trabalha em conjunto com compiladores estáticos para otimizar continuamente o código de rede arbitrária. Para isso eles introduziram novas técnicas, desde análise estática de código até instrumentação de código adaptável e implementaram uma caixa de ferramentas de otimizações específicas. Os resultados mostraram que a solução proposta consegue trazer até 2x a melhoria da Taxa de Transferência e reduzir pela metade a Latência.

A arquitetura HIKE (HybrId Kernel/eBPF forwarding), apresentada em (MAYER et al., 2021), é uma solução de plano de dados programável para roteadores Linux que permite uma abordagem híbrida unindo o eBPF/XDP e o encaminhamento baseado em kernel para acelerar o desempenho dos roteadores de software SRv6. A arquitetura também promove uma abordagem modular combinando diferentes programas eBPF para realizar processamento de pacotes SDN complexos e personalizáveis. A solução apresentou melhoria na Taxa de Transferência de até 5x em relação a uma solução convencional baseada em Linux.

Uma arquitetura híbrida para construir e acelerar VNFs no XDP do kernel é proposta por (TU; YOO; HONG, 2020). A solução eVNF proposta processa as tarefas simples e críticas no XDP dentro do kernel, e o processamento de tarefas complexas é realizado fora do XDP. Com eVNF foram obtidas melhorias significativas para Taxa de Transferência, redução da Latência e uso da CPU, quando comparadas com soluções tradicionais. Para melhorar o paralelismo em nível de aplicativo, em (ENBERG; RAO;

TARKOMA, 2019) é proposta uma abordagem combinando o particionamento em nível de aplicativo com o direcionando de pacotes através de uma NIC programável. Para isto, um programa eBPF particiona seus dados e recursos em DRAM para uso entre os núcleos e outro programa XDP, executando numa NIC programável, verifica os cabeçalhos do protocolo L7 para direcionar a solicitação para sua partição correta.

Em (LUIZELLI et al., 2021), os autores abordam ideias preliminares no campo de redes neurais em rede e discutem os desafios técnicos de executar técnicas de aprendizado de máquina inteiramente no plano de encaminhamento. São destacados ainda, possíveis casos de uso de uma rede inteligente autônoma capaz de se adaptar a mudanças dinâmicas de comportamento de rede com mínima ou nenhuma intervenção humana. Também são apresentados os desafios que devem ser abordados pela comunidade de pesquisa para realizar a visão de um plano de encaminhamento totalmente programável, inteligente e autônomo. No trabalho (SAQUETTI et al., 2021), os autores discutem os desafios de pesquisa envolvidos na expressão de redes neurais para planos de dados programáveis, mapeamento de neurônios para switches e habilitação da comunicação neuronal. Uma abordagem também é desenvolvida com a ideia de distribuir os neurônios de uma RNA em vários switches programáveis, em vez de executar uma ANN inteira em um único dispositivo. As vantagens dessa abordagem incluem maior visibilidade do fluxo de rede e melhor uso de recursos entre switches e links. Os resultados mostram que a abordagem melhora as tarefas de gerenciamento de rede, mantendo a sobrecarga de provisionamento semelhante a uma linha de base.

Com o propósito de aproveitar todo o potencial de eBPF/XDP e torná-lo utilizável para vários segmentos, muitos pesquisadores já estão desenvolvendo soluções para Gerenciamento de Energia, Segurança em Sistemas Operacionais, Otimizações para Camada de Transporte, Balanceador de Carga e também aplicações de Middleware e Gateway. Em (FRASCARIA; TRIVEDI; WANG, 2021), é apresentado o middleware Griffin para a computação de borda da rede. Baseado em eBPF, o Griffin caracteriza-se como um serviço de armazenamento programável permitindo que aplicativos personalizam políticas como replicação, balanceamento de carga, migração de sessão, monitoramento, exclusão de dados, consistência, coleta de lixo e uso da computação para melhorar a Latência.

Os trabalhos (PAROLA; MIANO; RISSO, 2020) e (PANTUZA; VIEIRA; VIEIRA, 2021) apresentam soluções para Gateway, baseados em eBPF e XDP. No trabalho (PAROLA; MIANO; RISSO, 2020), uma solução open-source é implementada para Gateway Mobile 5G. Já o eQUIC, apresentado em (PANTUZA; VIEIRA; VIEIRA, 2021), atua como Gateway em um sistema distribuído entre o espaço de usuário e o espaço do kernel. Por meio da aplicação QUIC com eBPF/XDP(*Offload*), o serviço eQUIC tem como objetivo bloquear e controlar quais pacotes podem prosseguir pela pilha de protocolos do sistema operacional em direção às aplicações no espaço do usuário. O eQUIC Gateway aumentou a Vazão de pacotes em 30,9%, reduziu em 26,4% o tempo de CPU para bloquear

pacotes e diminuiu 89% das chamadas de sistema (*syscall*).

A solução Polycube, apresentada em (MIANO et al., 2021a), é um framework de software baseado em eBPF que fornece flexibilidade e personalização no desenvolvimento, implantação e gerenciamento de Virtualização de Funções de Rede (NFV) no kernel. O Polycube realiza o processamento de pacotes no kernel aproveitando-se do potencial NFV e do subsistema eBPF do Linux para injetar aplicações definidas pelo usuário em pontos específicos da pilha de rede do Linux. Escrito inteiramente em eBPF, bpfbox é implementado em (FINDLAY; SOMAYAJI; BARRERA, 2020) para ser um novo mecanismo de confinamento de processo. Sua implementação permite o confinamento na função de espaço do usuário, *Hook* LSM, chamada de sistema e limites de função do espaço do kernel. Ele faz uso de uma linguagem de política simples para ser usada para fins de confinamento ad hoc.

Um balanceador de carga conteinerizado de alto desempenho é implementado em (LEE et al., 2021) para distribuir o tráfego usando eBPF/XDP dentro do kernel Linux com gerenciamento via kubernets. Foram realizadas análises de desempenho entre o balanceador de carga proposto (LBN), iptables DNAT e loopback. Seus resultados mostraram que a Taxa de Transferência do balanceador de carga proposto é semelhante ao loopback, mas muito superior ao iptables DNAT.

No sentido de melhorar a segurança do sistema operacional, o trabalho (FINDLAY, 2020) discute sobre o papel do eBPF no cenário de segurança do sistema operacional e apresenta também duas novas aplicações para o tema. O ebpH é apresentado como um sistema de detecção de anomalias eBPF baseado em host e o ebpfbox como nova técnica de sandboxing que aproveita os programas BPF para impor regras de secomp externo e de forma transparente ao aplicativo de destino.

A eficiência energética vem sendo um tema de estudo em vários segmentos da tecnologia, com isso, no trabalho (LI et al., 2020) os autores estabelecem um modelo de fluxo de processamento de E/S de pacotes de alto desempenho para explorar técnicas de gerenciamento de energia. O modelo permite deduzir informações necessárias para técnicas de gerenciamento de energia e insights para equilibrar o consumo e a Latência. O modelo sugere o uso de instruções de pausa para diminuir o consumo de energia da CPU em um período curto de inatividade, assim como, duas formas para diminuir o consumo de energia para E/S de pacotes de alto desempenho por meio de uma abordagem com auxílio de informações do tráfego e outra sem. Seus resultados mostraram que com o Intel DPDK ambas as abordagens conseguem redução significativa de energia com pouco aumento de Latência.

Em virtude do grande potencial de desempenho de eBPF/XDP e exercendo papel fundamental no plano de dados para tornar possível a implementação do paradigma SDN nas atuais redes de computadores, muitos estudos estão sendo realizados com eBPF e XDP. Entretanto, como pode ser observado nos trabalhos mencionados, apenas uma

parte dos trabalhos desenvolvidos tem como foco o provimento de aplicações eBPF e XDP com alto desempenho. Desta forma, este trabalho se propõe a entender e quantificar o desempenho de aplicações genéricas de rede baseadas em eBPF e XDP quanto a métricas de desempenho de rede. Como resultado, este trabalho permitirá entender quais as características de aplicações eBPF e XDP que impactam nos desempenho das métricas de rede, assim como nos custos computacionais envolvidos (por exemplo, utilização de CPU).

3 METODOLOGIA

Neste capítulo, descreve-se a metodologia utilizada para realizar os experimentos, bem como a metodologia utilizada para coletar os dados e gerar os resultados para as métricas de desempenho relacionadas à Taxa de Transferência (Vazão), Latência e uso de CPU. Os resultados obtidos são apresentados no Capítulo 4.

3.1 Metodologia dos Experimentos

3.1.1 Ambiente de avaliação

O objetivo deste trabalho consiste em avaliar o desempenho de aplicações de rede baseadas em eBPF e XDP. Para tanto, considera-se um ambiente experimental composto por dois servidores Dell T440. Cada servidor possui um processador Intel Xeon 4214R com 32 GB de RAM e 8 núcleos de processamento com 16 threads ao total. Um dos servidores é nosso *Device Under Test* (DUT) – ou seja, o servidor no qual os programas eBPFs/XDPs são carregados – e o outro é nosso gerador de tráfego. Ambos os servidores possuem uma interface de rede Netronome SmartNIC Agilio CX 10 Gbit/s com duas interfaces de rede, as quais estão fisicamente conectadas. A Figura 9 ilustra o ambiente descrito. Nele, os servidores são conectados diretamente através de cabos DAC de 10Gbit/s.

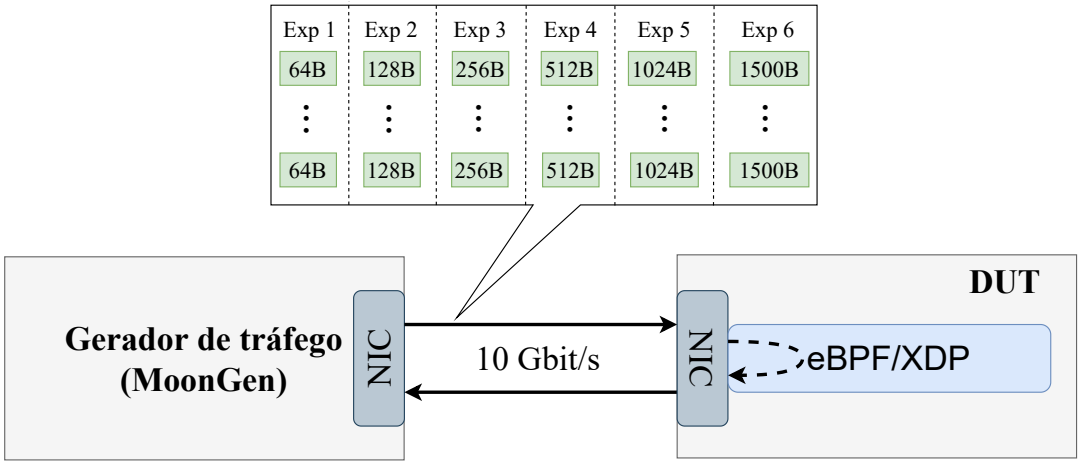

Figura 9 – Ilustração do ambiente de avaliação.

Do lado esquerdo da Figura 9 temos o gerador MoonGen (EMMERICH et al., 2015) baseado em DPDK[1] que utiliza o Netronome Packet Generator[2] para gerar e enviar milhares de pacotes IPv4 de mesmo tamanho, a uma taxa de 10 Gbit/s (máximo suportado pela interface de rede instalada) para o nosso servidor DUT, representado no lado direito. O MoonGen realiza o envio desses pacotes gerados para a rede e também coleta o retorno dos mesmos. Ele permite coletar informações sobre esses pacotes, ou seja, consegue coletar os dados de quantos pacotes foram enviados/recebidos, qual a Taxa de Transferência

[1] <https://www.dpdk.org/>
[2] <https://github.com/emmericp/MoonGen/tree/master/examples/netronome-packetgen>

de envio/recepção e ainda a Latência desses milhares de pacotes. O MoonGen possui outras funcionalidades importantes como, personalizar o número de execuções de cada geração de pacotes, personalizar o tamanho dos pacotes, variando de 64B a 1500B, e ainda, personalizar a variação dos endereços MACs/IPs de origem/destino dos pacotes IPv4 gerados.

Do lado direito da Figura 9, ilustra-se o servidor utilizado como DUT, o qual é utilizado para carregar e executar cada programa eBPF/XDP. A interface de rede do servidor DUT suporta todos os modos do *Hook* XDP, ou seja, os programas eBPF criados podem ser carregados para a interface de rede nos modos *AF_XDP*, *Generic*, *Native* e *Offload*, como ilustrado na Figura 7.

3.1.2 Programas eBPF e AF_XDP

Atualmente, poucos estudos foram realizados pela comunidade científica sobre a execução de programas eBPF com o *Hook* XDP *Offload*. Ao longo dos experimentos conduzidos, encontramos inúmeras dificuldades e limitações para carregar os programas eBPF gerados no modo *Offload* com a interface Netronome SmartNIC. Desta forma, de modo a viabilizar a condução da experimentação, definimos reduzir o escopo de experimentos para executar os programas eBPF nos modos XDP *Generic*, XDP *Native* e também utilizamos o novo modo *AF_XDP* que redireciona os pacotes para serem processados por aplicações no espaço do usuário.

Para realização da avaliação de desempenho, utilizou-se os programas disponíveis no tutorial do XDP-Project[3] como base de desenvolvimento. A partir dos exemplos desenvolvidos pelo tutorial, dois programas eBPF genéricos e dois programas para espaço do usuário foram desenvolvidos e utilizados ao longo dos experimentos. Como base de comparação (baseline), utilizou-se um programa eBPF cujo objetivo é apenas receber pacotes da rede pela fila RX (entrada) e reencaminhar esses pacotes para a fila TX (saída) da interface de rede, sem realizar qualquer tipo de tratamento dos pacotes. O programa eBPF é descrito na Figura 10. Esse programa eBPF baseline é necessário para entender o desempenho e os limites máximos alcançáveis pela SmartNIC e o servidor DUT.

```
1 SEC("xdp_pass")
2 int xdp_pass_func(struct xdp_md *ctx){
3     return XDP_TX;
4 }
```

Figura 10 – Programa eBPF base-line.

Na Figura 11, ilustra-se a base do segundo programa eBPF desenvolvido para servir como referência para a criação de outros *nove* programas eBPF que possuem comparações diferentes na *linha 8*, variando de *1 a 12800*. Esses programas possuem a mesma

[3] <https://github.com/xdp-project/xdp-tutorial>

semântica, porém cada um possui diferentes complexidades em termos da quantidade de instruções executadas em cada programa eBPF. Portanto, quanto maior for o valor comparado na *linha 8*, maior será o número de instruções executadas por este programa para cada pacote recebido.

```
1  SEC("xdp_pass")
2  int xdp_pass_func(struct xdp_md *ctx){
3          int var= 1;
4          __u32 *pkt_count;
5          int index= ctx->rx_queue_index;
6          goto out;
7  out:
8          if(var <= 1){
9                  var= var+1;
10                 pkt_count = bpf_map_lookup_elem(&xdp_stats_map, &
       index);
11                 goto out;
12         }
13         return XDP_TX;
14 }
```

Figura 11 – Código base para programas eBPF com laço de repetição.

Para processar pacotes de rede com programas que utilizam o soquete *AF_XDP* diretamente no espaço do usuário, necessita-se de um programa eBPF específico executando na interface de rede (ilustrado na Figura 12). Esse programa eBPF recebe os pacotes de rede na interface e redireciona-os diretamente para o programa *AF_XDP* no espaço do usuário, sem passar pelas pilhas de rede do kernel.

```
1  SEC("xdp_sock")
2  int xdp_sock_prog(struct xdp_md *ctx){
3      int index = ctx->rx_queue_index;
4      if (bpf_map_lookup_elem(&xsks_map, &index))
5          return bpf_redirect_map(&xsks_map, index, 0);
6
7      return XDP_PASS;
8  }
```

Figura 12 – Programa eBPF redirecionador do AF_XDP.

De acordo com a Figura 13, o primeiro programa *AF_XDP* desenvolvido recebe os pacotes no espaço do usuário. Esse programa baseline é necessário para entender o desempenho e os limites máximos alcançáveis pelo modo *AF_XDP* no espaço do usuário. Cada pacote recebido no espaço do usuário necessita de alterações em seu cabeçalho para ser devolvido para a interface de rede e posteriormente ser devolvido para o servidor MoonGen. Sendo assim, precisamos trocar as informações de IP origem/destino e MAC

origem/destino de cada pacote. Posteriormente, ao configurar as informações de cada pacote, precisamos realizar uma chama ao kernel para retornar este pacote para a interface de rede. Após o pacote chegar na interface, o mesmo é devolvido pela SmartNIC para o servidor MoonGen. Assim, se faz todo o processamento dos pacotes pelo modo *AF_XDP* no espaço do usuário.

```
1  static bool process_packet(struct xsk_socket_info *xsk, uint64_t
       addr, uint32_t len){
2          uint8_t *pkt= xsk_umem__get_data(xsk->umem->buffer,addr);
3          int ret;
4          uint32_t tx_idx= 0;
5          uint8_t tmp_mac[ETH_ALEN];
6          struct in_addr tmp_ip;
7          struct ethhdr *eth= (struct ethhdr *) pkt;
8          struct iphdr *ipv4= (struct iphdr *) (eth + 1);
9
10         memcpy(tmp_mac, eth->h_dest, ETH_ALEN);
11         memcpy(eth->h_dest, eth->h_source, ETH_ALEN);
12         memcpy(eth->h_source, tmp_mac, ETH_ALEN);
13         memcpy(&tmp_ip, &ipv4->saddr, sizeof(tmp_ip));
14         memcpy(&ipv4->saddr, &ipv4->daddr, sizeof(tmp_ip));
15         memcpy(&ipv4->daddr, &tmp_ip, sizeof(tmp_ip));
16
17         ret = xsk_ring_prod__reserve(&xsk->tx, 1, &tx_idx);
18         if(ret != 1){return false;}
19
20         xsk_ring_prod__tx_desc(&xsk->tx, tx_idx)->addr= addr;
21         xsk_ring_prod__tx_desc(&xsk->tx, tx_idx)->len= len;
22         xsk_ring_prod__submit(&xsk->tx, 1);
23         return true;
24 }
```

Figura 13 – Programa AF_XDP base-line.

O segundo programa *AF_XDP* desenvolvido para servir como base para a criação de outros *nove*, possui o mesmo código fonte que o ilustrado na Figura 13, porém, entre as linhas *15* e *17* o código ilustrado na Figura 14 é adicionado para gerar as derivações de laços de repetição variando de *1 a 12800*. Cada um desses programas possui a mesma sintaxe, porém cada um possui diferentes complexidades, ou seja, estamos analisando a complexidade por quantidade de instruções executadas em cada programa *AF_XDP* no espaço do usuário, da mesma maneira que para programas eBPF.

Os algoritmos visualizados nas Figuras 10 e 11 fazem parte dos programas eBPF que foram executados nos modos XDP *Generic* e *Native* da interface SmartNIC. Já os algoritmos dos programas *AF_XDP* (no espaço do usuário) foram executados no modo XDP *Generic* e estão representados nas Figuras 13 e 14. O código fonte dos programas

```
int i=0;
for(i=0; i < 1; i++){
    memcpy(eth->h_source, tmp_mac, ETH_ALEN);
}
```

Figura 14 – Laço de repetição para programa AF_XDP.

eBPF e *AF_XDP* desenvolvidos neste trabalho podem ser acessados no repositório[4].

3.1.3 Execução dos experimentos

Em nossa avaliação, foram executados dois grandes conjuntos de experimentos. Executamos um total de 480 experimentos, somente para programas eBPF e 240 experimentos, para programas *AF_XDP*.

A seguir, lista-se os 480 experimentos conduzidos para programas eBPF:

1. Foram executados 10 diferentes programas eBPF:
   - Um programa baseline que somente retorna cada pacote para o gerador de pacotes.
   - Nove programas que processam cada pacote executando um laço de repetição variando de 1 até 12800 para depois retornar o pacote.
2. Quatro modos de filas TX/RX de processamento (1, 2, 4 e 8) foram configuradas para executar os programas.
3. Os programa eBPF foram executados nos *Hooks* XDP *Generic* e *Native*.
4. Cada programa eBPF processou 6 diferentes tipos de tráfego IPv4 com tamanhos de pacotes variando entre 64B, 128B, 256B, 512B, 1024B e 1500B. Em todos os experimentos, injetou-se 10Gbit/s de pacotes considerando os tamanhos de pacotes mencionados.

Em seguida, lista-se os 240 experimentos para programas *AF_XDP*:

1. Foram executados 10 diferentes programas *AF_XDP*:
   - Um programa baseline que somente troca as informações de IP e MAC no cabeçalho de cada pacote e em seguida retorna cada pacote para o gerador.
   - Nove programas que processam cada pacote executando um laço de repetição variando de 1 até 12800 para depois retornar o pacote.
2. Quatro modos de filas TX/RX de processamento (1, 2, 4 e 8) foram configuradas para executar os programas.

[4] <https://github.com/igor-capeletti/tcc_eBPF_XDP>

3. Os programa *AF_XDP* foram executados no espaço do usuário para processar os pacotes de rede vindos do *Hook* XDP *Generic*.

4. Cada programa *AF_XDP* processou 6 diferentes tipos de tráfego IPv4 com tamanhos de pacotes variando entre 64B, 128B, 256B, 512B, 1024B e 1500B. Em todos os experimentos, injetou-se 10Gbit/s de pacotes considerando os tamanhos de pacotes mencionados.

A Figura 15 ilustra todas as etapas envolvidas na realização dos experimentos com os programas eBPF e *AF_XDP* em nosso ambiente de avaliação. Cada servidor possui uma interface de rede com duas portas físicas, além de outra interface de rede para acesso local/externo. A placa de rede com acesso à Internet/rede local é utilizada apenas para que nosso servidor DUT envie uma chamada via SSH para o servidor MoonGen habilitar o gerador de tráfego pela placa SmartNIC. A placa SmartNIC do servidor DUT é responsável por receber os pacotes de rede vindos do MoonGen e processar todos com um programa eBPF ou *AF_XDP*. Esses pacotes processados pelo servidor DUT são retornados para o servidor MoonGen.

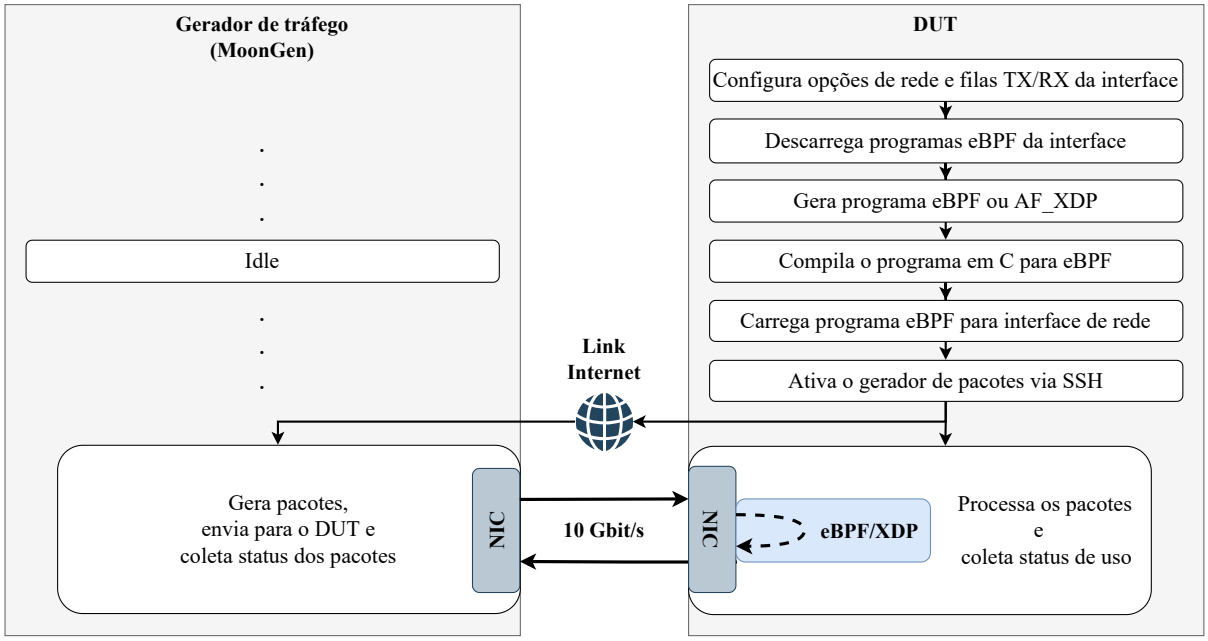

Figura 15 – Ilustração de cada experimento realizado.

Para cada experimento, precisa-se realizar algumas etapas de configurações no servidor DUT e algumas no gerador de tráfego. Ao iniciar um experimento, na primeira etapa é configurado o número de filas TX/RX de processamento de pacotes da interface, variando entre 1, 2, 4 e 8 filas (o limite máximo da interface da Netronome é 8 filas). Em seguida, configura-se os endereço IP da interface SmartNIC. Na segunda etapa são descarregados todos os programa eBPF que estão em execução em qualquer um dos *Hooks* XDP da interface SmartNIC.

Na terceira etapa, gera-se o programa em linguagem *C* que será utilizado no experimento. Para agilizar o processo de criação, desenvolveu-se um gerador para criar programas em linguagem *C* de maneira automatizada que considera o tipo de algoritmo que desejamos criar e também o tamanho do laço de repetição. Após a criação, o programa em *C* é movido para um diretório específico dentro da pasta do projeto XDP-Project[5] para facilitar a compilação, o qual já conta com a biblioteca libbpf[6] instalada.

A quarta etapa é ilustrada na Figura 16. Nela, primeiro o programa em linguagem *C* é compilado para linguagem de máquina (bytecode). Em seguida, o kernel verifica se este bytecode não possui acessos incorretos na memória e se não possui loops indefinidos. Se tudo estiver correto com a verificação, o bytecode é compilado para linguagem eBPF. Para a criação de programas *AF_XDP*, não é necessário realizar esta etapa de compilação, pois o programa em *C* criado na etapa anterior é reconhecido nas aplicações *AF_XDP* no espaço do usuário.

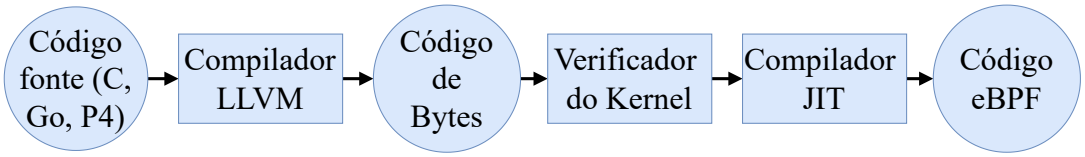

Figura 16 – Compilação do programa em C para eBPF.

Após a compilação do programa eBPF ou *AF_XDP*, na quinta etapa é realizado o carregamento deste programa para a placa de rede, para o espaço do kernel ou para o espaço do usuário. As três formas de carregamento/execução dos programas utilizadas nos experimentos estão ilustradas na Figura 17.

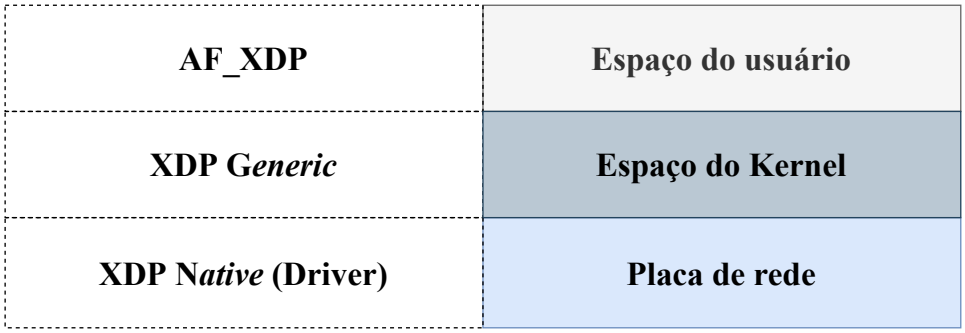

Figura 17 – Modos de carregamento dos programas.

Os programas em eBPF foram executados no XDP *Generic* e XDP *Native*. Já os programas em *AF_XDP* foram executados no espaço do usuário. Porém, para que um programa *AF_XDP* funcione normalmente, primeiro um programa eBPF específico é carregado para o XDP *Generic* do sistema. Cada pacote de rede recebido pela interface de rede é redirecionado por esse programa eBPF para ser processado no espaço do usuário.

[5] <https://github.com/xdp-project/xdp-tutorial>
[6] <https://github.com/libbpf/libbpf>

Esta forma de redirecionamento faz uma ligação por meio do soquete *AF_XDP* entre os *buffers* da interface de rede com o espaço do usuário, sem passar pelas pilhas de processamento de rede do kernel – o que diminui o overhead intrínseco do próprio Sistema Operacional.

Após o carregamento do programa eBPF para a interface de rede do servidor DUT, na sexta etapa é enviada uma chamada via SSH para o servidor gerador de tráfego. Nesta chamada via SSH, são passados alguns parâmetros para ativar a geração dos pacotes pela interface SmartNIC. Esses parâmetros informam o gerador sobre o tamanho fixo em Bytes desses pacotes, a variação de aleatoriedade dos endereços IPs/MACs de destino desses pacotes, o tempo de execução do gerador e ainda, o arquivo em que são salvas as informações sobre os pacotes enviados/recebidos de cada execução do gerador. Este arquivo é utilizado posteriormente para avaliar o desempenho da execução.

Na última etapa, os dois servidores (MoonGen e DUT) executam diferentes tarefas ao mesmo tempo. Do lado do servidor MoonGen, gera-se 10 Gbit/s de trafego através da SmartNIC de acordo com as informações instruídas. Em paralelo, a cada segundo o próprio MoonGen coleta informações referentes sobre a quantidade de pacotes enviados/-recebidos, a Taxa de Transferência de envio/recepção e também a Latência dos pacotes.

Do lado do servidor DUT, como o programa eBPF ou *AF_XDP* já está carregado na interface de rede, ao receber os pacotes vindos do servidor MoonGen, eles são então processados pelo programa eBPF ou *AF_XDP*. Em paralelo, utilizamos as ferramentas *Perf*[7] e *Sar*[8] para coletar diversas métricas de desempenho do servidor DUT durante o processamento dos pacotes com os programas eBPF ou *AF_XDP*. Essas informações coletadas são discutidas no próximo capítulo.

3.1.4 Automatizador de experimentos

Para executar apenas um experimento, é necessário realizar várias etapas. Para tornar mais rápida e personalizável a execução de cada experimento, desenvolvemos um *script*[9] para automatizar todas essas etapas. Com este *script* também conseguimos executar vários experimentos sequencias durante um grande período de tempo e ao mesmo tempo coletar todas as informações necessárias para as análises de desempenho do Capítulo Resultados Experimentais. Através deste automatizador conseguimos personalizar o número de filas TX/RX de processamento, o tipo de programa (eBPF ou *AF_XDP*), o tipo de algoritmo do programa, o modo de *Hook* XDP que o programa será carregado na interface de rede e também qual tipo de tráfego de pacotes desejamos enviar com o MoonGen.

[7] <https://perf.wiki.kernel.org>
[8] <https://github.com/sysstat/sysstat>
[9] <https://github.com/igor-capeletti/tcc_eBPF_XDP/blob/main/scripts/automatizador_experimentos.sh>

Cada experimento demora cerca de 120 segundos para ser executado. Cerca de 20 segundos são utilizados para executar as seis primeiras etapas da Figura 15. O gerador de tráfego demora cerca de 10 segundos para iniciar e 60 segundos para executar as 30 gerações de tráfego. Nesses 60 segundos da geração de tráfego, nosso servidor DUT processa os pacotes recebidos com um programa eBPF ou *AF_XDP*. Os segundos restantes estão relacionados com mensagens de informação e tempo de espera para inicializar o próximo experimento da sequência. A cada 12 experimentos reiniciamos o servidor do MoonGen por causa de problemas que acabam ocorrendo com o gerador de tráfego quando são executados muitos experimentos. Essa reinicialização também é executado de maneira automática pelo nosso *script* de automatização. Então, a cada 12 experimentos são adicionados mais 130 segundos por reinicialização do servidor MoonGen.

São necessárias 18 horas aproximadamente para executar o conjunto de 480 experimentos com programas eBPF e cerca de 10 horas para o conjunto com 240 experimentos de programas *AF_XDP*.

3.1.5 Coleta dos dados

Durante cada experimento os dados dos dois servidores são coletados. No servidor responsável pela geração do tráfego são coletados os dados referente a quantidade de pacotes enviados/recebidos, a taxa de envio/recepção, a taxa média de envio/recepção e a Latência média dos pacotes. O próprio MoonGen coletas esses dados a cada segundo que o gerador enviou/recebeu o tráfego de pacotes e salva em um arquivo de *logs*.

No servidor DUT, em que executamos os programas eBPF e *AF_XDP*, utiliza-se as ferramentas *Perf*[10] e *Sar*[11] para coletar as informações durante o processamento dos pacotes recebidos. Com a ferramenta *Sar*, coleta-se os dados relacionados com a percentagem de uso dos cores lógicos do processador. Similarmente, por meio da ferramenta *Perf*, coleta-se as informações como número de instruções por ciclo, número de acertos/erros de desvios (*branches*) e número de acertos/erros de acesso à cache L1. Cada ferramenta coleta os dados durante o processamento dos pacotes e salvam os dados em arquivos separados.

[10] <https://perf.wiki.kernel.org>
[11] <https://github.com/sysstat/sysstat>

4 RESULTADOS EXPERIMENTAIS

Neste capítulo, descreve-se os resultados para as métricas de Taxa de Transferência, Latência e Uso de CPU obtidos a partir da metodologia proposta no Capítulo 3.

4.1 Taxa de Transferência

A Figura 18 ilustra os resultados obtidos ao processar diferentes tamanhos de pacotes, em cada modo XDP, para diferentes quantidades de filas TX/RX de processamento. Esses dados são referentes ao algoritmo *baseline*. Desta forma, os resultados ilustrados representam os melhores valores alcançados para cada modo XDP, por quantidade de filas de processamento e também por tamanho de pacote.

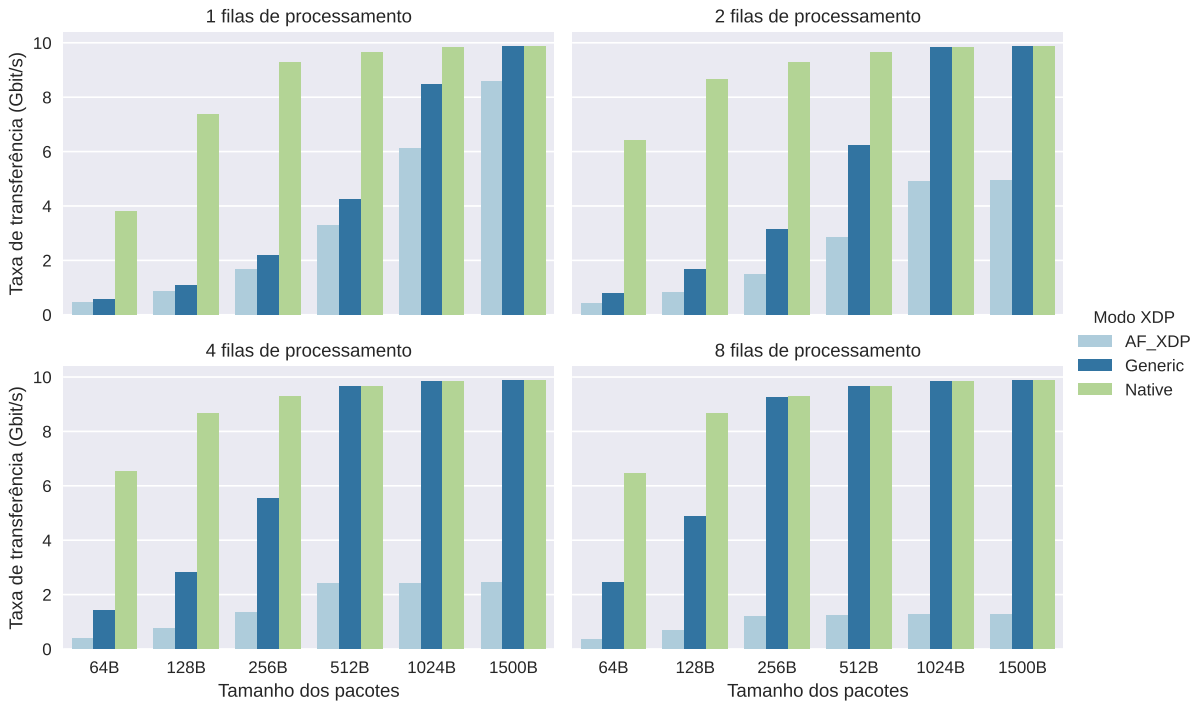

Figura 18 – Taxa de Transferência dos modos XDP para diferentes tamanhos de pacote (algoritmo baseline).

Observa-se, na Figura 18, que o XDP *Native* obteve as melhores taxas de transferência para processar todos os tamanhos de pacotes. Ao processar pacotes pequenos, o XDP *Native* obteve taxas superiores quando comparado aos modos *AF_XDP* e XDP *Generic*. Para pacotes grandes, em grande parte dos casos, o XDP *Generic* conseguiu alcançar taxas similares ao XDP *Native*. Nos modos *AF_XDP* e XDP *Generic* as taxas de transferência aumentam gradativamente conforme aumenta-se o tamanho dos pacotes. Em relação à quantidade de filas RX/TX, observa-se que os modos XDP *Generic* e *Native* obtiveram o melhor desempenho ao utilizar mais filas TX/RX de processamento. Já o modo *AF_XDP* obteve seus melhores resultados ao processar pacotes com apenas uma

fila. Ao aumentar o número de filas, o desempenho do *AF_XDP* decai, principalmente ao comparar a Taxa de Transferência entre os modos XDP para cada tamanho de pacote.

A seguir, nas Figuras 19, 20 e 21, apresenta-se os dados referentes à Taxa de Transferência obtida para cada fila de processamento ao variar os algoritmos de acesso à memória em cada modo XDP. A Figura 19 refere-se a pacotes de 64 Bytes, enquanto que a Figura 20 para pacotes de 1024 Bytes e a Figura 21 para pacotes de 1500 Bytes.

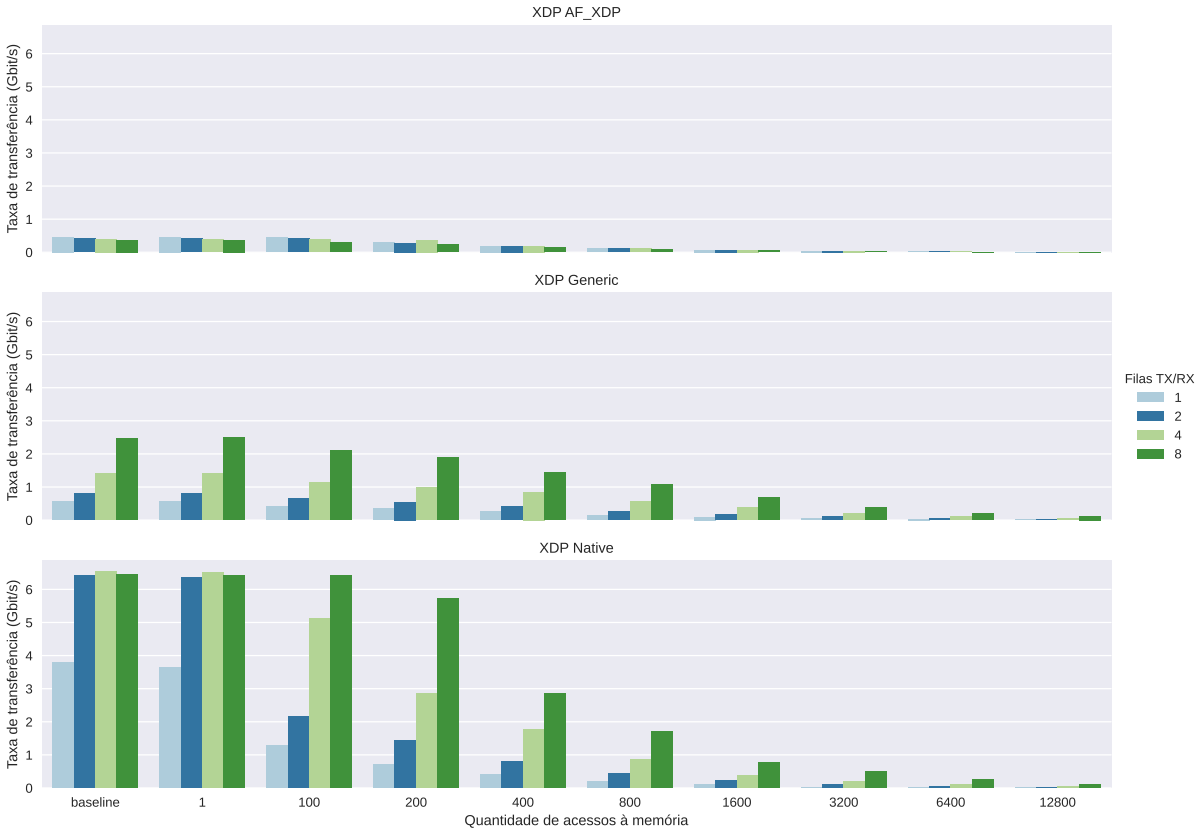

Figura 19 – Taxa de Transferência das filas de processamento para diferentes algoritmos (pacotes de 64B).

Em geral, o modo XDP *Native* atinge melhores taxas de transferência para todos os algoritmos analisados. Para cada quantidade de filas, também foi superior aos demais modos XDP. No *baseline* e os algoritmos com poucos acessos à memória, o XDP *Native* obteve taxas de transferência acima de 5 Gbit/s ao utilizar mais de 1 fila de processamento. O XDP *Native* teve quedas consideráveis de Taxa de Transferência ao aumentar a quantidade de acessos à memória, com quedas bem mais acentuadas que os demais modos XDP.

O XDP *Generic* conseguiu manter taxas mais estáveis que o modo *Native* ao aumentar a quantidade de acessos à memória, porém as taxas de transmissão foram limitadas a 3 Gbit/s para todos os experimentos considerados. De todos os modos, o *AF_XDP* obteve as taxas de transferência mais estáveis para quantidade de filas e quantidade de acessos à memória, porém, mostrou o pior desempenho de processamento.

Os modos XDP *Generic* e *Native* tiveram as melhores taxas de transferência para a grande maioria dos algoritmos ao utilizar mais filas de processamento. Já o modo *AF_XDP* obteve melhores taxas ao processar com menores quantidades de filas.

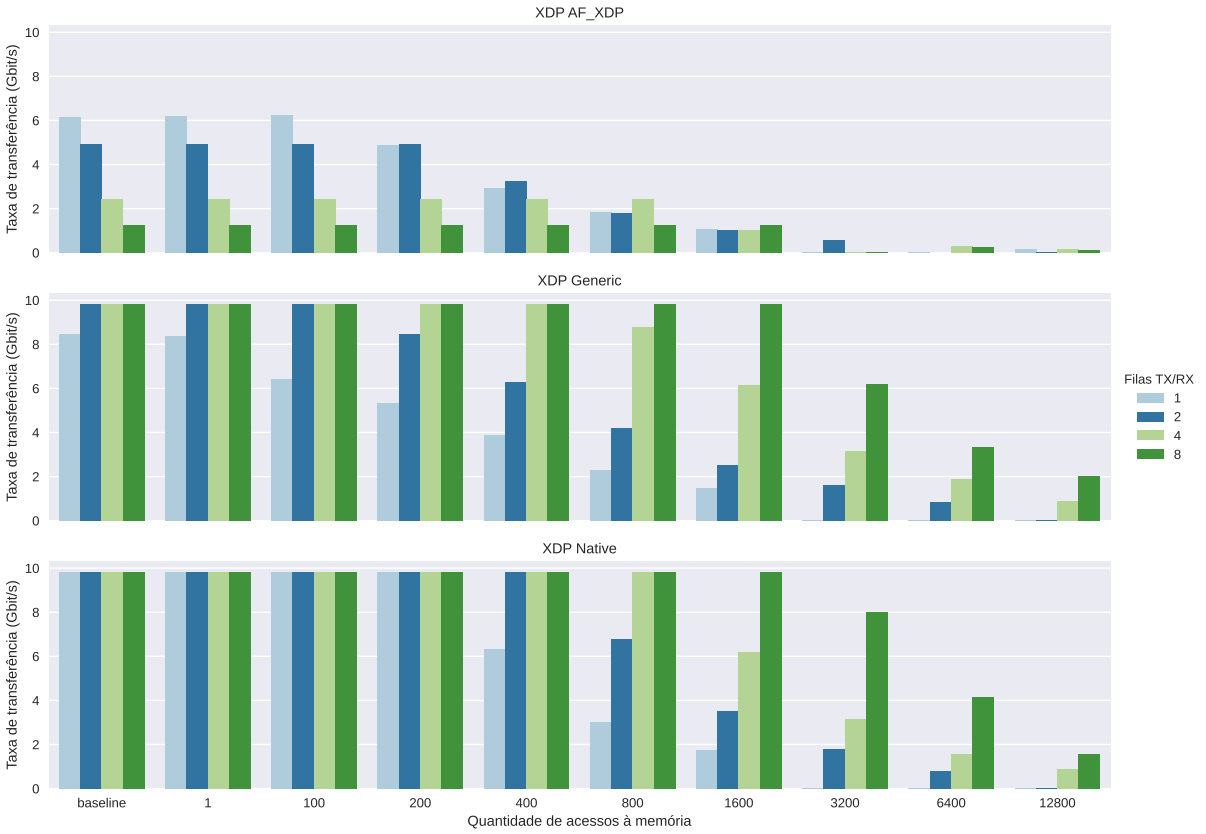

Figura 20 – Taxa de Transferência das filas de processamento para diferentes algoritmos (pacotes de 1024B).

Na Figura 20, em que os pacotes são de 1024 Bytes, as taxas de transferência para todos os modos XDP tiveram melhoras consideráveis, em relação ao experimento anterior (Figura 19). O modo *AF_XDP* alcançou taxas de 500 MBit/s com pacotes de 64 Bytes e 6 GBit/s para pacotes de 1024 Bytes.

Podemos observar que para pacotes de 1024 Bytes os modos XDP *Generic* e *Native* tiveram desempenho próximo ao line-rate ao manter estável o processamento de pacotes com taxas de 10 Gbit/s para a maioria dos algoritmos de acesso à memória. Desta vez, o modo *AF_XDP* conseguiu melhores taxas ao utilizar apenas 1 fila para processar pacotes com algoritmos que realizam poucos acessos à memória. Para algoritmos com maiores acessos à memória, o *AF_XDP* mostrou poucas oscilações de desempenho para diferentes quantidades de filas.

Por meio da análise da Figura 21, para pacotes de 1500 Bytes, percebe-se que as taxas de transferência aumentaram para 8 Gbit/s no modo *AF_XDP*. Os demais modos também mostraram melhoras gerais, mas conseguiram melhorar as taxas de transferência para todas as quantidades de filas e também para algoritmos com mais acessos à memória.

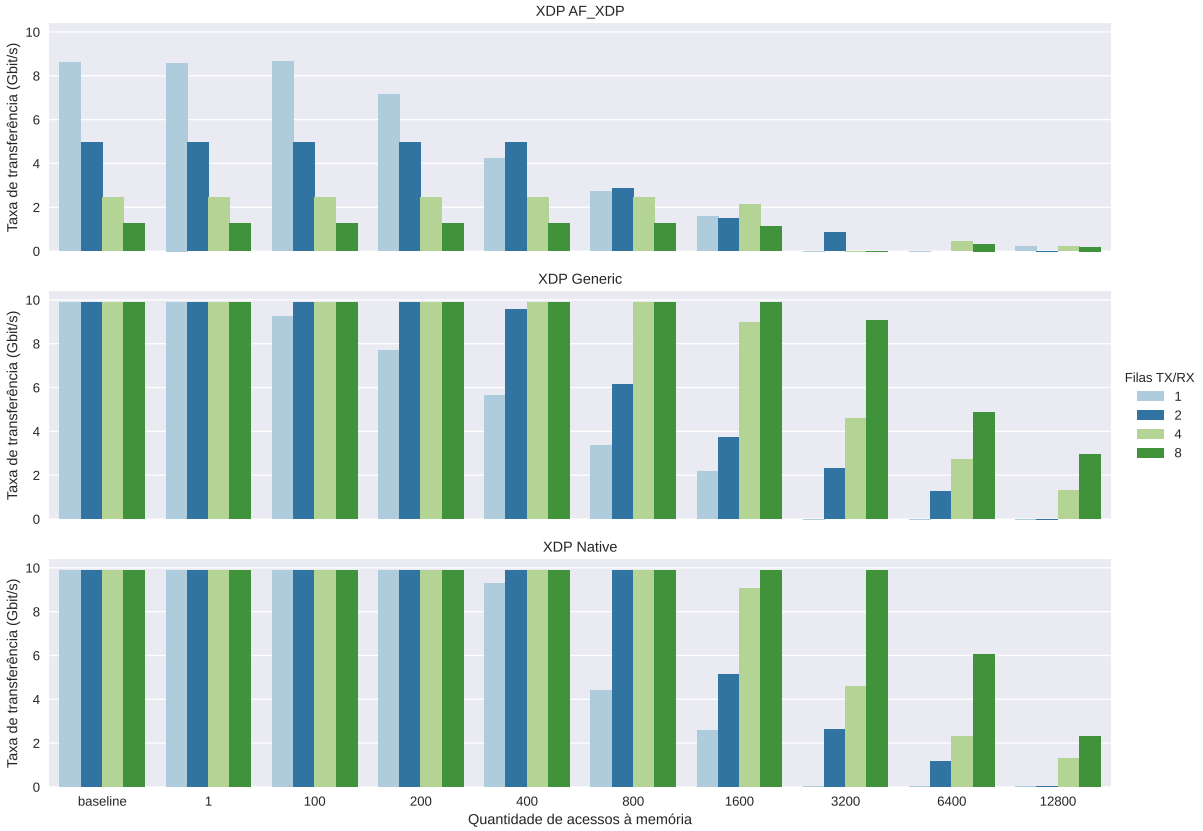

Figura 21 – Taxa de Transferência das filas de processamento para diferentes algoritmos (pacotes de 1500B).

Avaliando as Figuras 19, 20 e 21, percebe-se que a Taxa de Transferência de todos os modos XDP aumentam significativamente quando aumentamos o tamanho dos pacotes. Com pacotes maiores, os modos XDP também conseguiram melhores taxas de transferência para todos os algoritmos avaliados. Na maioria dos casos, ao processarmos pacotes com maior número de filas, obtemos os melhores resultados, excluindo o modo *AF_XDP* que obteve melhores resultados com apenas 1 fila de processamento.

Para gerar 10 Gbit/s de pacotes, são necessários cerca de 14.88 milhões de pacotes de 64 Bytes, 1.3 milhões de pacotes de 1024 Bytes ou 900 mil pacotes de 1500 Bytes. Cada fila está atribuída a um único core da CPU e consegue, desta forma, processar uma determinada quantidade de pacotes por segundo. Portanto, em nossos experimentos, a maioria dos pacotes pequenos são descartados pela impossibilidade de processá-los. Isso acontece devido a grande quantidade de pacotes que chegam mas não existem filas suficientes para realizar o processamento ou também as filas que estão ativas não conseguem fornecer Vazão suficiente. Esses fatores justificam as taxas de transferências obtidas ao processar diferentes tamanhos de pacotes e também ao processar com quantidades diferentes de filas.

Em todos os modos XDP e para todos os tamanhos de pacotes avaliados, quando é aumentado o número de acessos à memória, as taxas de transferência diminuem. Esse

efeito acontece pelo fato que ao aumentar o número de acessos à memória, o número de instruções para cada pacote processado aumenta, assim como, o tempo de processamento.

A seguir, na Figura 22 é ilustrada as melhores taxas de transferência que cada modo XDP obteve para diferentes tamanhos de pacotes variando a quantidade de acessos à memória. Os dados exibidos no modo *AF_XDP* são referentes ao processamento de pacotes com 1 fila TX/RX. Para os modos XDP *Generic* e *Native* foram utilizados os dados referentes ao processamento de pacotes com 8 filas TX/RX. Esses dados foram escolhidos devido ao seu melhor desempenho, como discutido anteriormente.

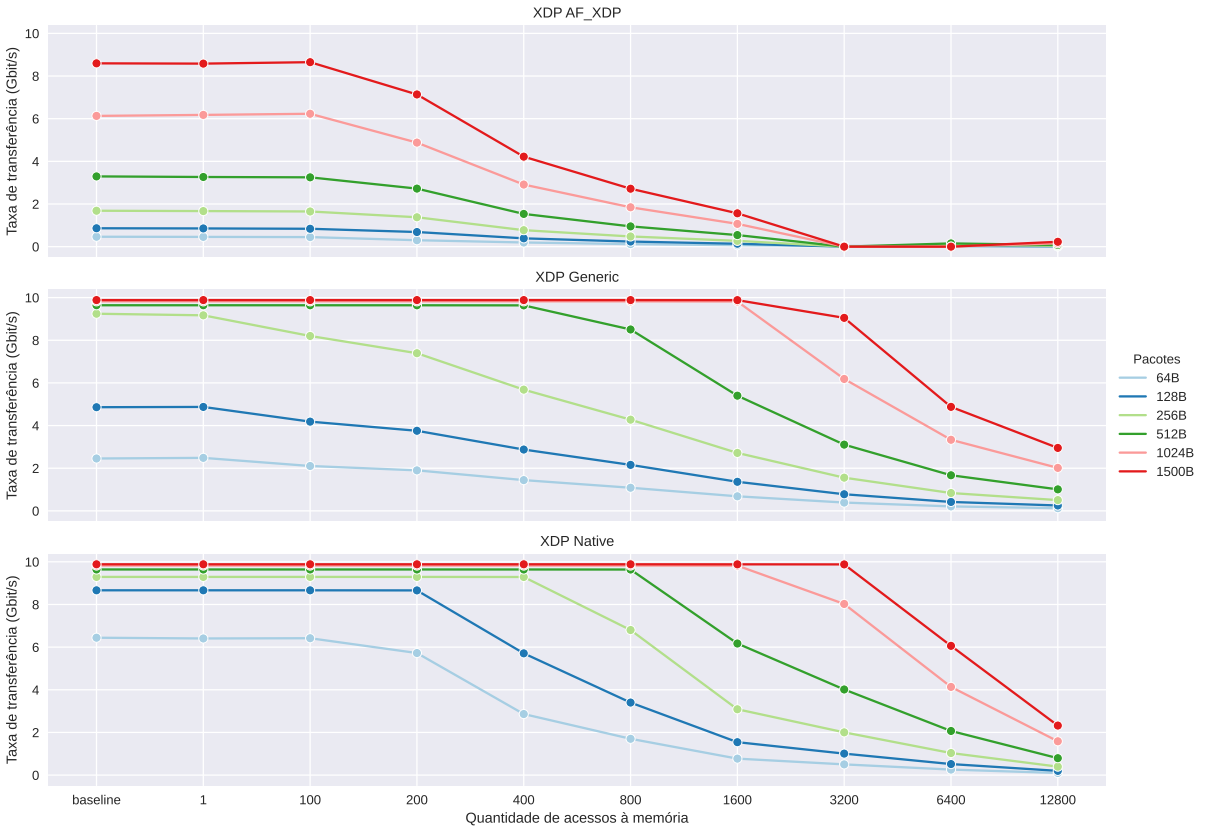

Figura 22 – Taxa de Transferência para diferentes pacotes em diferentes algoritmos.

De acordo com a Figura 22, os modos XDP *Generic* e *Native* obtiveram o melhor desempenho mantendo a Taxa de Transferência em 10 Gbit/s ao processar pacotes desde o programa *baseline* até programas com 3200 acessos à memória. Mesmo abaixo dos demais modos, o *AF_XDP* mostrou um desempenho próximo ao line-rate, conseguindo manter taxas de transferência acima de 8 Gbit/s para o algoritmo *baseline* e algoritmos de até 100 acessos à memória. Dentre todos os modos, o modo XDP *Native* obteve as melhores taxas de transferência para todos os tamanhos de pacotes, em todos os algoritmos de acessos à memória, exceto pelo algoritmo com 12800 acessos.

Ao aumentar a quantidade de acessos à memória, o *AF_XDP* consegue manter quedas mais suaves que os demais modos para todos os tamanhos de pacotes, porém, suas taxas de transferência chegaram próximas à zero quando é realizado mais de 3200 acessos

à memória. Para pacotes pequenos, as taxas de transferência ficam abaixo de 2 Gbit/s.

Os modos XDP *Generic* e *Native* conseguem obter melhores desempenhos devido ao fato de realizar o processamento de pacotes o mais cedo possível no sistema. O XDP *Generic* realiza o processamento dos pacotes nas camadas iniciais da pilha de protocolos de rede do kernel e o XDP *Native* realiza o processamento com o auxílio do driver da placa de rede e o kernel. Dessa forma seus desempenhos são superiores ao *AF_XDP* que processa os pacotes no espaço do usuário.

4.2 Latência

A seguir, nas Figuras 23 e 24 apresentamos as Latências que cada tamanho de pacote e algoritmo impactou nos modos XDP *Native*, *AF_XDP* e *Generic*. Essas Latências são medidas pelo MoonGen para cada geração do tráfego de pacotes. Na Figura 23, os dados para o modo *AF_XDP* foram filtrados para os experimentos que utilizaram apenas 1 fila TX/RX de processamento. Já os modos *Generic* e *Native* foram filtrados para os experimentos que utilizaram 8 filas de processamento. Esse filtro foi realizado para essas filas TX/RX específicas, devido ao melhor desempenho obtido em cada modo XDP nas análises da seção anterior. Na Figura 24 todos os dados são referentes ao processamento de pacotes com 4 filas TX/RX.

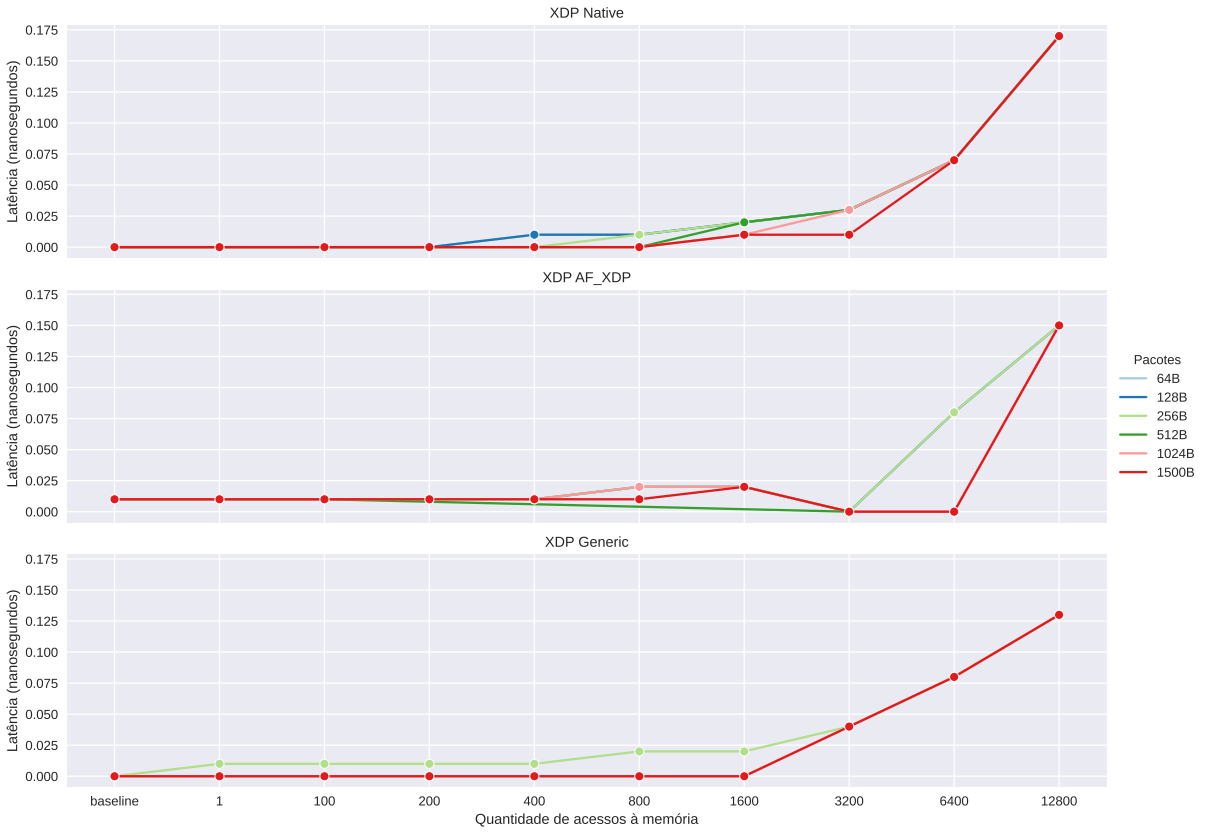

Figura 23 – Latência de processamento para cada tamanho de pacote em diferentes algoritmos (melhor configuração de fila para cada XDP).

Analisando a Figura 23, nota-se que a grande maioria dos experimentos obteve Latência próxima de zero. Isso mostra que todos os modos XDP conseguem processar pacotes com mínima Latência, porém, a Latência aumenta gradativamente conforme aumenta-se o número de instruções de acesso à memória. Em todos os modos XDP, percebe-se que a Latência geral aumenta gradualmente a partir do processamento com mais de 1600 acessos à memória.

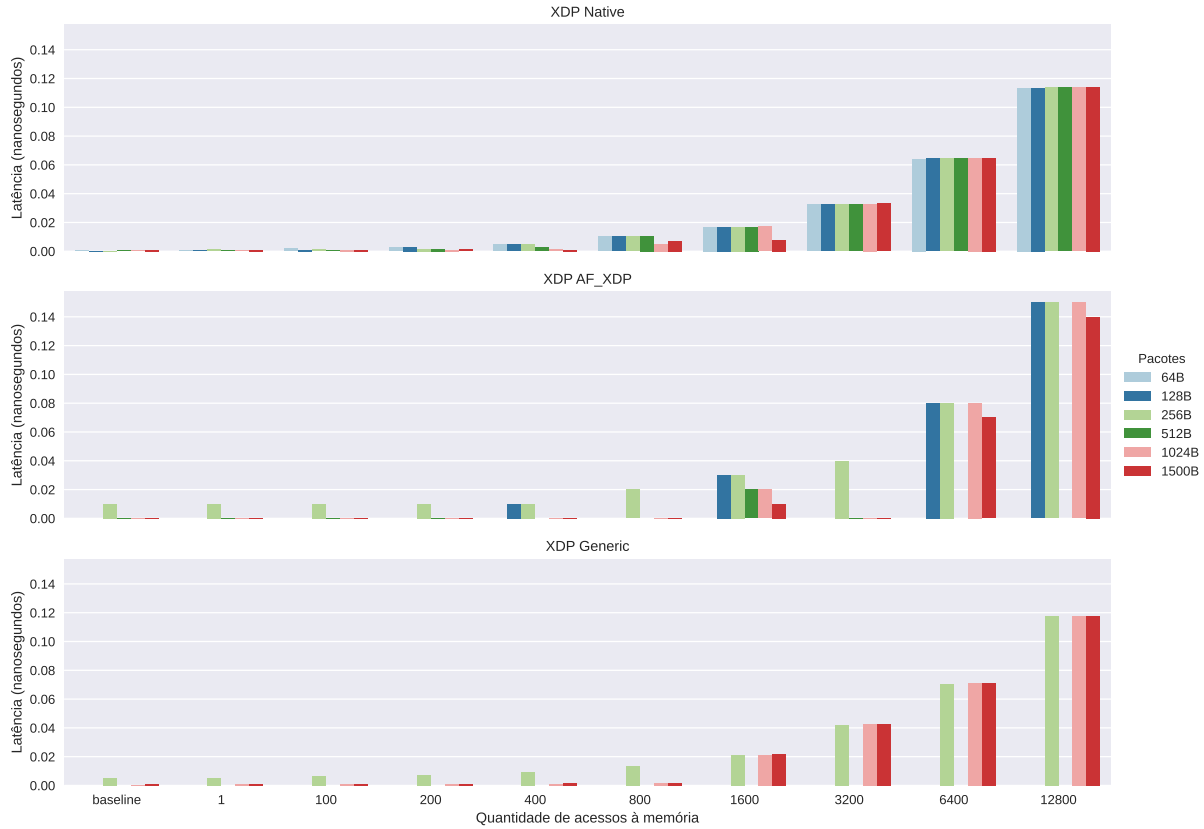

Figura 24 – Latência de processamento para cada tamanho de pacote em diferentes algoritmos (4 filas de processamento).

A Figura 24 ilustra como as Latências ficam muito próximas a zero nanosegundos. Nesta ilustração também percebe-se que conforme são realizados mais acessos à memória, mais tempo de resposta é necessário para processar os pacotes.

Ao analisar as Figuras 23 e 24, percebemos que muitos valores estão zerados. Provavelmente esses valores foram muito pequenos e o coletor do MoonGen não conseguiu medir com precisão necessária. Um dos motivos para pouca precisão é devido ao MoonGen coletar esses dados via software, e não via hardware. A coleta de dados muito sensíveis precisa ser realizada via hardware, porém a placa SmartNIC utilizada nos experimentos não possui suporte via hardware para coleta desses dados.

4.3 Uso de CPU

O servidor DUT utilizado nos experimentos possui 8 núcleos físicos e ao todo 16 núcleos lógicos. No atual cenário a interface de rede SmartNIC consegue suportar apenas 8 filas TX/RX de processamento, por default, somente os primeiros 8 núcleos lógicos serão utilizados para realizar o processamento dos pacotes. Portanto, será exibido apenas os dados dos 8 primeiros núcleos lógicos do servidor DUT.

As Figuras 25, 26 e 27 ilustram o uso dos núcleos do processador do servidor DUT durante o processamento dos pacotes com 8 filas TX/RX de processamento. Cada figura mostra o uso de CPU para os diferentes algoritmos de acesso à memória para cada modo XDP. A Figura 25 refere-se ao processamento de pacotes de 64 Bytes, enquanto que a Figura 26 refere-se à pacotes de 1024 Bytes e a 27 refere-se à pacotes de 1500 Bytes.

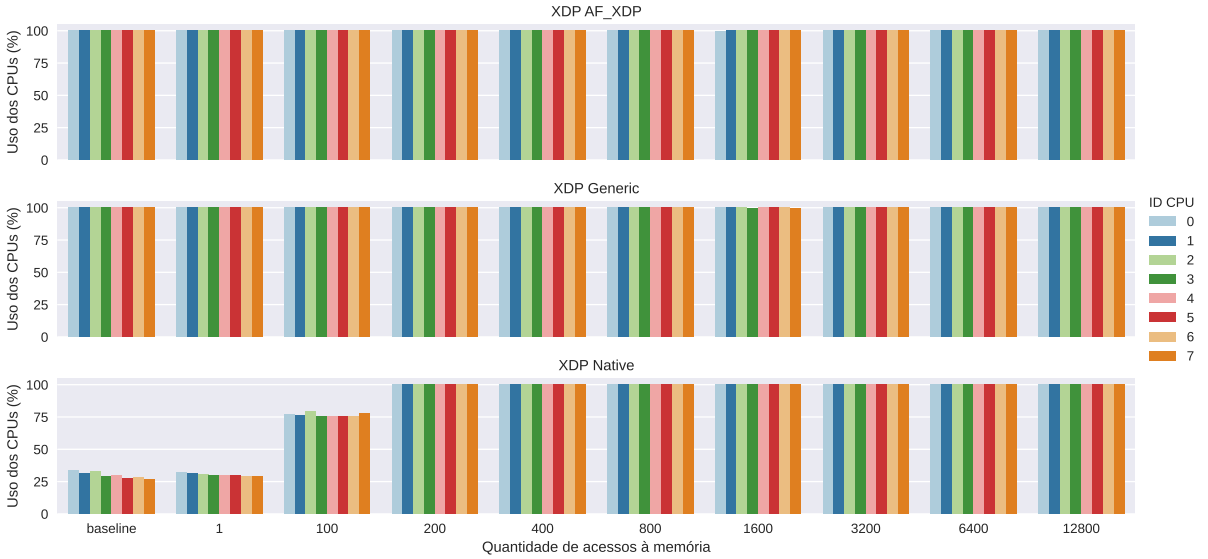

Figura 25 – Uso do CPU para o tratamento de pacotes de 64B com 8 filas de processamento.

Ao analisarmos a Figura 25, percebe-se que nos modos *AF_XDP* e *Generic* o uso dos núcleos permanece sempre em 100% para todos os algoritmos. Apenas o XDP *Native* gera pouca utilização dos núcleos ao processar pacotes com algoritmos de até 100 acessos à memória. No restante dos algoritmos, o uso de todos núcleos ficam em 100% de utilização.

Na Figura 26, o uso dos núcleos para o modo *AF_XDP* manteve-se na maioria dos casos em 100%, somente o núcleo 0 ficou com o uso abaixo dos 40% para todos os algoritmos. Desta vez, os modos *Generic* e *Native* tiveram uso em 100% de todos os seus núcleos a partir dos algoritmos com 3200 acessos à memória, ou seja, para algoritmos com muitas instruções de acesso. Ainda, para os algoritmos com até 1600 acessos, o uso dos núcleos não passou dos 25%. Para o algoritmo com 1600 acessos à memória, o uso chegou na faixa dos 75%.

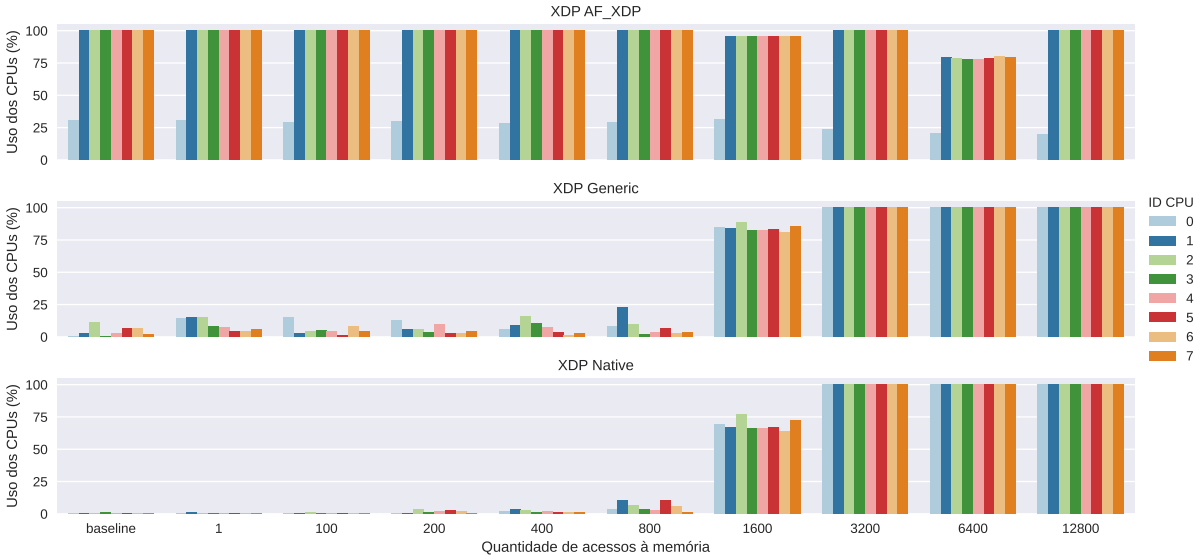

Figura 26 – Uso do CPU para o tratamento de pacotes de 1024B com 8 filas de processamento.

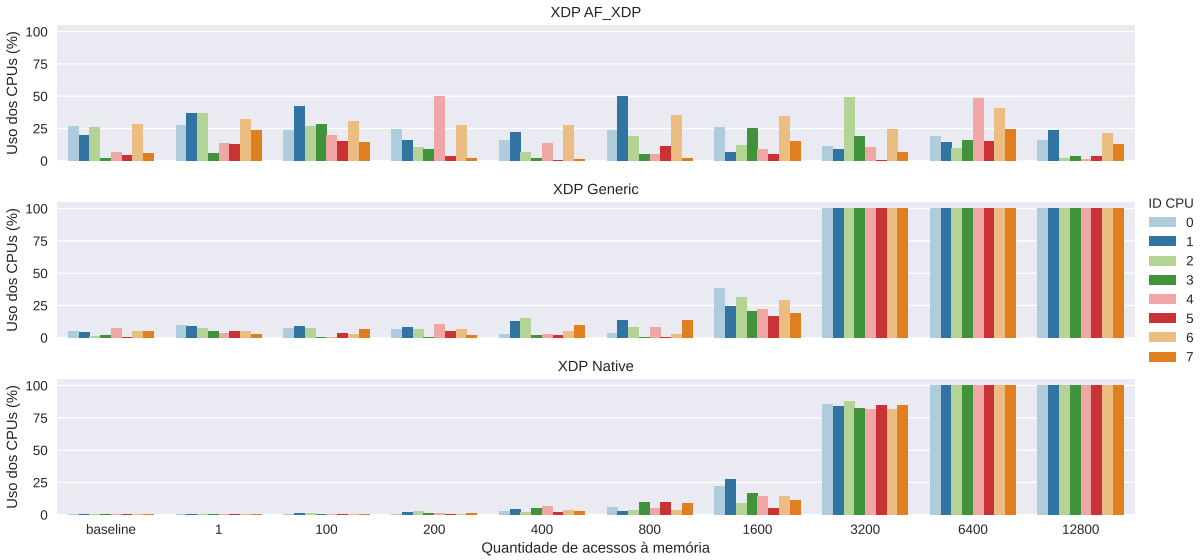

Figura 27 – Uso do CPU para o tratamento de pacotes de 1500B com 8 filas de processamento.

Ao analisarmos as três figuras que mostram o uso do CPU para diferentes tamanhos de pacotes e algoritmos com diferentes quantidades de acesso à memória, observa-se uma melhora no uso dos núcleos para processamento de grandes pacotes. Com pacotes de 1500 Bytes todos os modos XDP tiveram baixas significativas no uso dos núcleos, principalmente o *AF_XDP* em que o uso dos núcleos não passou de 50% para todos os algoritmos de acesso à memória. Para os modos XDP *Generic* e *Native*, o uso dos núcleos não passou dos 15% para processar pacotes com algoritmos de até 800 instruções de acesso à memória.

4.3.1 Instruções

As Figuras 28 e 29 mostram o número de instruções por ciclo de *clock* do processador ao realizar o processamento de pacotes. A Figura 28 ilustra o número de instruções que cada filas TX/RX obteve com diferentes algoritmos ao processar pacotes de 1024 Bytes. Já a Figura 29 ilustra o número de instruções que cada modo XDP utilizou ao processar diferentes tamanhos de pacotes para o algoritmo com 12800 acessos à memória.

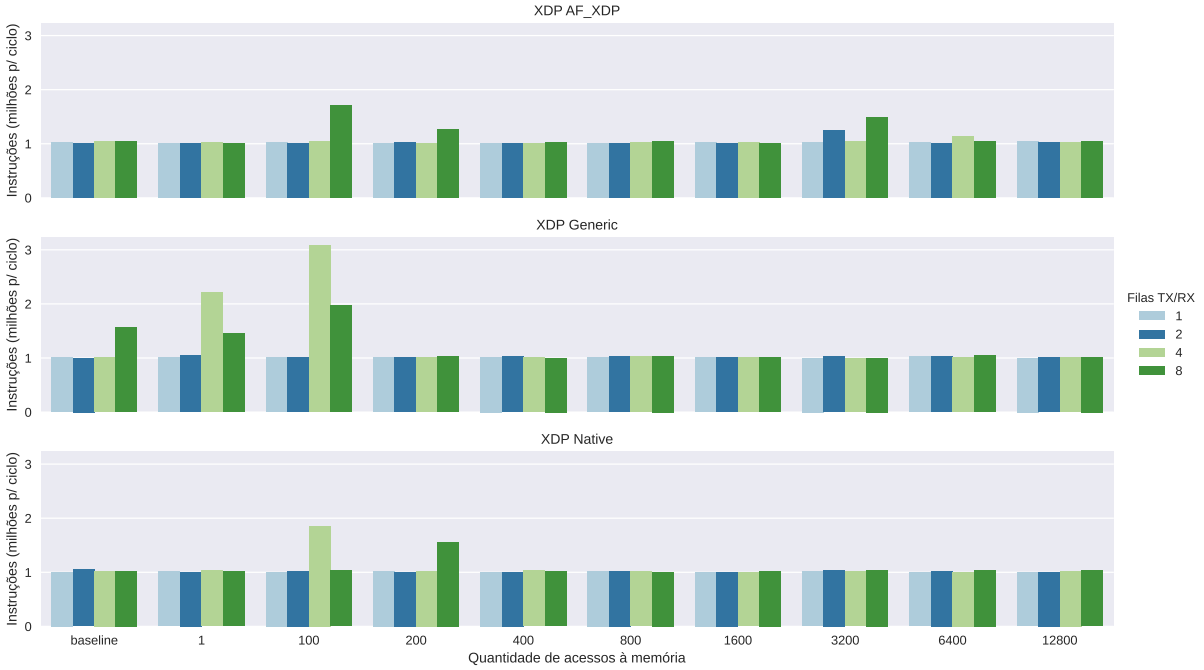

Figura 28 – Número de instruções para cada fila TX/RX ao variar as quantidades de acessos à memória (pacotes de 1024B).

A Figura 28 ilustra que o número de instruções entre os modos XDP não tiveram muitas variações, na maioria dos casos permaneceram iguais. Algumas oscilações quanto ao número de instruções pode estar relacionado com o próprio Sistema Operacional e não com os modos XDP. Ao variar o número de acessos à memória para cada modo XDP, também não ocorreram variações quanto ao número de instruções. Também não ocorreram variações no número de instruções ao utilizar diferentes quantidades de filas TX/RX de processamento.

Com a Figura 29 os dados mostram que ao variarmos o tamanho dos pacotes entre os diferentes modos XDP, o número de instruções não é afetado. Quando variamos a quantidade de filas TX/RX para processar os pacotes também não ocorreram alterações no número de instruções.

Ao analisar as duas Figuras 28 e 29, referentes ao números de instruções que são geradas ao processar pacotes, percebe-se que não obtivemos nenhuma variação significativa ao processar pacotes com diferentes quantidades de filas, com diferentes quantidades de acesso à memória, com os diferentes tamanhos de pacotes e também ao processar os

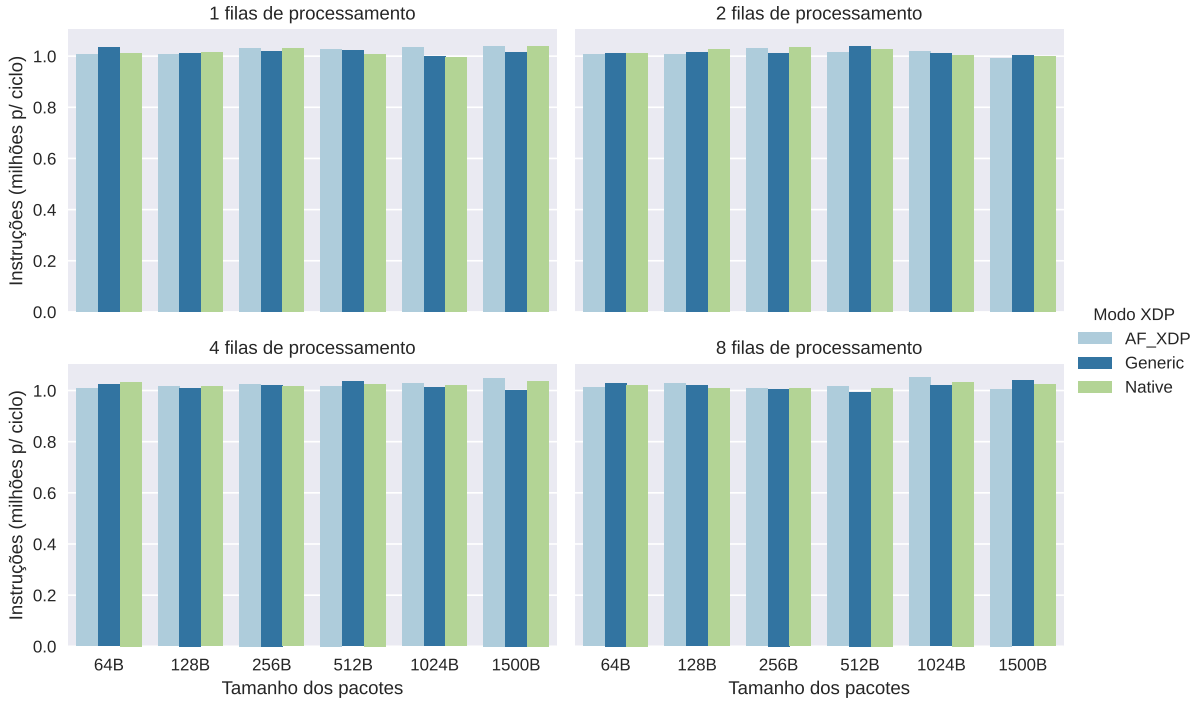

Figura 29 – Número de instruções para cada modo XDP ao processar diferentes tamanhos de pacotes (algoritmo com 12800 acessos a memória).

pacotes com diferentes modos XDP.

4.3.2 Branches

Branches (desvios) são instruções usadas para implementar o fluxo de controle em loops e condicionais de programas. Os branches mudam o fluxo de execução dos programas, ou seja, fazem com que o processador comece a executar uma sequência de instruções diferente somente se certas condições forem satisfeitas, desviando o programa de seu comportamento padrão de execução de instruções em ordem.

A seguir, nas Figuras 30 e 31 são ilustrados os números de branches (desvios) realizados por segundo pelo processador. A Figura 30 referencia o número de branches de diferentes quantidades de filas TX/RX para diferentes algoritmos de acesso à memória, ao processar pacotes de 64 Bytes. A Figura 31 ilustra os dados obtidos nos modos XDP ao processar diferentes tamanhos de pacotes com o algoritmo de 12800 acessos à memória.

Percebe-se que na Figura 30 ocorreram pequenos ruídos aleatórios com o número de branches, mas em praticamente todos os modos XDP não ocorreram varições de branches para diferentes filas de processamento e também para diferentes quantidades de acesso à memória.

Ao analisarmos a Figura 31 percebe-se a estabilidade no número de branches, ou seja, não ocorreram grandes variações entre os modos XDP ao processar diferentes tamanhos de pacote com diferentes filas TX/RX de processamento.

Os resultados das Figuras 30 e 31 mostram que ao processar diferentes tamanhos

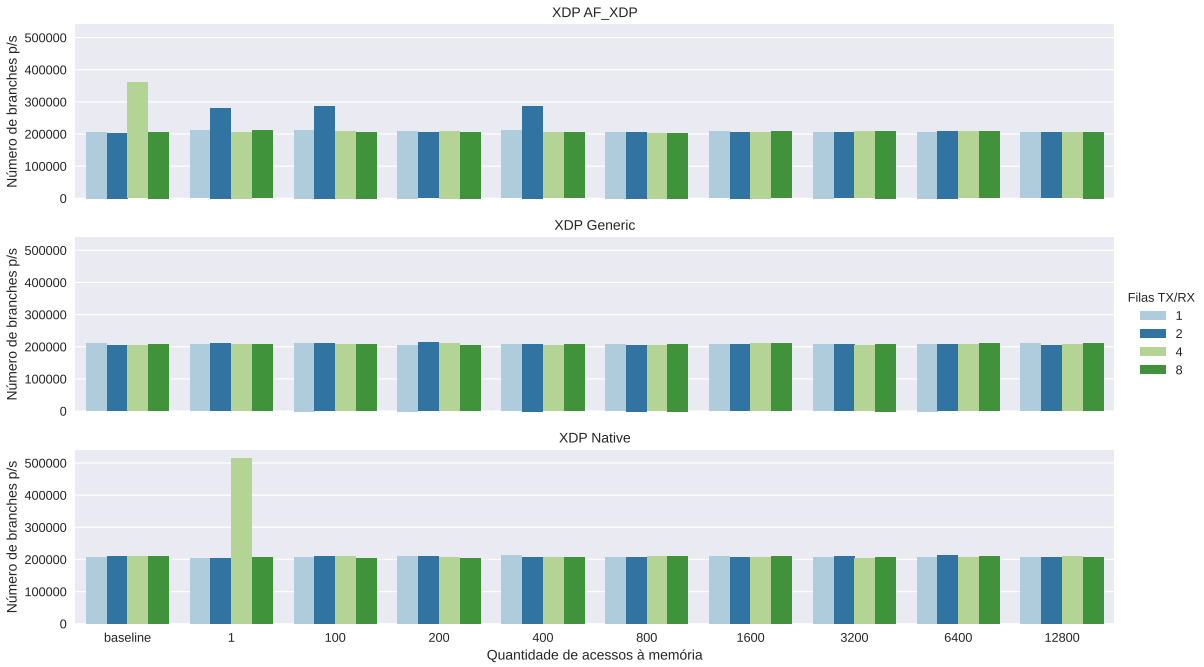

Figura 30 – Número de branches entre filas TX/RX para diferentes algoritmos de acesso à memória (pacotes de 64B).

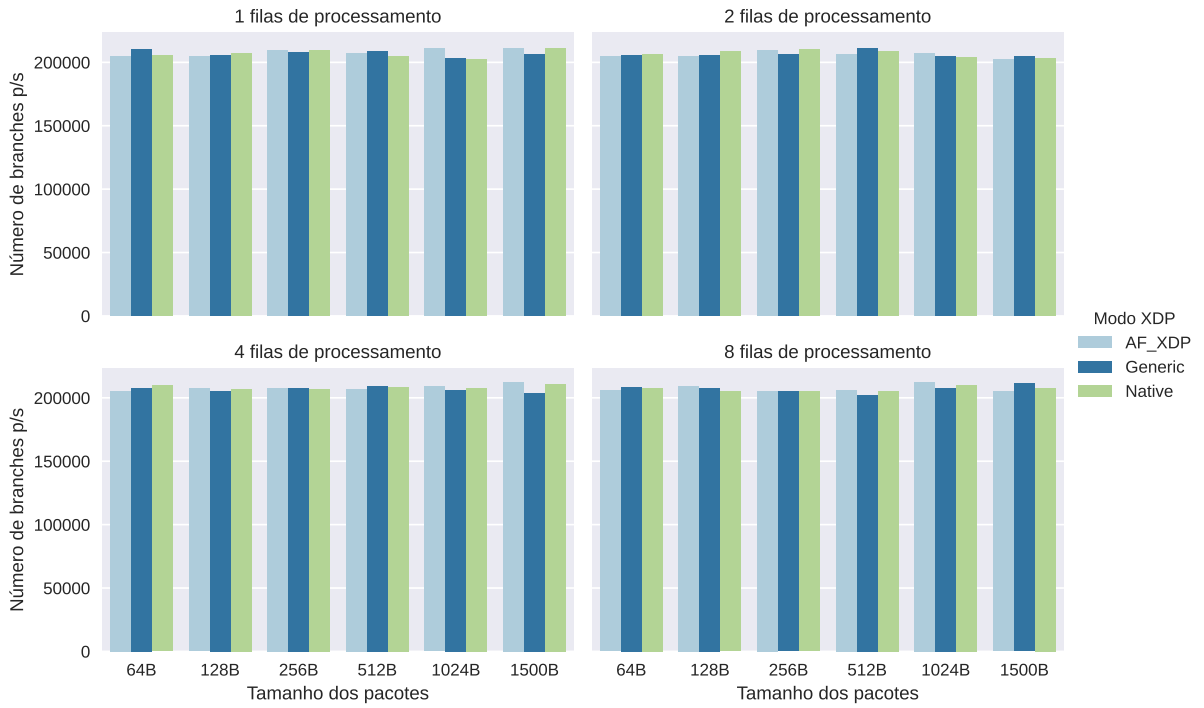

Figura 31 – Número de branches entre modos XDP para diferentes tamanhos de pacote (algoritmo com 12800 acessos à memória).

de pacotes, com diferentes filas TX/RX e também com diferentes algoritmos de acesso à memória, para cada modo XDP avaliado, não ocorrem variações de branches.

4.3.3 Loads

Loads são instruções específicas para realizar operações de leitura/escrita nos registradores e memórias cache L1, L2 e L3 do processador. Existem dois status para cada instrução de Load, Load Hit e Load Miss. O status Load Hit significa que os dados foram encontrados em algum dos registradores ou memórias cache. Load Miss significa que os dados não foram encontrados ou estão incorretos em algum os registradores ou memórias cache. Para nossos experimentos a ferramenta utilizada capturou os dados referentes aos Loads da memória cache L1.

Quando programas precisam alocar/acessar dados, essas instruções são executadas pelo processador. Como os programas dos experimentos acessam e alocam dados, a seguir, nas Figuras 32 e 33 são ilustradas a quantidade de Loads que os programas realizaram durante o processamento de pacotes de 1024 Bytes.

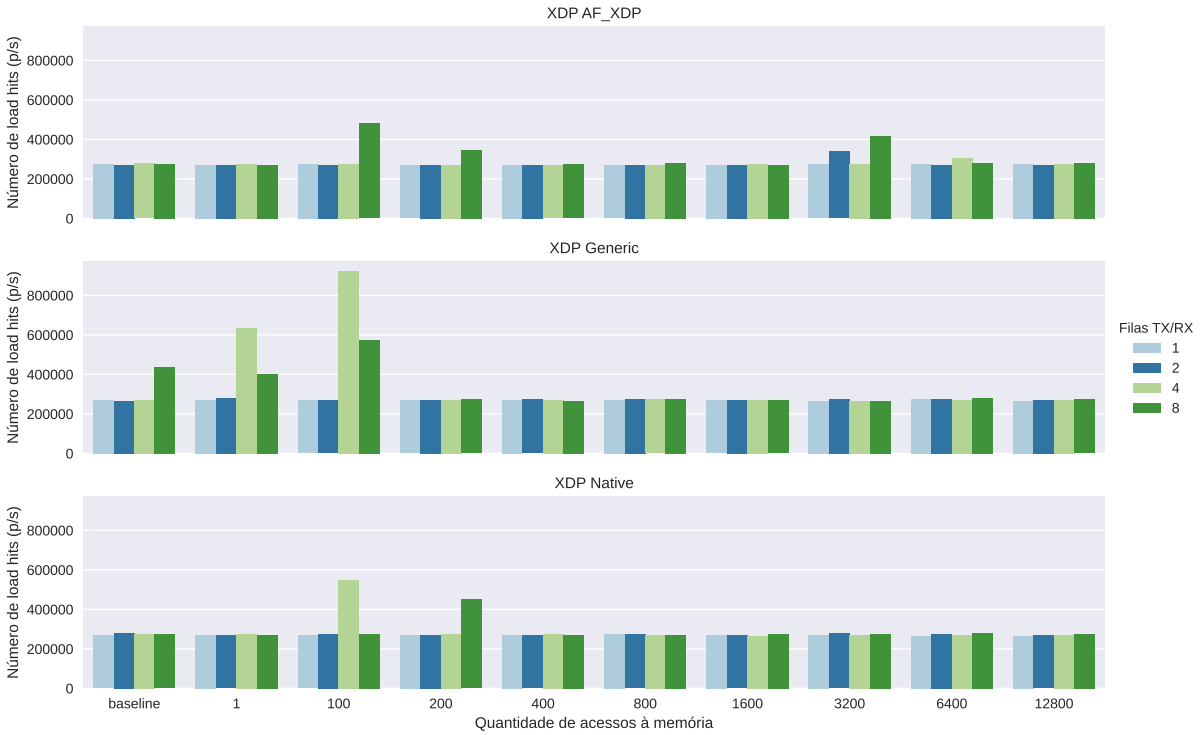

Figura 32 – Load Hits obtidos pelas filas TX/RX ao processar pacotes de 1024 Bytes e variar a quantidade de acesso à memória.

Na Figura 32 percebe-se que todos os experimentos mantiveram a mesma quantidade de Load Hit, com exceção de algumas oscilações para casos aleatórios. Estas oscilações provavelmente estão ligadas ao Sistema Operacional.

A Figura 33 mostra que ocorreram várias oscilações de Load Misses sem um comportamento explicito. Em todos os modos XDP, ao processar pacotes com diferentes quantidades de filas TX/RX e variar o número de acessos à memória, um padrão geral foi percebido. Neste padrão, a maioria dos experimentos mostraram que o número de Load Misses ficou próximo de 20 mil instruções por segundo.

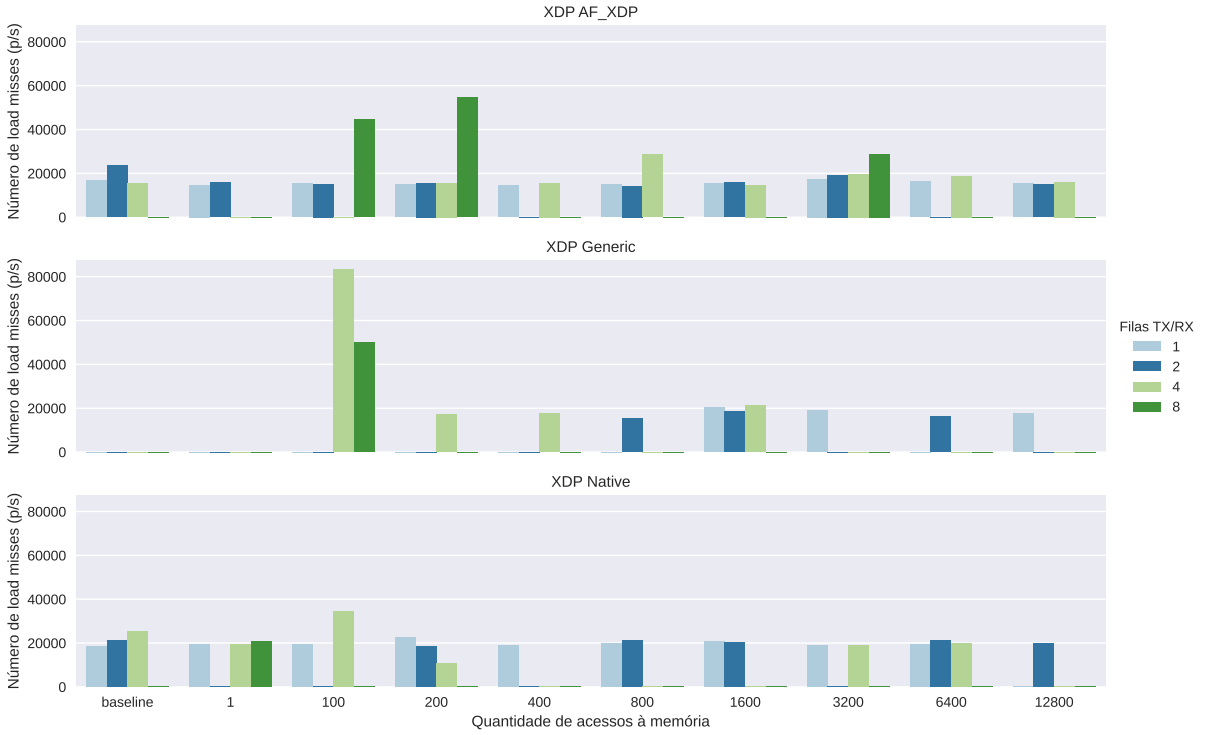

Figura 33 – Load Misses obtidos pelas filas TX/RX ao processar pacotes de 1024 Bytes e variar a quantidade de acesso à memória.

Ao analisar as Figuras 32 e 33, os resultados mostram que o número de Load Hits e Load Misses não possuem comportamentos explícitos, mas sempre um padrão constante com mínima variação entre os Loads para todos os experimentos.

5 CONSIDERAÇÕES FINAIS

Neste trabalho, nós realizamos vários experimentos com programas eBPF/XDP para avaliar métricas de desempenho como Taxa de Transferência, Latência e uso de CPU, ao processar pacotes de rede. Nos experimentos, utilizamos pacotes entre 64 e 1500 Bytes para serem processados por diferentes programas eBPF. Cada programa eBPF realiza uma quantidade específica de acessos à memória para processar cada pacote de rede. Também avaliamos os programas eBPF ao processar pacotes de rede com diferentes quantidades de filas TX/RX de processamento.

Nossos resultados mostram que todos os modos XDP avaliados conseguem excelentes taxas de transferência ao processar grandes pacotes de rede. Os modos XDP *Generic* e *Native* conseguiram manter a Taxa de Transferência máxima da SmartNIC ao processar grandes pacotes com algoritmos de até 3200 acessos à memória. O modo *AF_XDP* também alcançou boas taxas de transferência ao processar grandes pacotes, mas somente para algoritmos com até 100 acessos à memória.

Os resultados para a métrica de Latência mostraram que em todos os modos XDP e para a maioria dos tamanhos de pacotes, ao aumentarmos a quantidade de acessos à memória, o tempo de Latência também aumenta. Os modos XDP mostraram bons desempenhos com mínima variação de Latência para algoritmos com até 1600 acessos à memória.

Quanto ao uso de CPU, todos os modos XDP tiveram baixas taxas de uso dos núcleos ao utilizar mais filas TX/RX para processar tráfegos com grandes pacotes de rede.

Em todos os experimentos realizados, variando o tamanho dos pacotes, variando quantidades de acesso à memória e também variando a quantidade de filas TX/RX de processamento, os resultados mostraram que não ocorreram diferenças de comportamento para as avaliações de número de instruções, número de branches e número de load hits. Para a avaliação de load miss, os resultados mostraram números semelhantes entre todos os experimentos, com algumas oscilações para casos aleatórios. Estas oscilações provavelmente estão ligadas ao Sistema Operacional.

5.1 Trabalhos Futuros

Em nosso trabalho, desenvolvemos programas eBPF/XDP com redirecionamento de pacotes e diferentes quantidades de acessos à memória. Durante nossa avaliação, conseguimos observar vários comportamentos importantes de desempenho e limitações sobre eBPF/XDP ao executar os experimentos com os modos XDP *Generic*, *Native* e *AF_XDP*.

Para trabalhos futuros planejamos desenvolver novos programas com diferentes tipos de instruções e realizar experimentos com a adição do modo XDP *Offload*. Também planejamos executar os experimentos em *AF_XDP* com auxilio dos modos *Native*

e *Offload*. Por último, serão projetados modelos matemáticos visando generalizar o comportamento obtido na avaliação experimental. De modo geral, tais modelos permitirão inferir o desempenho de aplicações de rede baseado nas características observadas.

REFERÊNCIAS

ABRANCHES, M. et al. Efficient network monitoring applications in the kernel with ebpf and xdp. In: **2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)**. [S.l.: s.n.], 2021. p. 28–34. Cited in page 33.

BCC. Xdp compatible drivers. In: . [s.n.], 2019. Disponível em: <https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp>. Cited in page 29.

BERTIN, G. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In: . [S.l.: s.n.], 2017. Cited in page 35.

BERTOLI, G. et al. Evaluation of netfilter and ebpf/xdp to filter tcp flag-based probing attacks. In: . [S.l.: s.n.], 2020. Cited in page 34.

BHAT, R. V. et al. Adaptive transport layer protocols using in-band network telemetry and ebpf. In: **2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)**. [S.l.: s.n.], 2021. p. 241–246. Cited in page 34.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM 14**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Cited in page 11.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/2656877.2656890>. Cited in page 22.

BRUNELLA, M. S. et al. hXDP: Efficient software packet processing on FPGA NICs. In: **14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)**. USENIX Association, 2020. p. 973–990. ISBN 978-1-939133-19-9. Disponível em: <https://www.usenix.org/conference/osdi20/presentation/brunella>. Cited in page 32.

CAESAR, M. et al. Design and implementation of a routing control platform. In: **Proceedings of the 2nd Conference on Symposium on Networked Systems Design Implementation - Volume 2**. USA: USENIX Association, 2005. (NSDI'05), p. 15–28. Cited in page 15.

CARRASCAL, D. et al. Analysis of p4 and xdp for iot programmability in 6g and beyond. **IoT**, v. 1, n. 2, p. 605–622, 2020. ISSN 2624-831X. Disponível em: <https://www.mdpi.com/2624-831X/1/2/31>. Cited in page 31.

CASADO, M.; FOSTER, N.; GUHA, A. Abstractions for software-defined networks. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 57, n. 10, p. 86–95, sep 2014. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/2661061.2661063>. Cited in page 20.

CASEMORE, B. **Northbound API: The standardization debate**. 2012. Disponível em: <https://nerdtwilight.wordpress.com/2012/09/18/northbound-api-the-standardization-debate/>. Cited in page 20.

Castro, A. G. et al. Near-optimal probing planning for in-band network telemetry. **IEEE Communications Letters**, p. 1–1, 2021. Cited in page 11.

CONSORTIUM, P. L. P416 language specification, version 1.2.1. 2020. Disponível em: <https://p4.org/p4-spec/docs/P4-16-v1.2.1.htmll>. Cited in page 22.

COSTA, L. C. et al. Openflow data planes performance evaluation. **Performance Evaluation**, v. 147, p. 102194, 2021. ISSN 0166-5316. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0166531621000110>. Cited in page 32.

DEEPAK, A. et al. **eBPF / XDP based firewall and packet filtering**. 2020. Disponível em: <https://lpc.events/event/2/contributions/100/attachments/99/119/ebpf-firewall-paper-LPC.pdf>. Cited in page 34.

DERI, L.; SABELLA, S.; MAINARDI, S. Combining system visibility and security using ebpf. In: **ITASEC**. [S.l.: s.n.], 2019. Cited in page 33.

DIMOLIANIS, M.; PAVLIDIS, A.; MAGLARIS, V. Signature-based traffic classification and mitigation for ddos attacks using programmable network data planes. **IEEE Access**, v. 9, p. 113061–113076, 2021. Cited in page 35.

DIMOLIANIS, M.; PAVLIDIS, A.; MAGLARIS, V. Syn flood attack detection and mitigation using machine learning traffic classification and programmable data plane filtering. In: **2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)**. [S.l.: s.n.], 2021. p. 126–133. Cited in page 35.

DIX, J. **Clarifying the role of software-defined networking northbound APIs**. 2013. Disponível em: <https://www.networkworld.com/article/2165901/clarifying-the-role-of-software-defined-networking-northbound-apis.html>. Cited in page 20.

DUARTE, L. et al. Sistema de detecção de intrusão serverless em uma smartnic. In: **Anais do XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Porto Alegre, RS, Brasil: SBC, 2021. p. 602–615. ISSN 2177-9384. Disponível em: <https://sol.sbc.org.br/index.php/sbrc/article/view/16750>. Cited in page 35.

EMMERICH, P. et al. Moongen: A scriptable high-speed packet generator. In: **Proceedings of the ACM IMC**. New York, NY, USA: ACM, 2015. (IMC '15), p. 275–287. ISBN 9781450338486. Cited in page 41.

ENBERG, P.; RAO, A.; TARKOMA, S. Partition-aware packet steering using xdp and ebpf for improving application-level parallelism. In: **Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms**. New York, NY, USA: Association for Computing Machinery, 2019. (ENCP '19), p. 27–33. ISBN 9781450370004. Disponível em: <https://doi.org/10.1145/3359993.3366766>. Cited in page 37.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn: An intellectual history of programmable networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 2, p. 87–98, apr 2014. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/2602204.2602219>. Cited in page 16.

FERRO, G. **Northbound API, southbound API, east/-north LAN navigation in an OpenFlow world and an SDN compass**. 2012. Disponível em: <https://etherealmind.com/northbound-api-southbound-api-eastnorth-lan-navigation-in-an-openflow-world-and-an-sdn-compas?doing_wp_cron=1646445782.2627348899841308593750>. Cited in page 20.

FINDLAY, W. **Security Applications of Extended BPF Under the Linux Kernel**. 2020. Disponível em: <https://www.cisl.carleton.ca/~will/written/findlay20bpfsec.pdf>. Cited in page 38.

FINDLAY, W.; SOMAYAJI, A.; BARRERA, D. Bpfbox: Simple precise process confinement with ebpf. In: **Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop**. New York, NY, USA: Association for Computing Machinery, 2020. (CCSW'20), p. 91–103. ISBN 9781450380843. Disponível em: <https://doi.org/10.1145/3411495.3421358>. Cited in page 38.

FRASCARIA, G.; TRIVEDI, A.; WANG, L. A case for a programmable edge storage middleware. **CoRR**, abs/2111.14720, 2021. Disponível em: <https://arxiv.org/abs/2111.14720>. Cited in page 37.

GHORBANI, S.; GODFREY, B. Towards correct network virtualization. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 4, aug 2014. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/2740070.2620754>. Cited in page 20.

GROUP, A. C. S. R. **POSIX**. 2014. Disponível em: <http://standards.ieee.org/develop/wg/POSIX.html>. Cited in page 20.

GUIS, I. **The SDN Gold Rush To The Northbound API**. SDN-Central, 2012. Disponível em: <http://www.sdncentral.com/technology/the-sdn-gold-rush-to-the-northbound-api/2012/11/>. Cited in page 20.

HARKOUS, H. et al. Towards understanding the performance of p4 programmable hardware. In: IEEE. **ACM/IEEE Symposium on Architectures for Networking and Communications Systems**. [S.l.], 2019. p. 1–6. Cited in page 12.

HOHLFELD, O. et al. Demystifying the performance of xdp bpf. In: **2019 IEEE Conference on Network Softwarization (NetSoft)**. [S.l.: s.n.], 2019. p. 208–212. Cited in page 32.

JOLY, C. Evaluation of tail call costs in ebpf. In: . [S.l.: s.n.], 2020. Cited in page 31.

KICINSKI, J.; VILJOEN, N. Hardware offload to smartnics : cls bpf and xdp. In: . [S.l.: s.n.], 2016. Cited in page 32.

KOSTOPOULOS, N.; KALOGERAS, D.; MAGLARIS, V. Leveraging on the xdp framework for the efficient mitigation of water torture attacks within authoritative dns servers. In: **2020 6th IEEE Conference on Network Softwarization (NetSoft)**. [S.l.: s.n.], 2020. p. 287–291. Cited in page 35.

KOSTOPOULOS, N. et al. Mitigation of dns water torture attacks within the data plane via xdp-based naive bayes classifiers. In: **2021 IEEE 10th International Conference on Cloud Networking (CloudNet)**. [S.l.: s.n.], 2021. p. 133–139. Cited in page 36.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14–76, Jan 2015. ISSN 0018-9219. Cited 3 times in the pages 15, 17 e 21.

KRUDE, J. et al. Determination of throughput guarantees for processor-based smartnics. In: _____. **Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2021. p. 267–281. ISBN 9781450390989. Disponível em: <https://doi.org/10.1145/3485983.3494842>. Cited in page 32.

LAZAR, A. Programming telecommunication networks. **IEEE Network**, v. 11, n. 5, p. 8–18, 1997. Cited in page 15.

LAZAR, A.; LIM, K.-S.; MARCONCINI, F. Realizing a foundation for programmability of atm networks with the binding architecture. **IEEE Journal on Selected Areas in Communications**, v. 14, n. 7, p. 1214–1227, 1996. Cited in page 15.

LEE, J.-B. et al. High-performance software load balancer for cloud-native architecture. **IEEE Access**, v. 9, p. 123704–123716, 2021. Cited in page 38.

LEVIN, J. Viperprobe: Using ebpf metrics to improve microservice observability. In: . [S.l.: s.n.], 2020. Cited in page 34.

LI, X. et al. Towards power efficient high performance packet i/o. **IEEE Transactions on Parallel and Distributed Systems**, v. 31, n. 4, p. 981–996, 2020. Cited in page 38.

LIU, C. et al. A protocol-independent container network observability analysis system based on ebpf. In: **2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)**. [S.l.: s.n.], 2020. p. 697–702. Cited in page 34.

LORETI, P. et al. Srv6-pm: Performance monitoring of srv6 networks with a cloud-native architecture. **CoRR**, abs/2007.08633, 2020. Disponível em: <https://arxiv.org/abs/2007.08633>. Cited in page 33.

LUIZELLI, M. C. et al. In-network neural networks: Challenges and opportunities for innovation. **IEEE Network**, v. 35, n. 6, p. 68–74, 2021. Cited in page 37.

MAYER, A. et al. Performance monitoring withh^2: Hybrid kernel/ebpf data plane for srv6 based hybrid sdn. **Computer Networks**, v. 185, p. 107705, 2021. ISSN 1389-1286. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1389128620313037>. Cited in page 36.

MCCANNE, S.; JACOBSON, V. The bsd packet filter: A new architecture for user-level packet capture. In: **In Proceedings of the USENIX Winter 1993 Conference**. Berkeley, CA, USA: USENIX Association, 1993. p. 2–2. Cited 2 times in the pages 12 e 23.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1355734.1355746>. Cited 2 times in the pages 16 e 17.

MIANO, S. et al. Securing linux with a faster and scalable iptables. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 49, n. 3, p. 2–17, nov 2019. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/3371927.3371929>. Cited in page 34.

MIANO, S. et al. Introducing smartnics in server-based data plane processing: The ddos mitigation use case. **IEEE Access**, v. 7, p. 107161–107170, 2019. Cited in page 31.

MIANO, S. et al. A framework for ebpf-based network functions in an era of microservices. **IEEE Transactions on Network and Service Management**, v. 18, n. 1, p. 133–151, 2021. Cited in page 38.

MIANO, S. et al. Dynamic recompilation of software network services with morpheus. **CoRR**, abs/2106.08833, 2021. Disponível em: <https://arxiv.org/abs/2106.08833>. Cited in page 36.

NAM, T.; KIM, J. Open-source io visor ebpf-based packet tracing on multiple network interfaces of linux boxes. In: **2017 International Conference on Information and Communication Technology Convergence (ICTC)**. [S.l.: s.n.], 2017. p. 324–326. Cited in page 34.

PANTUZA, G.; VIEIRA, M. A. M.; VIEIRA, L. F. M. equic gateway: Maximizing quic throughput using a gateway service based on ebpf + xdp. In: **2021 IEEE Symposium on Computers and Communications (ISCC)**. [S.l.: s.n.], 2021. p. 1–6. Cited in page 37.

PAROLA, F.; MIANO, S.; RISSO, F. A proof-of-concept 5g mobile gateway with ebpf. In: _____. **Proceedings of the SIGCOMM '20 Poster and Demo Sessions**. New York, NY, USA: Association for Computing Machinery, 2020. p. 68–69. ISBN 9781450380485. Disponível em: <https://doi.org/10.1145/3405837.3411395>. Cited in page 37.

PAROLA, F.; PROCOPIO, R.; RISSO, F. Assessing the performance of xdp and af_xdp based nfs in edge data center scenarios. In: _____. **Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2021. p. 481–482. ISBN 9781450390989. Disponível em: <https://doi.org/10.1145/3485983.3493352>. Cited in page 32.

PEPELNJAK, I. **SDN controller northbound API is the crucial missing piece**. 2012. Disponível em: <https://blog.ipspace.net/2012/09/sdn-controller-northbound-api-is.html>. Cited in page 20.

Pizzutti, M.; Schaeffer-Filho, A. E. Adaptive multipath routing based on hybrid data and control plane operation. In: **IEEE INFOCOM**. [S.l.: s.n.], 2019. p. 730–738. Cited in page 11.

RIVERA, S. et al. Ros-fm: Fast monitoring for the robotic operating system(ros). In: **2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)**. [S.l.: s.n.], 2020. p. 187–196. Cited in page 34.

ROSSI, F. et al. The actual cost of programmable smartnics: diving into the existing limits. In: . [S.l.: s.n.], 2021. Cited 3 times in the pages 11, 12 e 32.

SANTOS, E. et al. Aplicações de monitoramento de tráfego utilizando redes programáveis ebpf. In: **Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Porto Alegre, RS, Brasil: SBC, 2019. p. 417–430. ISSN 2177-9384. Disponível em: <https://sol.sbc.org.br/index.php/sbrc/article/view/7376>. Cited in page 33.

SAQUETTI, M. et al. Toward in-network intelligence: Running distributed artificial neural networks in the data plane. **IEEE Communications Letters**, v. 25, n. 11, p. 3551–3555, 2021. Cited in page 37.

SCHENKER, S. **The future of networking, the past of protocols**. 2011. Disponível em: <http://www.youtube.com/watch?v=YHeyuD89n1Y>. Cited in page 18.

SCHOLZ, D. et al. Performance implications of packet filtering with linux ebpf. In: **2018 30th International Teletraffic Congress (ITC 30)**. [S.l.: s.n.], 2018. v. 01, p. 209–217. Cited in page 35.

SHEINBEIN, D.; WEBER, R. P. Stored program controlled network: 800 service using spc network capability. **The Bell System Technical Journal**, v. 61, n. 7, p. 1737–1744, 1982. Cited in page 16.

SOMMERS, J.; DURAIRAJAN, R. **ELF: High-Performance In-band Network Measurement**. 2021. Disponível em: <https://tma.ifip.org/2021/wp-content/uploads/sites/10/2021/08/tma2021-paper17.pdf>. Cited in page 33.

STEADMAN, J.; SCOTT-HAYWARD, S. Dnsxp: Enhancing data exfiltration protection through data plane programmability. **Computer Networks**, v. 195, p. 108174, 2021. ISSN 1389-1286. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1389128621002310>. Cited in page 36.

T., H.-J. et al. The express data path: Fast programmable packet processing in the operating system kernel. In: **In Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies**. New York, NY, USA: CoNEXT 2018, 2018. p. 54–66. Cited in page 25.

TENNENHOUSE, D. et al. A survey of active network research. **IEEE Communications Magazine**, v. 35, n. 1, p. 80–86, 1997. Cited in page 15.

TU, N. V. et al. Intcollector: A high-performance collector for in-band network telemetry. In: **2018 14th International Conference on Network and Service Management (CNSM)**. [S.l.: s.n.], 2018. p. 10–18. Cited in page 33.

TU, N. V.; YOO, J.-H.; HONG, J. W.-K. Accelerating virtual network functions with fast-slow path architecture using express data path. **IEEE Transactions on Network and Service Management**, v. 17, n. 3, p. 1474–1486, 2020. Cited in page 36.

VIEIRA, M. et al. Processamento rápido de pacotes com ebpf e xdp. In: _____. [S.l.: s.n.], 2019. p. 92–141. ISBN 9786587003559. Cited in page 24.

WENG, T. et al. Kmon: An in-kernel transparent monitoring system for microservice systems with ebpf. In: **2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)**. Los Alamitos, CA, USA: IEEE Computer Society, 2021. p. 25–30. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/CloudIntelligence52565.2021.00014>. Cited in page 33.

XIONG, Z.; ZILBERMAN, N. Do switches dream of machine learning? toward in-network classification. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2019. p. 25–33. Cited in page 11.

ÍNDICE