Roteador SDN em hardware independente de protocolo com análise, casamento e ações dinâmicas

Racyus D. G. Pacífico[1], Gerferson R. Coelho[2], Marcos A. M. Vieira[1], José A. M. Nacif[2]

[1]Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

[2]Universidade Federal de Viçosa (UFV) – Florestal, MG – Brasil

{racyus,mmvieira}@dcc.ufmg.br, {gerferson.coelho,jnacif}@ufv.br

Abstract. *The OpenFlow standard is the most used solution in SDN, separating the data plane from the control plane and using a limited set of fields and actions. However, OpenFlow does not allow to include new fields outside the specification, making it difficult to adopt new protocols and services. In this work, we propose a hardware-implemented SDN router that enables the use of dynamically defined new fields and protocols, without the need to recompile or restart the router when the user changes, at run time, how the flows should be processed. In addition, our system enables the processing of protocol-independent SDN network flows using eBPF instructions generated from C or P4 language programs created by the user. Our prototype was instrumented on the NetFPGA platform. Our results show that the system allows to change parsings, matchings, and actions at run time with zero downtime.*

Resumo. *O padrão OpenFlow é a solução mais utilizada em SDN, separando o plano de dados do plano de controle e usando um conjunto limitado de campos e ações. Porém, o OpenFlow não permite utilizar novos campos fora da especificação, dificultando a adoção de novos protocolos e serviços. Neste trabalho propomos um roteador SDN implementado em hardware com o objetivo de possibilitar a utilização de novos campos e protocolos definidos dinamicamente, sem a necessidade de recompilar ou reiniciar o roteador quando o usuário altera, em tempo de execução, a forma como os fluxos devem ser processados. Além disso, ele permite o processamento de fluxos de redes SDN independente de protocolo utilizando instruções eBPF geradas a partir de programas em linguagem C ou P4 criados pelo usuário. Nosso protótipo foi implementado na plataforma NetFPGA. Nossos resultados mostram que o sistema permite alterar as análises, casamentos e ações em tempo de execução com tempo de inatividade zero.*

1. Introdução

Rede Definida por *Software* (*Software-Defined Networking*, ou SDN) é um paradigma para o desenvolvimento de pesquisas em redes de computadores que ganhou a atenção de parte da comunidade científica e da indústria da área [Guedes et al. 2012]. SDN é um paradigma que separa o plano de controle do plano de dados e permite programar os seus dispositivos [Macedo et al. 2015]. Em SDN, o plano de controle configura as regras de encaminhamento da rede de modo logicamente centralizado por *software*, por uma entidade chamada controlador, enquanto que o plano de dados encaminha os pacotes de

acordo com as ações definidas por esse controlador. Devido à estrutura que SDN fornece, áreas de pesquisa como engenharia de tráfego, qualidade de serviço (QoS) e virtualização têm evoluído rapidamente [Stubbe 2017].

O padrão OpenFlow [McKeown et al. 2008] é um exemplo de SDN, que teve um crescimento significativo desde o lançamento de sua primeira versão em 2008 até o lançamento da versão atual (1.5) [Kreutz et al. 2015]. A primeira versão do OpenFlow surgiu com uma tabela de casamento contendo apenas dez campos e evoluiu para múltiplas tabelas com 44 campos diferentes [Kreutz et al. 2015]. No entanto, o número de campos suportados pelo OpenFlow está sendo atualizado constantemente para suportar novos campos e protocolos como, por exemplo, o protocolo IPv6. Infelizmente, o OpenFlow tem um plano de dados dependente de protocolo com análise, casamento e ações fixas, dificultando o suporte de novos campos e protocolos [Jouet and Pezaros 2017].

Este trabalho propõe um roteador SDN implementado em *hardware* que possibilita a utilização de novos campos e protocolos definidos dinamicamente, sem a necessidade de recompilar ou reiniciar o roteador quando o usuário altera, em tempo de execução, a forma como os fluxos devem ser processados. O roteador SDN foi implementado na plataforma NetFPGA 1G, permitindo alto poder de processamento de pacotes da rede. Os testes realizados foram em ambiente real. A nossa arquitetura é um sistema SDN independente de protocolo e linguagem que permite alterar análises (*parsing*), casamentos e ações de fluxos SDN no plano de dados do *hardware* em tempo de execução com tempo de inatividade zero.

As principais contribuições deste trabalho são: (i) possibilitar a utilização de novos campos e protocolos definidos dinamicamente, sem a necessidade de recompilar ou reiniciar o roteador quando o usuário altera, em tempo de execução, a forma como os fluxos devem ser processados; (ii) processar fluxos de redes SDN independente de protocolo utilizando instruções eBPF geradas a partir de programas em linguagem C ou P4 criados pelo usuário; (iii) implementação do roteador SDN em *hardware* incluindo o processador eBPF. Nosso sistema permite alterar as análises, casamentos e ações em tempo de execução com tempo de inatividade zero. Ressaltamos que todas essas contribuições juntas não são possíveis de serem realizadas em outros sistemas como, por exemplo, BPFabic [Jouet and Pezaros 2017], PISCES [Shahbaz et al. 2016], P4FPGA [Wang et al. 2017] e dRMT [Chole et al. 2017].

Esse trabalho tem a seguinte organização: na seção 2 é apresentada uma visão geral da arquitetura SDN. Na seção 3 são descritos detalhes de implementação do roteador SDN na plataforma NetFPGA 1G. Na seção 4 são apresentados os resultados dos testes em um ambiente real. Na seção 5 são descritos os trabalhos relacionados. Finalmente, na seção 6 apresentamos as conclusões e trabalhos futuros.

2. Visão geral da arquitetura

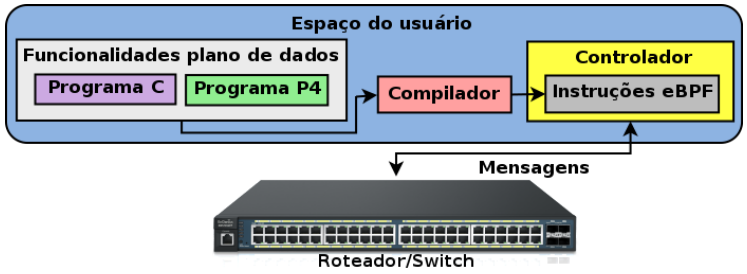

Figura 1. Visão geral da arquitetura.

A figura 1 apresenta uma visão geral da arquitetura. A arquitetura possui um controlador logicamente centralizado que comunica através de um soquete com os elementos de rede (roteadores e comutadores). Entre o controlador e o roteador existe uma API que permite o controlador enviar programas independentes de plataforma e protocolos.

O controlador pode instalar programas no plano de dados dos elementos de rede em tempo de execução sem interrupção da rede (sem tempo de inatividade). A vantagem dessa abordagem é que a análise, casamento e ações podem ser modificados em tempo de execução.

2.1. Plano de Dados

No sistema proposto, o roteador possui um processador de pacotes genérico, sem nenhum comportamento pré-estabelecido. O comportamento do roteador é definido pelo conjunto de instruções que acessa os cabeçalhos, o pacote de dados, e faz a análise, casamento e ações em tempo de execução. O sistema utiliza o processador de pacote eBPF.

2.1.1. eBPF

Berkeley Packet Filter (BPF) [McCanne and Jacobson 1993] é uma máquina virtual com um conjunto de instruções bem definido usado para filtragem de pacotes. A biblioteca *libpcap*, presente no utilitário *tcpdump*, é um exemplo que utiliza o BPF.

O eBPF (*enhanced* BPF) é uma nova versão do BPF na qual a arquitetura foi expandida de 32 para 64 bits, aumentando o número de registradores de escrita de 2 para 10, e adicionando operações de tabela e chamada de funções [Schulist et al.]. O eBPF está presente no *kernel* do Linux desde a versão 3.18. O eBPF permite projetar arquiteturas independente de plataformas ou protocolos. Não é necessário conhecimento prévio do protocolo ou da estrutura do pacotes. É possível fazer a análise (*parsing*) do pacote executando o conjunto de instruções.

Existe um verificador para programas com instruções eBPF. O verificador permite checar a validade, segurança e desempenho dos programas eBPF. O verificador verifica estaticamente se um programa termina, se os acessos de memória estão no intervalo de espaço de memória e qual o maior profundidade do caminho de execução. A implementação do *kernel* do Linux já provê uma implementação do verificador eBPF.

O verificador pode ser usado depois do código ter sido compilado e antes de carregá-lo no plano de dados.

Neste trabalho, é proposto utilizar o conjunto de instrução do eBPF para análise, casamento e definição das ações no plano de dados.

2.1.2. Linguagem de Alto Nível

Linguagens de alto nível podem ser utilizadas para escrever código para o plano de dados e compilá-lo para o conjunto de instruções do eBPF. Já existe um subconjunto de C, que exclui algumas bibliotecas externas, chamadas de sistemas e aritmética de ponteiro enquanto provê funções para definição e manipulação de tabelas. O compilador LLVM 3.9 possui um *backend* para a plataforma eBPF, permitindo programar nesse subconjunto de C e gerando código executável no formato eBPF.

Também é possível gerar instruções eBPF através de linguagens de domínio específico, como por exemplo, P4 [Bosshart et al. 2014]. Existem esforços do projeto de código aberto Iovisor [IOv 2016] que já implementaram um compilador parcial de P4 para eBPF.

```
uint64_t prog(struct packet *pkt){
    // Verifica se o quadro Ethernet contem um pacote ipv4
    if (pkt->eth.h_proto == 0x0008) {
        struct ip *ipv4 = (struct ip *)(((uint8_t *)&pkt->eth) + ETH_HLEN);
        // Verifica se o pacote IP contem um segmento TCP
        if (ipv4->ip_p == 6) {
            struct tcphdr *tcp = (struct tcphdr *)(((uint32_t *)ipv4) + ipv4->ip_hl);
            // Acessa o numero de sequencia
            return tcp->th_seq;
}
```

Figura 2. Exemplo de programa em C que acessa o número de sequência do TCP. Isso não é possível no padrão OpenFlow.

A figura 2 ilustra um exemplo não suportado pelo padrão OpenFlow, mas que pode ser executado pelo sistema proposto: analisar o segmento TCP e acessar o número de sequência. O exemplo de código, escrito no subconjunto de C, verifica se o quadro Ethernet possui um pacote IP e se contém um segmento TCP. Depois, ele acessa o valor do campo número de sequência do TCP. Dysco [Zave et al. 2017] é um exemplo de sistema que provê um protocolo de sessão que utiliza o número de sequência do TCP.

Após a apresentação deste exemplo podemos contextualizar melhor a ideia geral apresentada na figura 1. Neste cenário, o gerente de rede escreve um programa nas linguagens C (figura 2) ou P4 e, em seguida, compila para eBPF. Este programa eBPF é então instalado no roteador através do controlador.

3. Implementação

A figura 3 apresenta uma visão geral da implementação do roteador SDN na plataforma NetFPGA [Net 2007]. A implementação do roteador SDN é composta de dois componentes: plano de dados e ferramentas no espaço do usuário. O plano de dados contém módulos em *hardware* que pertencem à biblioteca padrão da NetFPGA (módulos na cor cinza) e módulos específicos do roteador SDN (módulos na cor laranja). As ferramentas no espaço de usuário são compostas por um controlador responsável por

definir como o plano de dados encaminhará os pacotes, e *softwares* para configurar as funcionalidades do plano de dados, de acordo com os programas C e P4 criados pelo usuários.

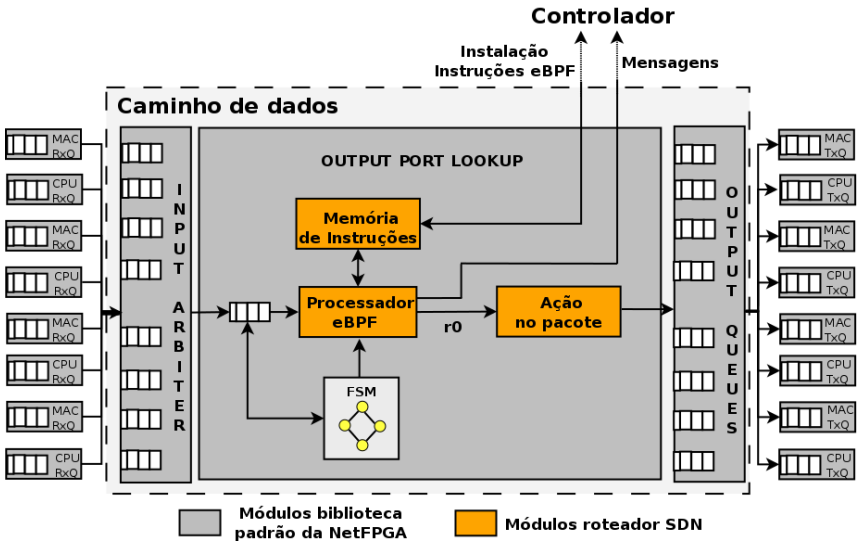

Figura 3. Caminho de dados do roteador implementado na NetFPGA 1G.

3.1. Plano de Dados

A biblioteca padrão da NetFPGA fornece os seguintes módulos: *input arbiter*, *output port lookup* e *output queue*. O *input arbiter* recebe os pacotes da rede (via interface MAC) ou do barramento PCI (via CPU) no formato de palavras de 64 bits e as armazena em filas internas do módulo. Em seguida, as palavras são retiradas das filas utilizando o algoritmo *round robin*, e as encaminha para o módulo *output port lookup*. O módulo *output port lookup* é responsável por definir qual porta o pacote será enviado baseado nas propriedades do pacote, por exemplo, porta de entrada ou endereço destino. A porta de saída de todos os pacotes recebidos é configurada no campo porta de saída do metadado.

O roteador SDN estende o caminho de dados da NetFPGA com três módulos em *hardware* (cor laranja): memória de instruções, processador eBPF e ação no pacote. Na memória de instruções são armazenados as instruções eBPF geradas a partir do código C e P4 criados pelo usuário. Estas instruções definem o comportamento do plano de dados. O processador eBPF é responsável por realizar a análise, casamento e ações utilizando as instruções armazenadas na memória de instruções. Além disso, o processador se comunica com o plano de controle através de um soquete. O módulo ação no pacote apenas encaminha ou descarta o pacote de acordo com o valor armazenado no registrador r0 após o processamento das instruções eBPF.

3.1.1. Máquina de Estados

A máquina de estados (*finite state machine*, FSM) no módulo *output port lookup* controla todo o funcionamento do processamento de pacotes. Ela retira o pacote da fila interna do módulo (encaminhado pelo módulo *input arbiter*), começa a execução das instruções eBPF e encaminha o pacote para

o próximo módulo (ação no pacote) quando a última instrução (`exit`) do eBPF for executada. A FSM é composta de oito estados: *read_data_in_fifo*, *write_header_data_mem*, *write_pkt_data_mem*, *start_ebpf_eng*, *wait_stop_ebpf_eng*, *unload_data_mem*, *wait_read_data_mem* e *forward_read_data_mem*.

3.1.2. Processador eBPF

O processador eBPF é responsável por realizar a análise, casamento e ações de acordo com as instruções eBPF geradas a partir do código C ou P4 criados pelo usuário no espaço do usuário. Antes de iniciar o funcionamento do roteador SDN, o usuário deve carregar as instruções eBPF na memória de instruções para definir qual será o comportamento do plano de dados do roteador.

A figura 4 apresenta o caminho de dados e de controle do processador eBPF em nível de transferência de registrador (*register transfer level*, RTL) contendo cinco unidades funcionais: contador de programa, memória de instruções, controle, unidade lógica aritmética (ALU) e memória de dados.

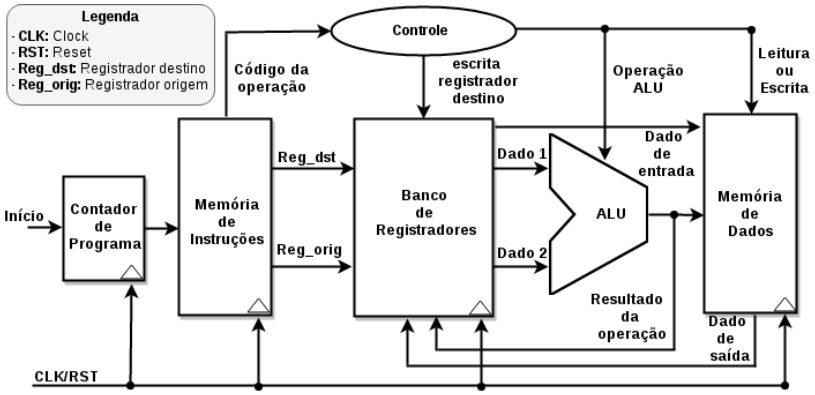

Figura 4. Caminho de dados e de controle do processador eBPF.

Após a saída da instrução da memória de instruções, a instrução é dividida em cinco partes: código da operação, endereço registrador destino e registrador origem, deslocamento e imediato. Cada parte da instrução é enviada para uma unidade específica. O controlador recebe o código da operação e encaminha os sinais de controle para as unidades funcionais, definindo qual o comportamento de cada unidade. Por exemplo, instruções da classe ALU (unidade lógica e aritmética) não utilizam a memória de dados, então os sinais de leitura/escrita não são ativos na memória de dados.

O opcode do eBPF é divido em sete classes: *load* imediato, *load*, *store* imediato, *store*, operações lógicas e aritméticas, desvios, operações lógicas e aritméticas de 64 bits, e a outra classe é reservada para uso futuro. O processador eBPF possui instrução de 64 bits, sendo 32 bits para representação de um número imediato (`imm`), 16 bits para o `offset`, 4 bits para registrador de origem, 4 bits para registrador de destino e, por fim, o código da operação (`opcode`) de 8 bits. O opcode pode ser redividido dependendo da classe da instrução. Para *load* e *store*, o opcode possui 3 bits mais significativos para modo de acesso à memória, 2 bits para tamanho da palavra e 3 bits para classe de instrução. Para as instruções de desvio, aritméticas e lógicas, os 4 bits mais significativos especificam qual

é a operação, 1 bit para especificar se a instrução é realizada com o valor imediato ou com o operando de origem e por fim 3 bits menos significativos para a classe da instrução.

No banco de registradores é realizado operações de leitura e escrita de dados nos registradores. Na operação de leitura o banco de registradores recebe os endereços do registrador destino/origem e encaminha os dados referente aos endereços para ALU. A operação de escrita no banco de registradores é realizada após a leitura do dado na memória de dados ou operação da ALU.

A tabela 1 descreve a funcionalidade de cada registradores eBPF.

Tabela 1. Descrição do conjunto de registradores do eBPF

Registrador	Descrição
R0	valor de retorno de funções e da saída do programa eBPF.
R1 - R5	passagem de argumento de funções.
R6 - R9	registradores que preservam valores em chamada de função (*callee-saved*).
R10	ponteiro do quadro para acesso a pilha. É apenas de leitura.

A ALU do processador eBPF é capaz de realizar operações aritméticas e lógicas baseado no sinal enviado pelo controle. O resultado da ALU pode ser encaminhado como endereço da memória de dados ou como dado de escrita do registrador destino no banco de registradores.

A memória de dados tem como funcionalidade o armazenamento do metadado e palavras do pacote para que o processador eBPF realize as ações no pacote de acordo com as instruções eBPF. O tamanho da memória de dados do processador eBPF tem capacidade de até 256 palavras de 64 bits. Com essa capacidade apenas um metadado e um pacote de até 1.500 bytes pode ser armazenado. Nós definimos a memória de dados com esta capacidade devido ao pouco recurso lógico (53.000 células lógicas) disponível na FPGA da NetFPGA 1G.

O processador eBPF possui um conjunto de instruções responsáveis por ler e escrever na memória de dados com os seguintes tipos de tamanho: 8 bits (byte (B)), 16 bits (*half-word* (H)), 32 bits (*word* (w)), e 64 bits (*double word* (DW)). Para ter a possibilidade de ler e escrever dados de tamanhos variados na memória de dados, foi necessário criar sinais de controle responsáveis por controlar o módulo da memória de dados que especificasse o tamanho do bloco a ser lido.

3.1.3. Ações

O valor de retorno do motor eBPF, que fica armazenado no registrador r0, é utilizado para determinar qual ação deve ser realizada no pacote. A tabela 2 descreve os valores de retorno do eBPF e suas respectivas ações realizadas no pacote. Depois que o eBPF termina sua computação, o pacote pode: ser encaminhado para porta de saída; encaminhado para o controlador; ser descartado; fazer a inundação.

O *hardware* NetFPGA é composto por quatro portas Ethernet. Cada porta contém duas filas MAC e CPU (figura 3) onde o pacote pode ser encaminhado. Os valores da

Tabela 2. Ações realizadas no pacote.

Ação	Código	Descrição
Encaminhamento	0 - 0xFFF5	Encaminha o pacote.
Controlador	0xFFF6	Enviar o pacote para o controlador.
Descarte	0x0000	Descartar o pacote.
Inundação	0xFFFF	Enviar o pacote para todas as portas exceto a porta de entrada.

codificação das respectivas ações são baseados na quantidade de portas da plataforma NetFPGA.

3.1.4. Memória de Instrução

As instruções eBPF definem o comportamento de como o processador eBPF processará os pacotes. As instruções são inseridas na memória de instruções através da interface de registradores da NetFPGA. Foi criado registradores de *software* para inserir as instruções eBPF na memória de instruções do processador. As instruções são recebidas no roteador via mensagem enviada pelo controlador. Em seguida, as instruções são escritas nos registradores de *software* e encaminhadas para memória de instruções via o barramento PCI do *hardware*.

Como decisão de projeto definimos colocar a memória de instruções fora do processador eBPF com o objetivo de, no futuro, conectar vários processadores utilizando uma memória de instrução de forma compartilhada. Aumentando o número de processadores eBPF no caminho de dados do roteador SDN, mais pacotes podem ser processados, contribuindo para o aumento da vazão do roteador.

3.1.5. Metadados

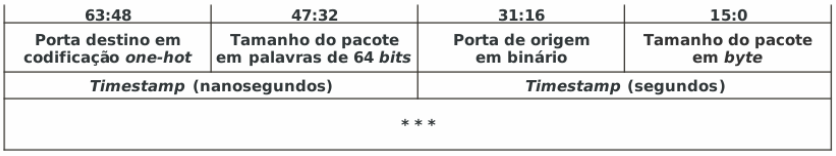

Figura 5. Metadado: Informações retiradas do pacote e armazenadas na memória de dados do processsador eBPF.

O plano de dados recebe o pacote pela interface de entrada e armazena o pacote na fila de entrada com alguns informações adicionais chamadas de metadados. A figura 5 apresenta a estrutura armazenada. Primeiramente os metadados são recebidos, em seguida, o quadro Ethernet ou pacote IP. Os metadados pré-afixados também podem ser usados pelo programa eBPF assim como qualquer outro campo de protocolo. Os metadados atualmente definidos são a porta de destino, o tamanho do pacote em múltiplos de 64 bits, a porta de origem, o tamanho do pacote em *bytes*, o *timestamp* em segundos

e nanosegundos. Os campos sobre o tamanho do pacote em 64 bits e em *bytes* foram incluídos porque o módulo da fila de entrada já provia essa informação.

3.2. Espaço do usuário

Tabela 3. Mensagens suportadas entre controlador e roteador SDN.

Tipo da mensagem	Direção	Descrição
Oi	controlador $\leftrightarrow$ roteador	Comunicação estabelecida
Instalar	controlador $\rightarrow$ roteador	Instalar instruções eBPF no roteador

O espaço do usuário e o plano de dados do roteador SDN se comunicam através de soquete. Com a comunicação estabelecida, ambos controlador e roteador podem trocar mensagens com informações específicas. A tabela 3 apresenta as mensagens suportadas entre controlador e roteador SDN.

A primeira mensagem enviada após a comunicação ser estabelecida é a mensagem "Oi". Esta mensagem é enviada para verificar que o controlador está conectado ao roteador. Em seguida, a mensagem "Instalar"é enviada contendo as instruções eBPF que serão instaladas na memória de instruções do processador eBPF.

3.3. Instância do *hardware*

Neste trabalho, foi implementado o roteador SDN no *hardware* NetFPGA 1G. O *hardware* NetFPGA 1G possui quatro portas Ethernet de 1 Gbps. A NetFPGA possui uma FPGA Vertex-II Pro 50 com aproximadamente 53.000 células lógicas, uma SRAM de 4 KiB com 2^{19} linhas e ciclo de relógio de 8 ns (125 MHz).

Os pacotes na NetFPGA são processados em forma de palavras de 64 bits e 8 bits de controle para identificar o tipo de palavra (dados ou metadado). Palavras são transmitidas pelo sinal `data` e o controle pelo sinal `ctrl`. Se o sinal de controle é igual a zero significa que as palavras do pacote estão sendo transmitidas, caso contrário as palavras são do metadados.

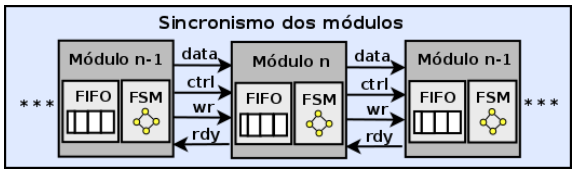

Figura 6. Sincronização dos módulos no roteador SDN.

O sincronismo dos módulos da NetFPGA funcionam de modo padronizado. A figura 6 apresenta o sincronismo dos módulos com os respectivos sinais no *hardware*. Os sinais *write* (`wr`) (escrita) e *ready* (`rdy`) (pronto) controlam a comunicação entre os módulos. O sinal `wr` informa quando o módulo anterior está pronto para transmitir (possui conteúdo para transmitir) e o sinal `rdy` informa que o módulo posterior está pronto para receber. A transferência dos dados entre módulos só ocorre se os dois sinais estiverem ativos.

A NetFPGA processa até 64 bits de dados do pacote a cada ciclo de relógio (8 ns). Considerando um pacote de 1.500 *bytes* de tamanho e com esse ciclo, um pacote leva aproximadamente 2 μs para ser processado.

Cada módulo contém uma fila de entrada e uma máquina de estados. A fila interna é utilizada para armazenar temporariamente o sinal `data` e `ctrl` até que a máquina de estado do módulo possa retirar da fila a palavra e o controle para começar o processamento. A máquina de estado de cada módulo tem um comportamento específico, por exemplo, o módulo *output port lookup* define a porta de saída de acordo com a porta de entrada.

4. Resultados

4.1. Validação do processador eBPF

Primeiramente, para validação do processador eBPF foi utilizado a plataforma da Altera Cyclone IV EP4CE115. A FPGA desse *hardware* possui 114.480 células lógicas, ciclo de relógio de 50 MHz e um conjunto de memórias (EEPROM, SRAM, *Flash*) com capacidade de armazenamento de 8 a 128 MB. Utilizamos a plataforma da Altera com o objetivo de detectar e também corrigir erros de codificação de forma rápida antes de integrarmos o processador ao caminho de dados da NetFPGA.

Na figura 7 apresentamos a forma de onda gerada da execução de um programa *assembly*. A primeira instrução é um mov64 imediato. Nesta instrução o registrador r2 recebe o valor imediato 3. A segunda instrução também é um mov64 imediato. O registrador r3 recebe o valor 6. Na terceira instrução é realizado a operação de soma entre r2 e r3. O registrador r2 ao final da operação de soma tem valor igual a 9. A execução da última instrução é um desvio absoluto (ja) para o início do programa. A última linha na figura (alu_out) representa a saída da ALU de acordo com as respectivas instruções.

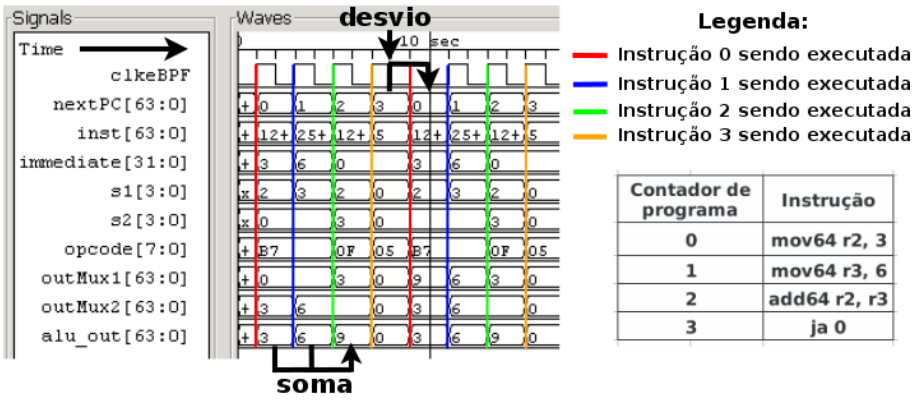

Figura 7. Testes das instruções em forma de onda do processador eBPF.

O processador eBPF foi sintetizado e testado para todas as instruções eBPF em uma plataforma Intel/Altera. Depois, o mesmo processador foi sintetizado na FPGA Vertex-II Pro 50 da Xilinx. O *software* da Xilinx sintetizou todas as instruções, com exceção das instruções de multiplicação, divisão e resto de divisão. Na prática, essas instruções não são muito utilizadas em um roteador. Além disso, essas instruções, com potência de 2, podem ser executadas com as operações de *shift* esquerdo, *shift* direito e o operador lógico E (*and*), respectivamente. Acreditamos que uma versão mais recente da NetFPGA possa aceitar a síntese dessas instruções.

4.2. Experimentos

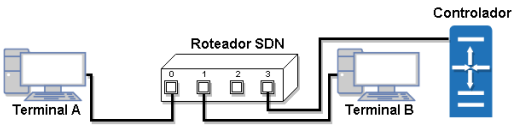

Figura 8. Topologia do experimento de validação do roteador SDN.

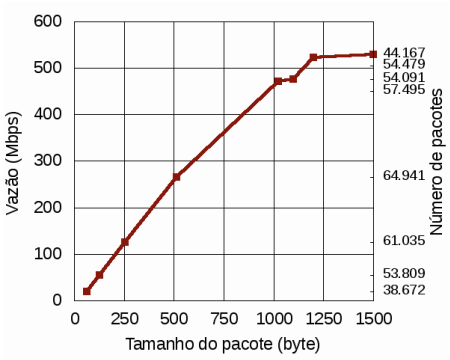

Figura 9. Vazão por tamanho de pacote.

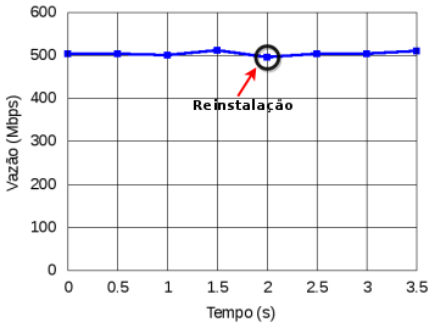

Figura 10. Vazão por tempo após carregar as instruções novamente.

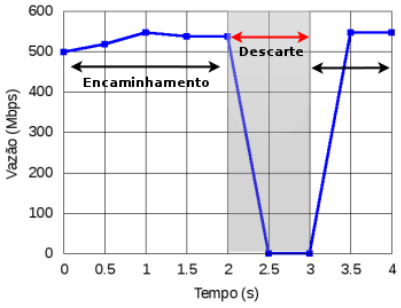

Figura 11. Vazão por tempo após carregar uma aplicação ACL.

A figura 8 apresenta a topologia do experimento. O roteador SDN interliga dois computadores hospedeiros pelas portas 0 e 1 e conecta com o controlador pela porta 3.

Foi escrito na linguagem C duas aplicações: encaminhamento de quadros e ACL (*Access Control List*). Ambos os códigos foram compilados com LLVM 3.9 para a plataforma eBPF. Utilizando a ferramenta *objcopy*, foi extraído o segmento *.text* do arquivo no formato `elf` que contém o conjunto de instrução. O controlador instalou ambos os códigos no roteador SDN. Utilizando a ferramenta *iperf*, foram criados conexões TCP e UDP entre os terminais A e B. A figura 9 mostra a vazão por tamanho do pacote. Neste experimento medimos a vazão do roteador para tamanho de pacotes: 64, 128, 256, 512, 1.024, 1.100, 1.200 e 1.500 *bytes*. A vazão máxima obtida foi de 530 Mbps com 44.167 pacotes processados em 1 s para pacotes de 1500 *bytes*.

Na figura 10 reinserimos o código de encaminhamento no tempo 2 s, avaliando o custo da instalação. Foram instaladas 20 instruções. Pela queda na vazão no período $[2, 2.5]$ infere-se que o tempo de instalação foi de 8 ms. Na figura 11 executamos o encaminhamento no período $[0, 2)$. Em 2 s, o ACL (descarta todos os pacotes de A $\leftrightarrow$ B) foi instalado. No período de $[2, 2.5]$ s a vazão diminuiu até chegar a zero devido ao esvaziamento das filas internas. Em 3 s, o encaminhamento sem ACL foi reinstalado. Com estes resultados validamos o sistema mostrando que o roteador tem um plano de dados com análise, casamento e ações dinâmicas.

5. Trabalhos relacionados

A tabela 4 compara os trabalhos relacionados, mostrando: descrição do trabalho, ano de publicação, qual linguagem de alto nível utiliza, se foi implementado em *hardware* ou *software*, qual modelo utilizado, e se uma nova instalação pode ser feita em tempo de execução com zero tempo de inatividade.

Tabela 4. Visão geral dos trabalhos relacionados.

Título	Descrição	Ano	Conversão	*Hardware* ou *Software*	Modelo	Instalação com zero tempo de inatividade
Este trabalho	Arquitetura com eBPF em *hardware*	2017	C ou P4 para eBPF	*Hardware*	NetFPGA 1G	✓
P4FPGA [Wang et al. 2017]	Implementação P4 em *hardware*	2017	P4 para *System* Verilog	*Hardware*	FPGA Virtex 7 XC7V690T	
BPFabric [Jouet and Pezaros 2017]	Arquitetura com suporte eBPF	2017	C ou P4 para eBPF	*Software*	Intel DPDK	✓
PISCES [Shahbaz et al. 2016]	Comutador com suporte P4	2016	P4 para código binário	*Software*	OpenvSwitch	
BEBA [BEBA 2015]	Arquitetura com suporte eBPF	2015	C FSM para eBPF	*Software*	Kernel hook	

BPFabric [Jouet and Pezaros 2017] foi proposto como uma plataforma em *software* que permite o processamento de pacotes independente do protocolo. O BPFabric utiliza instruções eBPF para definir como o processamento e o encaminhamento dos pacotes no plano de dados serão realizados. O BPFabric foi implementado sobre uma interface *socket raw* do Linux e o *framework* Intel DPDK. Nosso trabalho estende o BPFabric implementando o roteador SDN em *hardware*.

PISCES [Shahbaz et al. 2016] é um comutador em *software* baseado no *OpenvSwitch* (OVS) que suporta à linguagem de alto nível P4. P4 é utilizado para definir o plano de dados do protótipo. O comutador PISCES é uma versão modificada do OVS com análise (*parse*), casamento (*match*) e ações (*action*) gerados pelo compilador P4. O projeto do PISCES não é projetado para ser executado em *hardware*. O código fonte do PISCES deve ser compilado a cada alteração no programa P4.

P4FPGA [Wang et al. 2017] é uma plataforma desenvolvida em *hardware* que realiza conversão de programas P4 para *System Verilog*. O P4FPGA foi instrumentado na FPGA Xilinx Virtex-7 XC7V690T. Os três principais componentes da plataforma são: gerador de código, sistema *runtime* e o otimizador. O gerador de código tem como funcionalidade produzir um *pipeline* de processamento de pacotes. O sistema *runtime* fornece uma abstração independente do *hardware* para funcionalidades básicas incluindo gerenciamento de memória, gerenciamento de *transceiver* e comunicação controle/hospedeiro. O otimizador é utilizado para proporcionar o paralelismo no *hardware* aumentando a vazão e diminuindo a latência. P4FPGA não permite trocar as análises, casamentos e ações em tempo de execução.

O dRMT [Chole et al. 2017] é uma arquitetura que permite tabelas de casamento reconfiguráveis para processadores de pacotes em vários estágios implementada em ASIC que provê certa programabilidade e consegue processar os pacotes na taxa de entrada das interfaces de rede.

KeyFlow [de Oliveira et al. 2013] é um sistema que descreve uma nova abordagem para comutar pacotes. Neste sistema os pacotes recebem um rótulo baseado no Teorema Chinês do Resto. O comutador KeyFlow comuta pacotes com base em operações de resto da divisão (mod) no encaminhamento de pacotes. KeyFlow é complementar ao nosso sistema pois ele poderia ser implementado no nosso sistema, que já possui o processador eBPF que contem a instrução mod em hardware.

BEBA [BEBA 2015] é uma arquitetura baseada no processamento de pacotes usando uma FSM controlada por programas eBPF. Os estados da FSM são armazenados em tabelas de fluxo.

6. Conclusão e Trabalhos Futuros

Foi projetado e implementado em *hardware* na plataforma NetFPGA um roteador SDN com uma máquina virtual eBPF em seu núcleo. Esse sistema permite fazer a análise, casamento e ações de forma dinâmica através de instruções eBPF. O sistema é independente de protocolo e, ao contrário do padrão OpenFlow, permite utilizar novos campos, facilitando a adoção de novos protocolos e serviços em rede de computadores. As instruções eBPF são geradas a partir de programas escrito em linguagem C ou P4. O sistema permite alterar a imagem do programa eBPF em tempo de execução, permitindo modificar como os fluxos devem ser processados com tempo zero de inatividade.

Para trabalho futuro, é pretendido utilizar as tabelas TCAM e SRAM presente na plataforma em *hardware* e permitir que os programas escritos em C possam acessar essas tabelas. A TCAM já implementa o casamento do prefixo mais longo (*longest prefix matching*) em *hardware*, o que pode melhorar ainda mais o desempenho do sistema. Também é desejável utilizar um processador eBPF para cada fila de entrada de pacotes. Neste caso, será necessário uma FPGA com maior número de células disponíveis. Uma solução é adquirir a plataforma NetFPGA-SUME que possui maior poder de processamento. O nosso sistema já foi projetado de forma a integrar vários processadores eBPF, por isso, a memória de instrução está em outro módulo. A memória de instrução poderia ser lida em paralelo por vários processadores eBPF.

Agradecimentos

Agradecemos as agências de pesquisa CNPq, CAPES e FAPEMIG pelo apoio financeiro.

Referências

(2007). Netfpga project. www.netfpga.org. Acessado em 21/12/2017.

(2016). Iovisor project. www.iovisor.org. Acessado em 15/12/2017.

BEBA (2015). Behavioural based forwarding. http://www.beba-project.eu/.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

Chole, S., Fingerhut, A., Ma, S., Sivaraman, A., Vargaftik, S., Berger, A., Mendelson, G., Alizadeh, M., Chuang, S.-T., Keslassy, I., Orda, A., and Edsall, T. (2017). drmt:

Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 1–14, New York, NY, USA. ACM.

de Oliveira, R. E. Z., Vitoi, R., Martinello, M., and Ribeiro, M. R. N. (2013). Keyflow: Comutação por chaves locais de fluxos roteados na borda via identificadores globais. In *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBRC 2013, Brasília, DF. SBC.

Guedes, D., Vieira, L. F. M., Vieira, M. A. M., Rodrigues, H., and Nunes, R. V. (2012). Redes definidas por software: uma abordagem sistêmica para o desenvolvimento de pesquisas em redes de computadores. *Minicursos do Simpósio Brasileiro de Redes de Computadores*, 30(4):160–210.

Jouet, S. and Pezaros, D. P. (2017). Bpfabric: Data plane programmability for software defined networks. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '17, pages 38–48, Piscataway, NJ, USA. IEEE Press.

Kreutz, D., Ramos, F. M. V., Veríssimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76.

Macedo, D. F., Guedes, D., Vieira, L. F. M., Vieira, M. A. M., and Nogueira, M. (2015). Programmable networks: From software-defined radio to software-defined networking. *IEEE Communications Surveys Tutorials*, 17(2):1102–1125.

McCanne, S. and Jacobson, V. (1993). The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA. USENIX Association.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.

Schulist, J., Borkmann, D., and Starovoitov, A. Linux socket filtering aka berkeley packet filter (bpf). www.kernel.org/doc/Documentation/networking/filter.txt.

Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N., and Rexford, J. (2016). Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA. ACM.

Stubbe, H. (2017). P4 compiler & interpreter: A survey. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)*, 47.

Wang, H., Soulé, R., Dang, H. T., Lee, K. S., Shrivastav, V., Foster, N., and Weatherspoon, H. (2017). P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA. ACM.

Zave, P., Ferreira, R. A., Zou, X. K., Morimoto, M., and Rexford, J. (2017). Dynamic service chaining with dysco. SIGCOMM '17, pages 57–70, New York, NY, USA. ACM.