Sistema de processamento de pacotes *Serverless*

Racyus D. G. Pacífico[1], Lucas F. S. Duarte[2], Matheus S. Castanho[1], José A. M. Nacif[2], Marcos A. M. Vieira[1]

[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,{lucas.f.duarte,jnacif}@ufv.br

Resumo. *Serverless é um novo paradigma para computação em nuvem. Neste paradigma usuários apenas implementam o núcleo da aplicação sem preocupar com a infraestrutura. Processamento de funções ocorre em servidores com CPUs x86_x64 utilizando virtualização. Essa abordagem não é eficiente, gera atrasos de processamento e sobrecarrega recursos, aumentando o custo operacional. Neste trabalho é proposto um sistema Serverless com offloading em hardware que processa pacotes independente de protocolo via programas C eBPF. Nosso sistema foi implementado sobre NetFPGA SUME. Para melhorar o desempenho do sistema realizamos otimizações de hardware e integramos ao OpenFaaS. Os resultados mostram que o sistema opera em taxa de linha.*

1. Introdução

Serverless é um novo paradigma crucial para computação em nuvem que permite funções serem implementadas facilmente de modo escalável. Neste paradigma, milhares de funções são executadas em paralelo sem o usuário preocupar com a infraestrutura. Muitos provedores de nuvem já suportam funcionalidades *Serverless* como Amazon AWS Lambda [Services 2017], IBM Apache OpenWhisk [Foundation 2017], Microsoft Azure Functions [Microsoft 2017] e Google Cloud Functions [Google 2017]. Também existem provedores de código aberto, por exemplo, OpenLambda [Hendrickson et al. 2016].

Plataformas *Serverless* executam funções sobre servidores com CPUs x86_x64 utilizando virtualização. Essa abordagem não é eficiente, gera atrasos de processamento e sobrecarrega recursos, aumentando o custo operacional. Computação *offloading* em hardware permite que milhares de funções sejam processadas em paralelo com alta vazão e baixa latência. Um estudo recente [pow 2017] mostrou que computação *offloading* pode reduzir custos com despesas de capital e operacional. Por exemplo, para 2.200 funções por site é possível economizar até 12 milhões de dólares via economia de energia.

Este trabalho propõe um sistema de processamento de pacotes *Serverless* híbrido, com OpenFaaS orquestrando funções via contêineres de modo escalável com *offloading* em hardware. Nosso sistema visa contornar limitações existentes em plataformas *Serverless* com a computação ocorrendo sobre a plataforma NetFPGA SUME [Zilberman et al. 2014], um hardware programável com alto poder de processamento de pacotes de redes 10 G composto por uma FPGA. Ao projetar este sistema vários desafios foram encontrados devido aos requisitos existentes em *Serverless* que devem ser atendidos, como suportar reprogramação em tempo de execução, eficiência de energia, e tempo de inatividade zero para requisições de agendamento de funções.

As principais contribuições deste trabalho são: (i) processar funções usando programas eBPF (*extended Berkeley Packet Filter*) via OpenFaaS em hardware; (ii) estender um processador eBPF em hardware sobre a plataforma NetFPGA SUME; (iii) projetar e implementar otimizações a nível de hardware para melhorar o desempenho do sistema. Nosso sistema permite que funções sejam executadas via software sem o usuário conhecer comandos de baixo nível ou especificações do hardware. Além disso, o sistema fornece programabilidade no processamento de pacotes devido ao núcleo do sistema ter sido construído sobre um mecanismo que suporta o conjunto de instruções eBPF.

Este trabalho tem a seguinte organização: na seção 2 apresentamos uma visão geral do sistema. Na seção 3 descrevemos detalhes de implementação. Na seção 4, testes em um ambiente realista e resultados são discutidos. Na seção 5, trabalhos relacionais são descritos. Por fim, na seção 6 são apresentados as conclusões e trabalhos futuros.

2. Visão geral do sistema

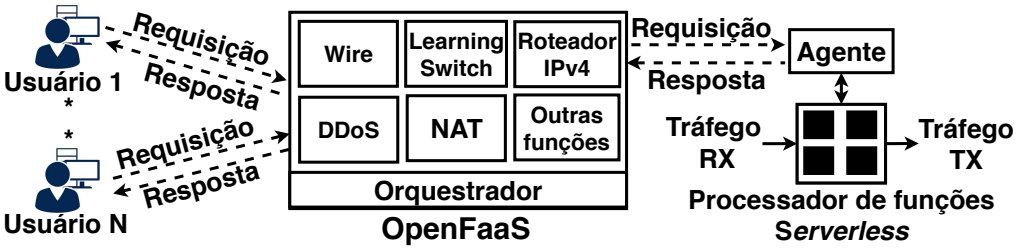

Figura 1. Visão geral do sistema.

A figura 1 apresenta uma visão geral do sistema. Usuários podem construir funções de rede eBPF no OpenFaaS, e executar tais funções em hardware sem preocupar com especificações de baixo nível. Antes de executar uma função, o usuário precisa adicionar o código da função no OpenFaaS para enviar requisições de execução da função via interface gráfica ou linha de comando. Quando uma requisição chega no OpenFaaS, um contêiner é alocado para função. O orquestrador é responsável por gerenciar contêineres de funções de modo escalável. Utilizamos o orquestrador *Docker Swarm* devido à simplicidade. Após essa etapa, uma requisição de execução com o tempo de processamento e função eBPF são enviados para o agente. O agente escalona a execução de funções de acordo com disponibilidade do processador. A computação da função termina com o usuário recebendo o resultado de processamento, liberando processador e contêiner.

2.1. Computação *Serverless*

Serverless adota o modelo FaaS (*Function as a Service*) onde serviços são escritos como funções individuais que podem ser gerenciadas separadamente sobre contêineres. Funções são orientadas à eventos e podem ser invocadas por requisições HTTP ou outros eventos que acontecem internamente ou externamente. Além disso, funções são executadas durante um período de tempo com a plataforma ficando responsável por alocar o recurso necessário para cada função de acordo com a demanda [Aditya et al. 2019]. A combinação *Serverless* e FaaS permite usuários executarem funções sem preocupar com recursos operacionais, gerenciamento, manutenção, escalabilidade

e tolerância a falhas [Baldini et al. 2017]. Atualmente, processamento utilizando virtualização está sendo substituído por computação *offloading* em hardwares como ASICs, FPGAs e SmartNICs para melhorar requisitos de processamento, eficiência de energia e programabilidade [Dargahi et al. 2017]. Mais detalhes sobre *Serverless* são descritos em [Vieira et al. 2020a], um minicurso sobre o assunto.

2.2. OpenFaaS

É uma plataforma *Serverless* de código aberto baseado na licença MIT que gerencia e executa funções via interface gráfica ou linha de comando de modo escalável. OpenFaaS está implementado na linguagem Golang, e contém um extenso conjunto de funcionalidades disponíveis em seu repositório devido à adesão pela comunidade da área. Todos os componentes e funções do OpenFaaS são executados sobre contêineres, ambientes isolados na máquina hospedeiro [Ellis 2016]. Contêineres comportam como uma instalação completa do Linux com seus próprios usuários, sistema de arquivos, processos e pilha de rede. Existe uma similaridade entre contêineres e máquinas virtuais. No entanto, contêineres são implementados no *kernel* do Linux compartilhando o mesmo recurso disponível no hospedeiro.

2.3. eBPF

É uma máquina virtual RISC 64 bits de propósito geral inserida no *kernel* do Linux desde a versão 3.15. eBPF proporciona processamento rápido de pacotes em tempo de execução dentro do *kernel* provendo programabilidade na computação de pacotes. Programas eBPF são compilados em bytecode eBPF antes de serem injetados no *kernel*. Linguagens como C e P4 já suportam essa tecnologia. Mais detalhes sobre eBPF são descritos em [Vieira et al. 2019, Vieira et al. 2020b], um minicurso completo sobre o assunto.

2.4. Processador de funções *Serverless*

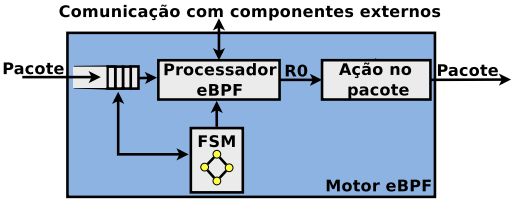

Figura 2. Design motor eBPF.

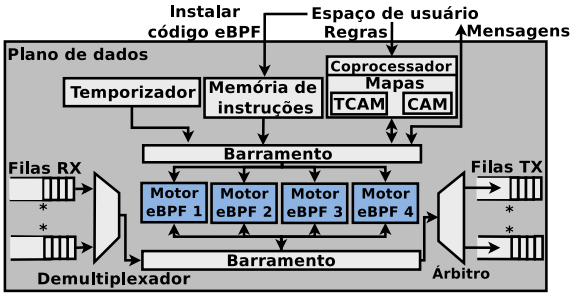

Figura 3. Design plano de dados.

Pacífico et al. [Pacífico et al. 2018] propôs um sistema em hardware que permite processamento de pacotes de redes 1 G via software utilizando um motor eBPF (figura 2). Neste sistema o motor eBPF é o núcleo do sistema, e suporta operações de análise, casamento e ações dinamicamente a partir de instruções eBPF. O motor eBPF é composto por quatro componentes: fila para armazenar pacotes, processador eBPF monociclo, máquina de estados para retirar o pacote da fila e copiar para memória de dados do processador, por fim, o módulo de ação no pacote para definir a ação a ser realizada no pacote de acordo com valor do registrador R0. Todo sistema foi implementado sobre o hardware NetFPGA 1 G.

Neste artigo estendemos o motor eBPF de Pacífico et al. [Pacífico et al. 2018] para a plataforma NetFPGA SUME com objetivo de realizar processamento de pacotes *Serverless* em taxa de linha de redes 10 G. Além disso, realizamos várias otimizações de hardware para prover tempo de inatividade zero, cópia zero de pacotes, paralelismo de instruções, uso de mapas e um processador por fila para melhorar a vazão. O sistema está dividido em dois componentes: plano de dados e ferramentas do espaço de usuário. O plano de dados (figura 3) é composto por quatro motores eBPF que compartilham uma memória de instruções, temporizador e coprocessador.

Para iniciar os motores eBPF, instruções eBPF devem ser carregadas na memória de instruções. O processamento no plano de dados começa com a chegada do pacote em uma das filas RX. Em seguida, o pacote é encaminhado pelo demultiplexador para um motor eBPF específico, de acordo com a fila de entrada que o pacote chegou. O motor eBPF processa e encaminha o pacote para o árbitro que decide qual fila TX o pacote será armazenado, para depois ser enviado. Com temporizador, o sistema é capaz de fornecer medições de desempenho da rede marcando o tempo de chegada de cada pacote no processador. O coprocessador é usado para trabalhar com mapas eBPF em hardware usando memórias TCAM/CAM para armazenar pares <chave,valor>. O espaço de usuário é composto por ferramentas para compilar/carregar programas, manipular mapas e comunicar com o plano de dados.

3. Implementação

3.1. Imagem *Docker*

Criamos uma imagem *Docker* denominada *alpine-ebpf* [Duarte 2019] com pacotes necessários para manipular funções de rede eBPF sobre contêineres. Utilizamos a imagem Linux *Alpine*, uma versão simplificada do sistema operacional que ocupa somente 5 MB. Além disso, atualizamos o *kernel* da imagem para versão 4.15.0 com objetivo de suportar a versão mais recente do eBPF. Instalamos três pacotes requisitos do sistema: GCC 8.3, Clang 8.0 e Python 3.7. GCC e Clang são utilizados para compilar funções. Python é utilizado para construir interfaces de entrada e saída de funções no OpenFaaS.

3.2. *Template* C eBPF

Templates definem quais configurações e arquivos são necessários para compilar códigos de uma linguagem específica. OpenFaaS permite uma imagem ser criada para uma função a partir do *template*. No sistema proposto criamos um *template* chamado *c-ebpf* a partir da imagem *alpine-ebpf*. O *template* criado contêm três componentes: *watchdog*, *handler* e *function*. *watchdog* é nativo do OpenFaaS, e funciona como um servidor HTTP que interage com a função e o mundo exterior. *handler* recebe a requisição enviada do *watchdog* e aciona o componente *function* para compilar e enviar o executável do programa eBPF para o agente. *handler* e *function* foram implementados em Python. No template, o usuário pode definir o tempo de execução da função de 3 a 300 s pelo parâmetro *time*. Definimos 5 s o tempo de execução das funções no sistema de acordo com padrão AWS Lambda.

3.3. Funções *Serverless* eBPF

Programas eBPF são transformados em funções *Serverless* no sistema usando a ferramenta *ebpfaas-cli*. Nesta ferramenta operações com funções ocorrem de forma

automatizada apenas fornecendo o nome da função e código eBPF de acordo com a operação. *ebpfaas-cli* suporta quatro tipos de operações: criar, executar, atualizar e excluir. O processo de criar uma função consiste utilizar a imagem *alpine-ebpf* e o *template c-ebpf*. O *ebpfaas-cli* cria um novo contêiner, e habilita a função para o usuário. Na operação executar, o usuário seleciona o contêiner da função via ferramenta, e o processamento começa a ser executado com a chegada de uma requisição (<*IP, porta*>) enviada do cliente para o agente. A operação atualizar permite alterar o conteúdo da função em tempo de execução apenas fornecendo o código eBPF modificado, sem reconstruir o contêiner. Por fim, na operação excluir a função é removida do sistema e os recursos alocados são liberados.

3.4. Loader

O carregamento do código no processador é feito utilizando a ferramenta *loader*. *Loader* foi projetado para o processador eBPF, e tem como funcionalidades modificar o código eBPF original e interagir com a interface de registradores. Quando o programa eBPF é carregado, duas instruções extras são anexadas no início da memória de instruções para inicializar os registradores R1 e R10 que indicam o endereço do início do pacote e topo da pilha. Essas instruções são inseridas externamente porque os valores de R1 e R10 não fazem parte do código gerado pelo compilador *clang*. Após essa etapa, o *loader* interage com a interface de registradores, inserindo as instruções na memória de instruções. Além desses procedimentos, o *loader* também pode ser utilizado para consultar informações como valor de R0 e qual memória de instruções está sendo utilizada.

3.5. Agente e escalonador

O agente é responsável por receber requisições de funções e repassá-las para o escalonador que tem a função de enfileirar requisições em uma fila à medida que chegam. A fila recebe o identificador da requisição e o programa eBPF a ser processado. A comunicação entre OpenFaaS e agente ocorre via soquete. Inserimos no *template* eBPF do sistema um *script* soquete em Python que permite o contêiner estabelecer conexão automaticamente com processador quando a função é requisitada pelo usuário. As requisições são retiradas da fila quando o processador termina de processar a função atual. A comunicação entre fila e processador ocorre via interface de registradores do hardware. O escalonador utiliza o *loader* para carregar as instruções no processador eBPF.

3.6. Processador eBPF com *pipeline*

O processador eBPF é responsável por realizar operações no pacote de acordo com as instruções eBPF geradas a partir do programa criado pelo usuário. Nesse trabalho projetamos um processador eBPF com *pipeline* de cinco estágios baseado na arquitetura MIPS devido à simplicidade, regularidade e rapidez [Hennessy and Patterson 2011]. Nós implementamos um *pipeline* no processador eBPF com objetivo de reduzir o atraso combinacional do circuito e proporcionar paralelismo de instruções. Na figura 4 apresentamos o caminho de dados e controle do processador proposto com *pipeline* de cinco estágios: (1) buscar instruções (*instruction fetch* - IF); (2) decodificar instrução e ler registradores (*instruction decode* - ID); (3) executar ou calcular endereço (*execute* - EXE); (4) acessar memória de dados (*memory* - MEM); e (5) escrever de volta o resultado processado (*write back* - WB).

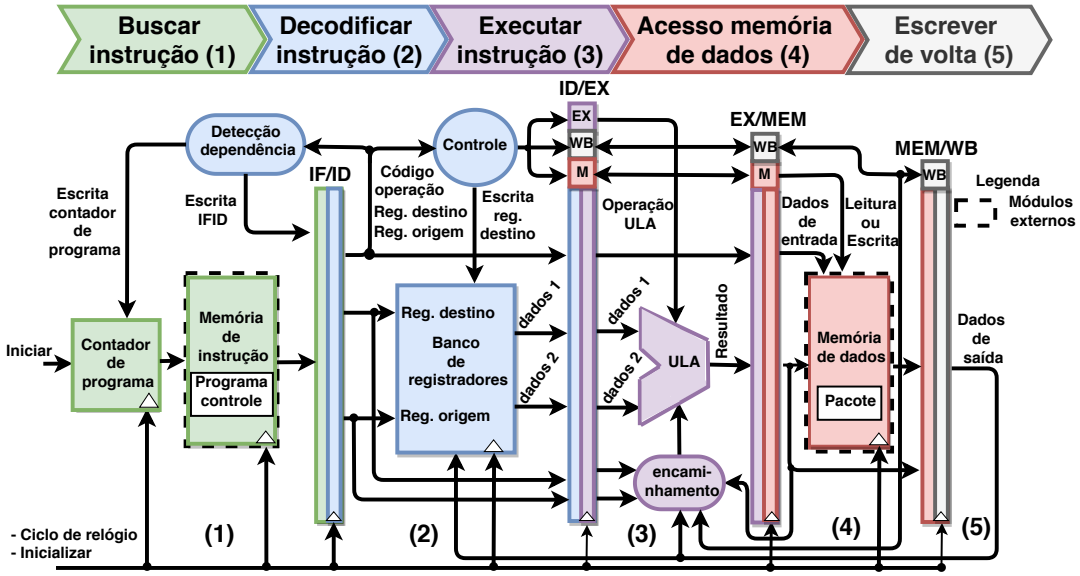

Figura 4. Caminho de dados e controle do processador eBPF.

Após a primeira palavra do pacote ser armazenada na memória de dados, o contador de programa é inicializado, e as instruções do programa começam a ser executadas no *pipeline*. No estágio IF (1) a instrução é lida da memória de instruções e o endereço do contador de programa é incrementado. O estágio ID (2) traduz o código da operação em sinais de controle e registradores de leitura para o banco de registradores. No estágio EXE (3) são realizados operações na ALU, por exemplo, computar se saltos devem ser tomados (*jump/branch*). No estágio MEM (4) a memória de dados é acessada por instruções *load* e *store*. Finalmente, no estágio WB (5) o banco de registradores é atualizado com o resultado da operação da ALU. Além dos registradores de *pipeline*, adicionamos unidades de encaminhamento e detecção de dependências para detectar dependências de dados e controle. Ambas as unidades foram implementadas com portas lógicas para minimizar a latência.

A unidade de encaminhamento foi projetada para resolver dependências de dados quando o registrador origem da instrução atual é o mesmo do registrador destino da instrução que está no estágio EX/MEM e MEM/WB. Quando existe dependência de dados com instruções *load*, a unidade de encaminhamento não consegue tratar essa situação, sendo necessário atrasar o *pipeline* em um ciclo usando uma bolha (*stall*) para que os dados estejam prontos. A unidade de detecção de dependência foi projetada para resolver dependências de controle quando instruções *jump/branch* são tomados. Quando essa situação ocorre, essa unidade descarta as instruções no estágio IF e ID.

3.7. Memória de dados (FIFO otimizada com zero cópia)

Para evitar *overhead* na cópia de pacotes da FIFO de transferência para memória de dados do processador, projetamos um novo tipo de dados abstrato, chamado FIFO memória de dados (FIFO_MD). FIFO_MD habilita o processador eBPF acessar palavras do pacote com zero cópia e executar operações de load e store com alta eficiência. Os módulos do caminho de dados da NetFPGA (*input arbiter* e *output queues*) estão sincronizados com

a FIFO_MD. Ela tem capacidade de 2 MB (64 de profundidade e 256 de largura) e pode armazenar até 32 pacotes de 64 bytes.

O motor eBPF pode começar processando o pacote sem todas as palavras do pacote chegarem. Isso acontece porque dentro da FIFO_MD existe um campo de bit válido que indica se a palavra está armazenada na FIFO_MD. Se a palavra é inválida, ou seja, o campo de bit válido desativado significa que a palavra do pacote ainda não chegou. Quando o processador solicita um endereço de uma palavra inválida, a FIFO_MD informa ao processador para mudar para o estágio inativo e espera a palavra ser armazenada para continuar o processamento do pacote.

3.8. Memória de instrução

Instruções eBPF são carregadas na memória de instrução via interface de registradores da NetFPGA. Registradores em software foram criados para inserir instruções na memória de instrução. As instruções são recebidas no agente, escritas nos registradores e depois encaminhadas para memória de instrução via barramento PCI do hardware.

Memória de instrução utiliza um sistema com *buffer* duplo (*double buffer system - DBS* - figura 5) para fornecer tempo de inatividade zero (*zero downtime*) na troca de programas eBPF. Este sistema é composto por duas memórias (M_1 and M_2). Ambas memórias nunca assumem o mesmo estado (escrita e leitura) ao mesmo tempo. Enquanto um programa está sendo escrito em uma memória, o processador lê as instruções de outra memória. DBS é inicializado com as instruções sendo escritas em M_1, enquanto M_2 permanece no estado inativo. O sistema permanece no estado de espera até um novo conjunto de instruções serem recebidos. Após as instruções serem escritas em M_1, o sistema retorna para o estado de espera. Quando um novo conjunto de instruções é recebido, as memórias trocam de estados. Este processo pode ser repetido infinitamente.

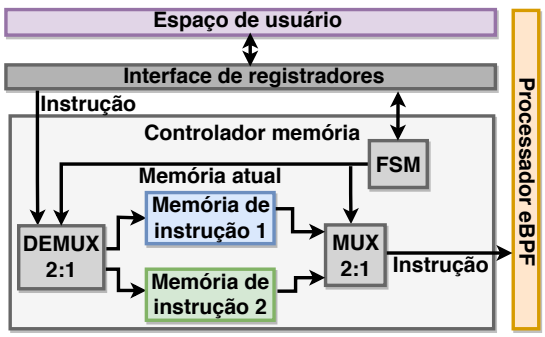

Figura 5. Memória de instruções.

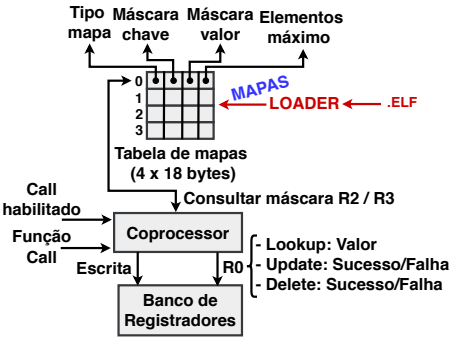

Figura 6. Design mapas.

3.8.1. Mapas

Mapas são estruturas de dados genéricas que armazenam diferentes tipos de dados no formato de pares <chave, valor>. Nosso sistema suporta o uso de dois tipos de mapas: casamento aproximado (*longest prefix matching-LPM*) e casamento exato. Para suportar casamento aproximado utilizamos o módulo TCAM (*Ternary Content Addressable Memory*) combinado com uma BRAM (*Block RAM*). TCAM usa três diferentes tipos

de entradas: 0, 1, e *don't care*. Uma operação de escrita na TCAM é realizada em 26 ciclos, e a leitura, em apenas 10 ciclos. No casamento exato usamos um módulo CAM (*Content Addressable Memory*) também combinado com uma BRAM. As entradas da CAM são apenas 0 e 1, não suportando *don't care*. O sistema gasta 12 ciclos para realizar uma escrita e 10 ciclos para leitura. A operação de escrita na CAM gasta menos ciclos se comparado com a TCAM, no entanto, na CAM não é possível inserir regras com coringas para permitir agregação de fluxos.

Nosso sistema suporta três funções para manusear mapas: *update*, *delete*, e *lookup*. A operação *update* atualiza um item dentro do mapa. A operação *delete* remove o item de uma determinada chave. Por fim, a operação *lookup* consulta a chave e retorna um item. Mapas são acessados através de chamadas de função do programa em C escrito pelo usuário. Eles podem conter pares <chave, valor> com tamanho em bytes diferentes.

O coprocessador precisa saber o tamanho real do mapa lido/escrito na memória de mapas. Esta informação está armazenada na tabela de mapas, apresentada na figura 6, que contém metadados sobre cada mapa declarado no programa carregado. A tabela de mapas é composta pelos campos: tipo mapa, máscara chave, máscara valor e número máximo de elementos permitidos no mapa. Os valores destes campos são passados para o processador via funções que operam os mapas, e depois salvos nos registradores R1-R4. A cada operação com mapas, a tabela de mapas é consultada para recuperar a chave e máscaras. Ambas informações recuperadas são utilizadas para remover bits não desejados por meio de uma operação AND com os dados armazenados nos registradores. O campo tipo de mapas é utilizado pelo coprocessador para definir qual memória será selecionada.

3.9. Instância do hardware

FPGA (*Field Programmable Gate Array*) permite construir sistemas lógicos em hardware. NetFPGA SUME é um hardware programável que contém quatro *transceivers* SFP+ que suportam portas de 10 Gbps Ethernet. Este hardware pode ser conectado a placa-mãe de um hospedeiro através do barramento PCIe x8 terceira geração. Ele contém uma FPGA Xilinx Virtex-7 690T FPGA [Xilinx 2010] com aproximadamente 693.120 células lógicas, uma SRAM 27 MiB e ciclo de relógio de 5 ns (200 MHz).

Pacotes na NetFPGA são processados no formato de palavras de 256 bits e 128 bits de controle para identificar o tipo da palavra (dados ou metadado). Palavras são transmitidas por sinais de dados e controle. Se o sinal de controle é zero, significa que as palavras do pacote estão sendo transmitidas, caso contrário, as palavras são do metadado. Cada módulo contém uma fila de entrada e uma máquina de estados. A fila de entrada é usada temporariamente para armazenar sinais de dados e controle até que a máquina de estados possa retirar a palavra da fila. Cada módulo contém uma máquina de estados com comportamento específico que pode ser modificada alterando o código Verilog. A sincronização entre módulos ocorre de maneira padronizada baseado no protocolo AXI-4 *Stream Xilinx* [Xilinx 2011].

4. Resultados

Nesta seção apresentamos e discutimos os resultados do sistema. Nós criamos quatro funções de rede: Wire, learning switch L2 (LSL2), roteador IPv4 (RIPv4) e mitigação DDoS (DDoS). Além disso, avaliamos fatores do sistema como processamento *Serverless*, tempo de inatividade, vazão, latência e energia.

O ambiente de testes foi composto por um computador para acessar as funções via navegador ou terminal, um servidor para executar o OpenFaaS, um servidor com a placa NetFPGA SUME, e um servidor gerador de tráfego executando *pktgen-DPDK*. O servidor gerador de tráfego estava equipado com uma placa da Netronome Agilio CX SmartNIC com duas interfaces de 10 Gbps conectadas diretamente nas portas 0 e 1 da NetFPGA. Com essa configuração conseguimos enviar tráfego em taxa de linha, e a NetFPGA receber e enviar 10 Gbps simultaneamente. Ambos computador e servidores contêm processador i7-7700@ de 3.60 GHz com 8 núcleos e 8 GB de RAM. Nós geramos tráfego variando o tamanho do pacote de 64 a 1500 Bytes (mínimo e máximo) usando o *pktgen-DPDK* rodando na placa da Netronome.

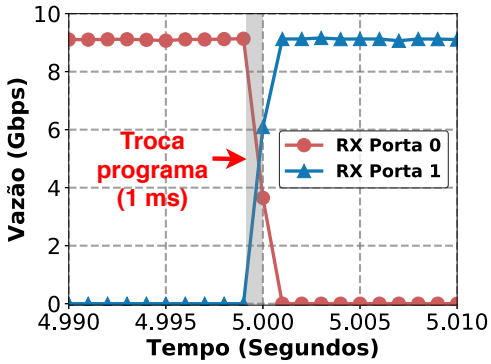

Figura 7. *Serverless*.

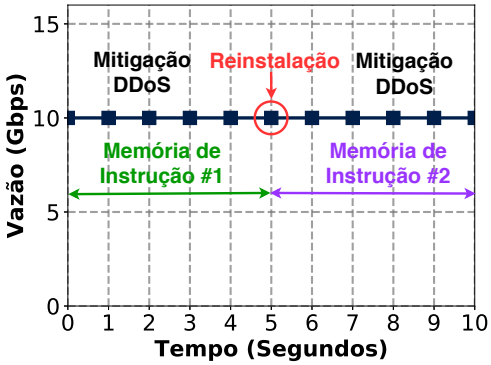

Figura 8. Inatividade zero.

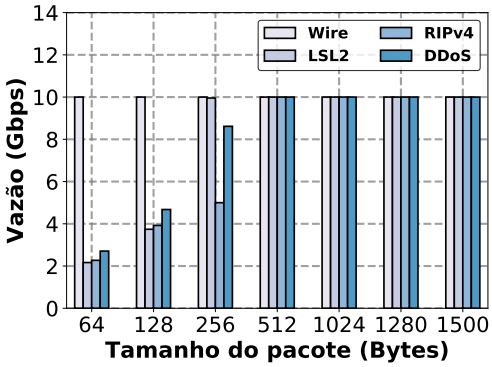

Figura 9. Vazão.

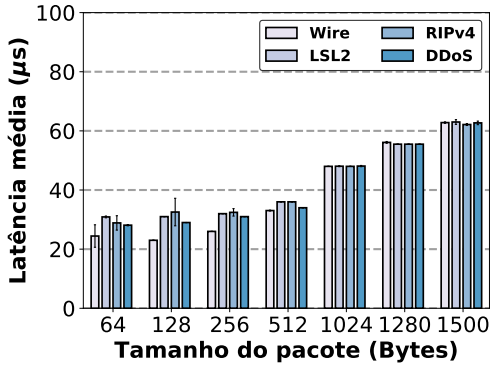

Figura 10. Latência.

Processamento *Serverless*: Neste experimento avaliamos a funcionalidade da computação *Serverless* do sistema. Enviamos duas requisições do programa Wire ($P1$ e $P2$) para serem executados por 5 segundos (s) cada. Wire apenas encaminha pacotes para as portas 0 e 1 da NetFPGA. O primeiro programa enviado foi $P1$ que encaminha o tráfego para a porta 1. Em $t = 5$ s, após o tempo expirar o escalonador carrega $P2$ que encaminha o tráfego para porta 0. A figura 7 mostra a vazão recebida nas portas 0 e 1 do gerador de tráfego conectadas na NetFPGA. A troca de $P1$ para $P2$ ocorreu no tempo $4,999$ s. Após a troca não ocorreu perda de pacotes, demonstrando que o agente e escalonador alternam entre funções dinamicamente.

Tempo de inatividade zero: Avaliamos a capacidade do sistema para executar um novo código com tempo de inatividade zero usando o DBS. A figura 8 mostra a

vazão (em Gbps) antes, durante e após o carregamento de um novo programa eBPF. Enviamos pacotes de 512 Bytes a uma taxa constante de 10 Gbps e trocamos o programa do processador em tempo de execução. Antes do início do experimento, carregamos o programa mitigação DDoS na memória de instruções. Em $t = 5$ s, carregamos o mesmo programa novamente. Nenhum pacote foi descartado nesse processo, mostrando que o sistema provê tempo de inatividade zero durante troca de programas.

Vazão: A figura 9 apresenta a vazão para pacotes de 64 a 1500 Bytes para cada função. A função Wire alcança vazão máxima para todos os tamanhos de pacotes, sendo o esperado devido à simplicidade da função e usar apenas oito instruções. Entre pacotes de 512 a 1500 Bytes, o *overhead* de processamento reduz com o aumento do tamanho do pacote, chegando na vazão máxima. Para pacotes de 64 a 256 Bytes, a vazão não alcança os 10 Gbps devido às operações *lookup* e *update* na TCAM/CAM gastarem vários ciclos, reduzindo a vazão. O experimento com pacote de 256 Bytes para LSL2 chega próximo dos 10 Gbps e para RIPv4 não, porque operações com TCAM gastam mais ciclos que operações com CAM.

Latência: Além da vazão, medimos a latência média de cada função (figura 10) usando *pktgen-DPDK* com precisão de 1 μs. Para cada tamanho de pacote repetimos o experimento 10 vezes para apresentar o desvio padrão no gráfico. Como esperado, a latência aumentou devido ao número de instruções de cada função, pois o processador leva mais tempo para executar o programa. As barras da figura 10 representam o desvio padrão, que foi próximo de zero na maioria dos casos, demonstrando que existe pouca alteração no tempo de processamento, resultando menor instabilidade do sistema.

Energia: Quando inativo a NetFPGA SUME consome 16 W. Após o projeto ser sintetizado, o consumo de energia passa de 16 para 22 W independente da taxa de pacotes ou programa executado. O sistema tem consumo de energia por interface de 5,5 W ou $0,55$ W/Gbps. O consumo energético da NetFPGA SUME é o mesmo independente da taxa de pacotes e código executado no hardware. O valor energético da placa foi baseado no manual que disponibiliza um valor teórico, e Xilinx Vivado que apresenta uma estimativa do consumo após o projeto ser sintetizado. Dispositivos como SmartNic Netronome, Intel Core i7-7700, P4 Wedge 100BF-32X e servidor x86 1U têm consumo de até 40 W, 65 W, 436 W, e 300 W a 350 W. Nosso sistema tem eficiência energética melhor que os demais dispositivos citados acima. Infelizmente, até o momento não existem ferramentas para medir o consumo direto na placa.

5. Trabalhos relacionados

***Serverless*:** E3 [Liu et al. 2019] estuda o uso de SmartNICs em microsserviços para economizar energia. λ-NIC [Choi et al. 2019] apresenta um *framework Serverless* baseado no modelo de abstração casamento-Lambda implementado sobre uma SmartNic, permitindo várias funções serem executas em paralelo. Como apresentado na seção 4, SmartNICs consomem mais energia que o sistema proposto. E3 e λ-NIC têm processamento *Serverless* com *offloading* em hardware. No entanto, nenhum deles estão disponíveis para compararmos com nosso sistema. Em Singhvi et al. [Singhvi et al. 2017] são discutidos algumas restrições de arquitetura ao processar funções de rede como microsserviços.

Linguagens de domínio específico: A linguagem P4 [Bosshart et al. 2014] adota o modelo de abstração casamento-ação. Ela pode ser utilizada para gerar instruções eBPF através de um compilador P4 para eBPF [Budiu 2015]. Domino [Sivaraman et al. 2016] é uma linguagem de alto nível que pode ser compilado dentro do Banzai, um modelo de máquina de baixo nível projetado para comutadores com taxa de linha. Ambas linguagens tem registradores pequenos e rápidos para armazenar estados, e fornecem um conjunto restrito de funcionalidades para maioria das funções com estado.

Casamento-ação com múltiplos estágios: RMT [Bosshart et al. 2013], dRMT [Chole et al. 2017], e FlexPipe [Ozdag 2012] são chips reconfiguráveis baseados no modelo de processamento casamento-ação. RMT e dRMT fornecem uma arquitetura com tabelas de casamento reconfiguráveis para processadores de pacote com múltiplos estágios implementados em ASIC. RMT não pode executar expressões quando a operação de análise manipula o payload do pacote. Intel's FlexPipe é um comutador em chip que fornece uma arquitetura com pipeline de 32 estágios casamento-ação programável através de um microcode dedicado. A programabilidade desta arquitetura não é direta. Tofino [bar 2016] do Barefoot e Xpliant [cav 2014,cav 2016] do Cavium são produtos comerciais que oferecem programabilidade em alta velocidade (Tbit/s) usando um pipeline.

FPGAs: P4→NetFPGA [Ibanez et al. 2019] apresenta o fluxo de trabalho de programas P4 sobre a plataforma NetFPGA SUME para processamento de pacotes em taxa de linha. P4→NetFPGA é limitado pois o processamento de funções sem estado ocorre fora do hardware, aumentando o tempo de processamento. Além disso, não tem tempo de inatividade zero na troca de programas, prejudicando a vazão com a perda de pacotes. ClickNP [Li et al. 2016] foca no aumento da flexibilidade da programabilidade. Para reprogramar o ClickNP são gastos $\sim$1-2 horas devido às ferramentas de síntese. FlowBaze [Pontarelli et al. 2019] processa pacotes usando estados em uma FPGA via programação usando máquinas de estado finito estendido. FlowBlaze não é Turing completo e modifica analisador e tabelas de transição sendo necessário uma nova síntese.

Smart NICs: SoftNIC [Han et al. 2015] fornece uma interface NIC programável com Click [Kohler et al. 2000]. SoftNic é uma abordagem diferente para prover programabilidade na rede, fornecendo novas funcionalidades para NICs hospedeiros através do software. Plataforma de filtragem virtual (VFP) [Firestone 2017] apresenta ação-casamento com abstração *offloading* para SmartNic nos datacenters da Azure. Netronome [Beckett et al. 2018] fornece uma SmartNIC que pode ser programado com instruções eBPF. Isso mostra a adoção do eBPF pela indústria.

6. Conclusões e trabalhos futuros

Foi projetado e implementado um sistema de processamento de pacotes *Serverless* que permite processar funções de rede sobre a plataforma NetFPGA SUME, via espaço de usuário. Neste sistema integramos o OpenFaaS e desenvolvemos ferramentas para automatizar e gerenciar funções criadas pelo usuário de modo escalável e programável. Além disso, para melhorar o desempenho do sistema realizamos otimizações de hardware, como processador eBPF com *pipeline*, sistema de memória com *buffer* duplo, estrutura FIFO_MD para cópia zero de pacotes, e uso de mapas. Nossos resultados mostram que o sistema opera em taxa de linha de modo programável via espaço de usuário.

Como trabalhos futuros, avaliaremos outros estudos de caso, por exemplo, funções de rede complexas (sistema de detecção de intrusão) e métricas de desempenho da rede. Além disso, mediremos o sistema com e sem otimização para apresentar o ganho de desempenho das otimizações realizadas.

Agradecimentos

Agradecemos as agências de pesquisa CNPq, FAPESP projeto 2018/23085-5 e FAPEMIG pelo apoio financeiro. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

Referências

(2014). Cavium attacks broadcom in switches. https://www.eetimes.com/document.asp?doc_id=1323931. Acessado em 28/01/2018.

(2016). Barefoot: The world's fastest and most programmable networks. `https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/`. Acessado em 28/01/2018.

(2016). Xpliant ethernet switch product family. `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`. Acessado em 28/01/2018.

(2017). The power of smartnics. `https://www.lightreading.com/nfv/nfv-elements/the-power-of-smartnics/d/d-id/738300`. Acessado em 08/08/2019.

Aditya, P., Akkus, I. E., Beck, A., Chen, R., Hilt, V., Rimac, I., Satzke, K., and Stein, M. (2019). Will serverless computing revolutionize nfv? *Proceedings of the IEEE*, 107(4):667–678.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. (2017). Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer.

Beckett, D., Joubert, J., and Horman, S. (2018). Host dataplane acceleration (hda).

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.

Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. (2013). Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 99–110, New York, NY, USA. ACM.

Budiu, M. (2015). Compiling p4 to ebpf.

Choi, S., Shahbaz, M., Prabhakar, B., and Rosenblum, M. (2019). Lambda-nic: Interactive serverless compute on smartnics. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, SIGCOMM Posters and Demos '19, pages 151–152, New York, NY, USA. ACM.

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.

Dargahi, T., Caponi, A., Ambrosin, M., Bianchi, G., and Conti, M. (2017). A survey on the security of stateful sdn data planes. *IEEE Communications Surveys & Tutorials*, 19(3):1701–1725.

Duarte, L. F. S. (2019). *alpine-ebpf*: An alpine-based image for compiling and running ebpf programs. `https://hub.docker.com/repository/docker/lucasfsduarte/alpine-ebpf`.

Ellis, A. (2016). Openfaas: Serverless functions made simple for docker and kubernates. `https://www.openfaas.com/`.

Firestone, D. (2017). VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA. USENIX Association.

Foundation, T. A. S. (2017). Apache OpenWhisk. `http://openwhisk.org/`. Acessado em Maio de 2019.

Google (2017). Google Cloud Functions. `https://cloud.google.com/functions/`. Acessado em Maio de 2019.

Han, S., Jang, K., Panda, A., Palkar, S., Han, D., and Ratnasamy, S. (2015). Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley.

Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2016). Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, pages 1–6, Denver, CO. USENIX Association.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

Ibanez, S., Brebner, G., McKeown, N., and Zilberman, N. (2019). The p4->netfpga workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 1–9, New York, NY, USA. ACM.

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297.

Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. (2016). Clicknp: Highly flexible and high performance network processing with

reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 1–14, New York, NY, USA. ACM.

Liu, M., Peter, S., Krishnamurthy, A., and Phothilimthana, P. M. (2019). E3: Energy-efficient microservices on smartnic-accelerated servers. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 363–378.

Microsoft (2017). Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`. Acessado em Maio de 2019.

Ozdag, R. (2012). Ethernet switch fm6000 series- software defined networking.intel corporation.

Pacífico, R. D., Coelho, G. R., Vieira, M. A., and Nacif, J. A. (2018). Roteador sdn em hardware independente de protocolo com análise, casamento e ações dinâmicas. In *Simpósio Brasileiro de Redes de Computadores (SBRC)*, volume 36.

Pontarelli, S., Bifulco, R., Bonola, M., Cascone, C., Spaziani, M., Bruschi, V., Sanvito, D., Siracusano, G., Capone, A., Honda, M., Huici, F., and Siracusano, G. (2019). Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA. USENIX Association.

Services, A. W. (2017). AWS Lambda. `https://aws.amazon.com/lambda/`. Acessado em Maio de 2019.

Singhvi, A., Banerjee, S., Harchol, Y., Akella, A., Peek, M., and Rydin, P. (2017). Granular computing and network intensive applications: Friends or foes? In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 157–163, New York, NY, USA. ACM.

Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. (2016). Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA. ACM.

Vieira, A. G., Pereira, G. H. A., Freire, J. H. F., Duarte, L. F. S., Pacífico, R. D. G., Pantuza, G., Vieira, M. A. M., Vieira, L. F. M., and Nacif, J. A. M. (2020a). Computação Serverless: Conceitos, Aplicações e Desafios. In *Minicursos do XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*, Rio de Janeiro, RJ, Brasil. SBC.

Vieira, M. A. M., Castanho, M. S., Pacífico, R. D. G., Santos, E. R. S., Câmara Júnior, E. P. M., and Vieira, L. F. M. (2019). Processamento Rápido de Pacotes com eBPF e XDP. In *Minicursos do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*, Porto Alegre, RS, Brasil. SBC.

Vieira, M. A. M., Castanho, M. S., Pacífico, R. D. G., Santos, E. R. S., Júnior, E. P. M. C., and Vieira, L. F. M. (2020b). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1).

Xilinx (2010). Virtex-7 family overview.

Xilinx (2011). Axi reference guide.

Zilberman, N., Audzevich, Y., Covington, G. A., and Moore, A. W. (2014). Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41.