# Calisto: Sistema para processamento rápido de pacotes com baixa latência em centro de dados

**Lucas A. C. Bleme**[1], **Gustavo Pantuza** [1],
**Marcos Augusto M. Vieira** [1], **Luiz Filipe M. Vieira** [1]

[1] Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brasil

andreybleme1@gmail.com, {pantuza,mmvieira,lfvieira}@dcc.ufmg.br

***Abstract.*** *Supporting low latency processing in high throughput environments is a growing trend in recent years. Shenango allows servers to fastly process network packets, preserving high CPU efficiency through a congestion detection algorithm and using its IOKernel, a dedicated component that orchestrates the packet allocation between new cores. This system though, allocates new cores using the round-robin scheduler policy. This work presents Calisto, a new system that implements an efficient scheduler policy that reduces processing overhead and promoting packet flow affinity, reducing the latency by up to 10%.*

***Resumo.*** *A demanda por suporte a baixa latência em ambientes de alta vazão tem aumentado nos últimos anos. Sistemas como Shenango permitem que servidores processem pacotes rapidamente, mantendo a eficiência na utilização de CPU através de um algoritmo de detecção de congestionamento e o IOKernel, componente dedicado a orquestrar a alocação de novos núcleos. O sistema porém aloca núcleos utilizando a política de escalonamento round-robin. Neste trabalho, apresenta-se Calisto, um novo sistema que implementa uma política eficiente de distribuição de pacotes entre núcleos que mantém afinidade de fluxos, reduzindo a latência em até 10%.*

## 1. Introdução

Para prover respostas de baixa latência, aplicações em *data centers* precisam suportar uma alta taxa de requisições. Mesmo com o *hardware* de rede tendo evoluído consideravelmente nos últimos anos sendo capaz de prover *round trip times* (RTT) na ordem de grandeza de microsegundos, aplicações executadas com estas placas de rede ainda apresentam taxa de latência na ordem de milisegundos. Sistemas existentes utilizam diferentes técnicas para aumentar a taxa de vazão no processamento de pacotes em aplicações em data centers.

Através da técnica de *kernel offload*, aplicações podem transferir parte de sua carga de trabalho para o núcleo do sistema operacional. O *kernel* do Linux permite através da máquina virtual eBPF [Vieira et al. 2020] que o processamento de pacotes possa ser customizado respeitando regras definidas por uma aplicação em espaço de usuário. Em outras abordagens, o código especializado pode ser transferido para uma *smart nic* através da linguagem de programação P4 [Bosshart et al. 2014]. Utilizando *kernel-bypass* o sistema operacional ZygOS [Prekas et al. 2017] por exemplo é capaz de suportar uma alta

taxa de requisições. Entretanto, este sistema assim como similares, monitora constantemente a interface de rede (NIC) para detectar a chegada de novos pacotes, técnica conhecida como *spin-polling*. Isto faz com que uma proporção significativa de CPU seja desperdiçada mesmo enquanto nenhum pacote é recebido para ser processado.

O sistema Shenango [Ousterhout et al. 2019], por outro lado, dedica um núcleo exclusivamente para a execução de um componente dedicado a receber, de forma contínua, novos pacotes a serem processados pelas aplicações. Utilizando um algoritmo de detecção de congestionamento de computação, o Shenango realoca núcleos de processamento disponíveis entre aplicações de um mesmo servidor conforme mostra a Figura 1. Este mecanismo de alocação de novos núcleos, detecta a necessidade de se distribuir pacotes entre diferentes núcleos para aumentar a vazão conforme rajadas de pacotes são recebidas. Esta distribuição de pacotes entre núcleos porém, utiliza uma política de balanceamento que não considera a afinidade entre os fluxos de pacotes. Pacotes pertencentes a um mesmo fluxo, isto é, que compartilham das mesmas características, como blocos de dados TCP por exemplo, idealmente são processados pelo mesmo núcleo. Isto reduz a necessidade de sincronização e comunicação entre diferentes núcleos que estejam processando pacotes de um mesmo fluxo, que quando ocorre em excesso, pode levar a contenção de CPU [Hood et al. 2010] e sub-utilização das linhas de cache. Afinidade de fluxo evita inter-comunicação entre núcleos.

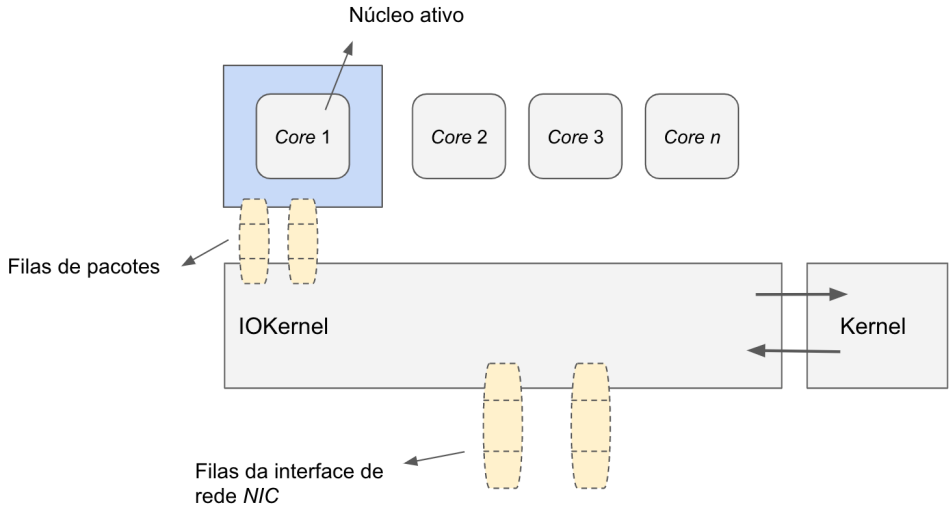


**Figura 1. Arquitetura do sistema Shenango**

A contribuição principal deste artigo é um sistema capaz de utilizar qualquer política de mapeamento de fluxos em núcleos dinamicamente. Para isso, utilizamos o conceito de nível de indireção. Este trabalho apresenta o sistema Calisto, que utiliza uma política de alocação de novos núcleos para processamento rápido de pacotes que agrupa pacotes de um mesmo fluxo em um mesmo núcleo, enquanto aloca núcleos dinamicamente para aplicações em uma escala de microsegundos. Utilizando uma tabela de indireção, os pacotes recebidos são encaminhados ao núcleo responsável, reduzindo assim, a quantidade de núcleos diferentes envolvidos no processamento de um mesmo

fluxo de pacotes. O sistema Calisto é capaz de realizar alocação dinâmica de pacotes de rede entre núcleos enquanto mantém sua afinidade de fluxo, o que promove melhoria de até 10% na latência quando comparado ao Shenango, ferramenta tida como estado da arte no processamento de pacotes em aplicações *multi-thread*.

Este trabalho apresenta os resultados de experimentos realizados em um servidor executando o sistema Calisto enquanto processa pacotes TCP de múltiplos fluxos. Por utilizar uma modificação do componente *IOKernel* do Shenango, o sistema Calisto se beneficia deste componente centralizado, dedicado a enviar novos pacotes a serem processados da NIC para as aplicações utilizando a técnica conhecida como *kernel-bypass*, que elimina a sobrecarga envolvida no processamento de pacotes no *kernel* movendo este processamento para o espaço de usuário do sistema operacional.

Este documento está organizado da seguinte forma: na Seção 2 são apresentados os trabalhos relacionados. Na Seção 3 é descrita a ferramenta Shenango, que representa o estado da arte no tópico deste artigo. Na Seção 4 é apresentado o sistema Calisto, a arquitetura proposta e sua implementação. Na Seção 5 são descritos os experimentos realizados para avaliar o desempenho do sistema Calisto comparado ao Shenango. São discutidos também os resultados. As conclusões são apresentados na Seção 6. Finalmente, a Seção 7 descreve os trabalhos futuros.

## 2. Trabalhos Relacionados

Para decidir como alocar núcleos e *threads* entre diferentes aplicações, sistemas prévios utilizam componentes controladores de recursos que monitoram métricas de desempenho para realizar realocação, como fazem os sistemas Arachne [Qin 2019] e IX [Belay et al. 2014]. Como estas métricas são coletadas na ordem de grandeza de milissegundos, a granularidade não é suficiente para lidar com alocações em latência de cauda. Ambos sistemas adotam a abordagem de manter núcleos alocados para aplicações mesmo que estas estejam ociosas ou ocupadas, causando desperdício de ciclos de CPU.

Outra abordagem conhecida como *kernel bypass* é aplicada em sistemas como ZygOS [Prekas et al. 2017] e Arrakis [Peter et al. 2014]. Estes sistemas contornam o kernel para alcançar baixa latência no processamento de pacotes de rede, evitando que o processamento exija troca de contexto entre núcleo e espaço de usuário. Nenhum destes sistemas porém, provê alocação de novos núcleos de maneira dinâmica como faz o Shenango [Ousterhout et al. 2019]. Em contrapartida, adotam a abordagem de estaticamente particionar núcleos entre aplicações. Nenhum dos dois projetos promove afinidade de fluxo durante o processamento de pacotes como faz o Calisto.

O projeto AccelTCP [Moon et al. 2020] realiza *offload* da pilha TCP/IP para interfaces de redes inteligentes *(smart nics)*. Com essa abordagem, transfere-se parte da complexidade do protocolo TCP para ser processada diretamente na interface de rede. O presente trabalho também busca reduzir a troca de contexto enquanto mantém alta taxa de sucesso no processamento de pacotes. No entanto, através do DPDK [DPDK 2021] juntamente ao Calisto, faz-se *kernel bypass* ao invés de *kernel offload*.

O sistema Danian [Pantuza et al. 2021] apresenta uma estratégia de escolha de *threads* através da técnica *memoization* na estrutura de dados dos processos. O sistema aumenta o uso de memória do escalador e reduz o tempo de computação do algoritmo de

escalonamento de *threads*. Este projeto porém, também não considera a afinidade entre os fluxos de pacotes em seu contexto.

O artigo RSS++ [Barbette et al. 2019], apresenta a implementação de um sistema com uma política de balanceamento de carga que modifica dinamicamente a busca na tabela do RSS (*Receive Side Scaling*). Essa política distribui a carga entre os núcleos (CPUs) seguindo um modelo matemático para escalonamento de fluxos em núcleos. O RSS++ precisa que a placa de rede tenha suporte para o RSS. Diferentemente do sistema RSS++, o sistema Calisto utiliza um núcleo dedicado para despachar tráfego a múltiplas aplicações que se conectam ao sistema através do componente *IOKernel*. Ao invés disso, o RSS++ faz o despacho direto de pacotes para as filas, sendo sua proposta principal, atender sistemas com múltiplos núcleos alocados para uma aplicação sendo executada em todos os núcleos. O RSS++ não tem ciência das aplicações sendo executadas. Ele assume, portanto, que as aplicações estão sendo executadas em todos os núcleos. Por exemplo, se tivermos um servidor web sendo executado apenas no núcleo 2 e o sistema *memcached* apenas no núcleo 3, quando um cliente abrisse uma conexão HTTP, o módulo RSS++ ao receber o primeiro pacote, não saberia para qual núcleo encaminhar. O módulo RSS++ apenas calcula o hash e envia para o núcleo correspondente do hash.

## 3. Fundamentação Prática: Shenango

Para melhor entendimento do artigo, esta Seção descreve o funcionamento básico do sistema Shenango que foi utilizado como base para o sistema Calisto. Para detectar novos pacotes a serem processados, o componente *IOKernel* modificado a partir da implementação disponibilizada pelo Shenango[Ousterhout et al. 2019], atua como um intermediário entre as aplicações que utilizam o Calisto e as filas em hardware da interface de rede que recebe novos pacotes. Estando constantemente monitorando as filas da interface de rede, o *IOKernel* recebe novos pacotes e os envia a cada aplicação (*runtime*), que se comunicam com o *IOKernel* através de um espaço de memória compartilhado com suas filas de pacotes, como demonstra a Figura 1.

Sendo o *IOKernel* responsável por detectar a chegada de novos pacotes e encaminhá-los às aplicações, neste componente se encontram as funções que implementam as decisões de balanceamento do envio de pacotes entre diferentes *threads*, núcleos e aplicações. Assim, o sistema Shenango envia cada pacote recebido a uma *thread* de processamento utilizando a função *rx_send_to_runtime* como demonstrado pelo Algoritimo 1, retirado do sistema Shenango. Este código está publicamente disponível na plataforma *Github*.

```
1 bool rx_send_to_runtime(struct proc *p, uint32_t hash, ...)
2 {
3   struct thread *th;
4   ...
5   /* load balance between active threads */
6   th = p->active_threads[hash % p->active_thread_count];
7
8   return lrpc_send(&th->rxq, cmd, payload);
9 }
```

**Listing 1. Função original do sistema Shenango, executada a cada novo pacote recebido**

O parâmetro *hash* desta função é corresponde à chave *RSS (Receive Side Scaling)*, valor inteiro que identifica unicamente pacotes que possuem as mesmas características, como endereço de origem e destino, porta de origem e destino, e o protocolo.

Utilizando uma operação de módulo entre o *hash RSS* do pacote e a quantidade de *threads* ativas no processo corrente, como demonstra a linha 6 do trecho de código acima, pacotes contendo o mesmo *hash* podem ser enviados a *threads* de núcleos diferentes. Considerando 4 diferentes fluxos de pacotes com *hashes*: 4, 5, 6 e 7, por exemplo. Quando existirem no Shenango 4 threads ativas, estes pacotes serão processados nos núcleos 0, 1, 2 e 3, respectivamente, como mostra a Figura 2.

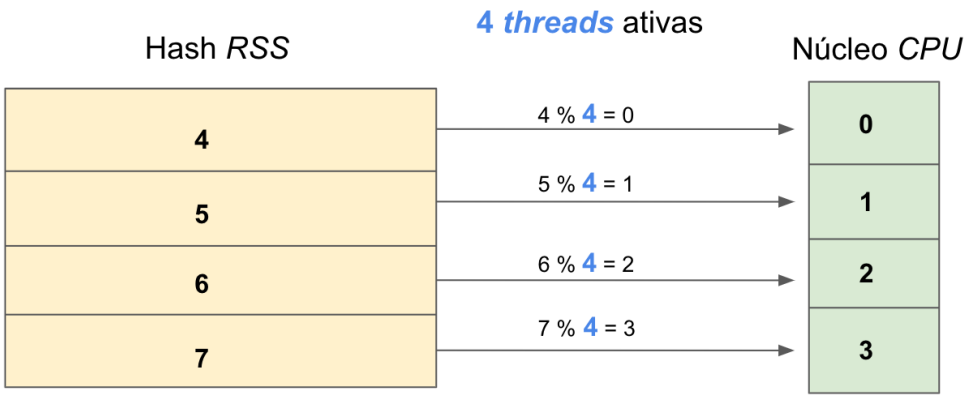


**Figura 2. Shenango processando pacotes com 4 *threads* ativas**

Caso a quantidade de *threads* ativas aumente de 4 para 5, os pacotes então serão realocados para diferentes núcleos: 4, 0, 1 e 2, como mostra a Figura 3.

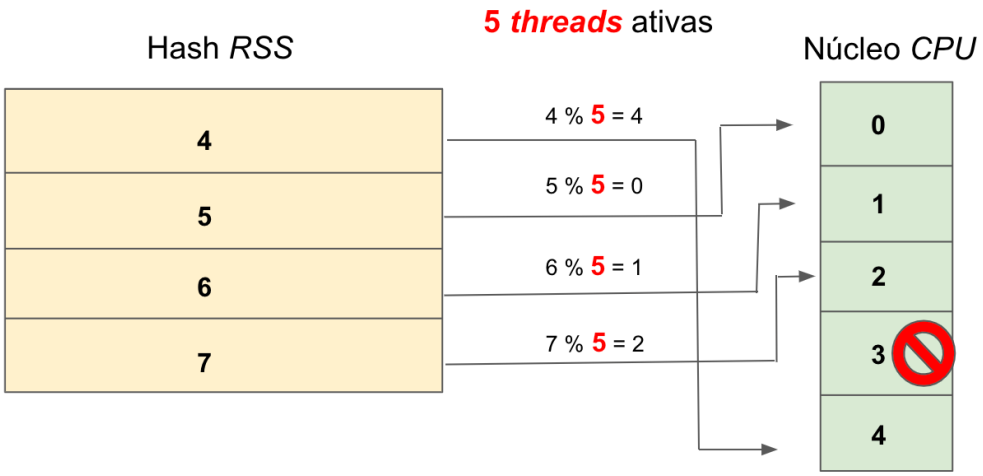


**Figura 3. Mudança de núcleos com aumento na quantidade de *threads* ativas**

O simples aumento na quantidade de *threads* ativas no sistema causou a mudança de núcleos de processamento de pacotes que, para reduzir a sobrecarga de mudança de contexto, deveriam continuar sendo processados por *threads* do mesmo núcleo.

## 4. Calisto

Para que pacotes de um mesmo fluxo sejam processados pelo mesmo núcleo, o sistema Calisto apresenta mudanças significativas no componente *IOKernel*. Entre o recebimento de um novo pacote e o encaminhamento deste pacote a uma *thread* pertencente a um núcleo, foi adicionada uma nova política de balanceamento de pacotes entre as *threads*.

A começar pela alteração na estrutura de dados que representa um processo, foi acrescentado uma *Hash Table* (*hash_to_core*) que armazena *hashes* que identificam pacotes, com seus núcleos correspondentes:

```
struct proc {
  pid_t pid;
  ...
  /* the flow hash table */
  struct rte_hash *hash_to_core;
}
```

**Listing 2. Processo original do Shenango, modificado para incluir a tabela hash**

Quando o sistema Calisto é iniciado, a *Hash Table* é configurada contendo chaves do tamanho de uma variável do tipo *uint32_t* e suportando uma quantidade máxima de até 700 mil registros. A função *rx_send_to_runtime*, executada a cada novo pacote detectado a ser processado, foi modificada para que a cada novo pacote recebido, esta tabela de indireção seja consultada a fim de que pacotes com mesmo *hash* sejam destinados às *threads* do mesmo núcleo:

```
unsigned int affinity_core = 0;

bool rx_send_to_runtime(struct proc *p, uint32_t hash, ...)
{
    int ret1;
    int reta;
    struct thread *th;
    int assigned_core;

    /*
     * If current RSS hash has no affinity core assigned,
     * add it in Hash Table. Otherwise use the assigned core
     */
    ret1 = rte_hash_lookup(p.hash_to_core, (const void *)&hash);
    if (ret1 < 0) {
        /*
         * Reset affinity to assign if max CPU allowed is reached
         */
        if (affinity_core == NCPU) {
            affinity_core = 1;
        }
        reta = rte_hash_add_key_data(p.hash_to_core, (const void *)&
    hash, affinity_core);
        assigned_core = affinity_core;
```

```
24          affinity_core++;
25          if (reta < 0)
26              log_err("rx: failed to add HASH to hash table in
        rx_send_to_runtime");
27      } else {
28          assigned_core = ret1;
29    }
30
31      th = cores_add_core(p, assigned_core);
32      return lrpc_send(&th->rxq, cmd, payload);
33  }
```

**Listing 3. Função do Calisto executada a cada novo pacote recebido**

A cada novo pacote, uma pesquisa *lookup* é realizada na tabela de indireção *hash_to_core* afim de saber se já existe um núcleo designado para o presente *hash*. Caso ainda não exista um núcleo designado para o pacote, um novo núcleo é escolhido e adicionado na tabela utilizando a função *rte_hash_add_key_data* conforme mostra a linha 22. Caso o presente pacote já tenha um núcleo designado, isto é a chave com o *hash* do pacote seja encontrado na *Hash Table*, a variável *assigned_core* recebe o valor correspondente a esta chave, sendo este valor o identificador inteiro do núcleo.

Tendo qual núcleo deve ser atribuído ao presente pacote, basta invocar a função *cores_add_core* que retorna uma nova *thread* para o núcleo (*assigned_core*) enviado via parâmetro.

### 4.1. Arquitetura proposta

Para criar o direcionamento entre os *hashes* dos pacotes e seu núcleos, o sistema Calisto utiliza a estrutura de dados *Hash Table* com um nível de direcionamento conforme mostra a Figura 4.

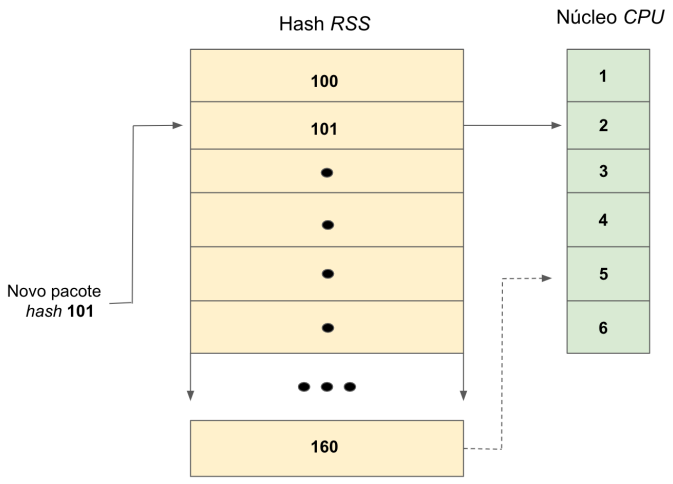


**Figura 4. Nível de indireção da associação entre *hash* e núcleo**

Cada chave da tabela corresponde a um *RSS hash* de um pacote, e seu valor corresponde a um núcleo existente no computador onde o sistema é executado. Sempre que

um novo pacote é recebido, a tabela é pesquisada através da chave *RSS hash* do pacote e o valor encontrado (núcleo) é retornado.

Quando nenhum valor é encontrado na tabela para o *hash* do pacote recebido, significa que esta é a primeira vez que um pacote deste fluxo chega ao sistema para ser processado. Neste caso, este novo *hash* é inserido na tabela e o núcleo utilizado nesta inserção será atribuído a todos os pacotes recebidos deste mesmo fluxo, isto é, que possuam o mesmo *RSS hash* que compõe a chave na tabela de direcionamento.

## 5. Análise e experimentação

Para medir o desempenho do sistema Calisto, foi realizado um experimento com a aplicação *synthetic* disponível no código fonte do Shenango. Uma aplicação funciona como cliente e outra como servidor, que processa os milhares de pacotes enviados a cada segundo pelo cliente, conforme demonstra a topologia na Figura 5.

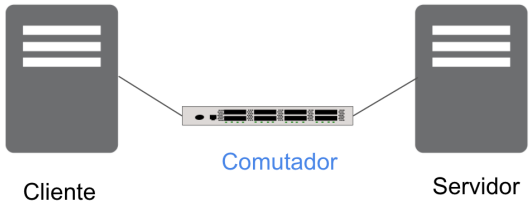


**Figura 5. Topologia do experimento no Cloudlab**

Este experimento foi executado no ambiente CloudLab [Duplyakin et al. 2019] com duas máquinas (*node-1* e *node-0*) conectadas através de um *switch*. Cada máquina possui um processador 8 *dual core* Intel Xeon 2.0 GHz com 16 núcleos, 64 Gb de memória RAM sendo 4x 16 Gb e uma interface de rede Mellanox ConnectX com 2 portas de 10 Gb cada conforme a tabela 1.

| **Máquina** | **CPU** | **Memória RAM** | **Disco** | **Interface de Redes** |
|---|---|---|---|---|
| m510 | 8 dual core Intel Xeon D-1548 à 2.0 GHz | 64 Gb 4x 16 GB DDR4 | 256 GB | Dual-port Mellanox ConnectX Portas de 10 Gb |

**Tabela 1. Configuração das máquinas utilizadas nos experimentos**

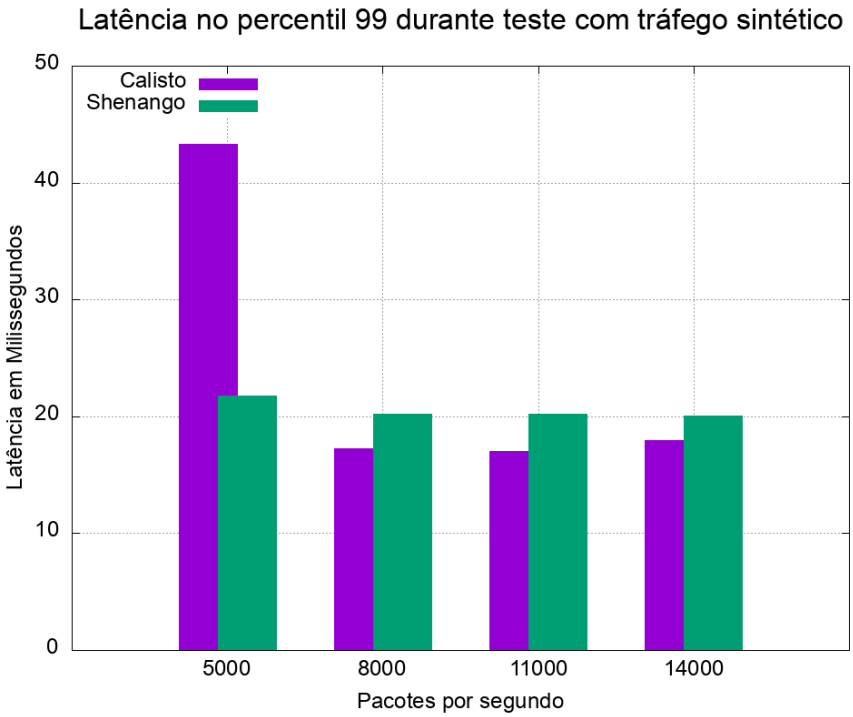


**Figura 6. Latência de cauda no percentil 99 da execução com Synthetic**

Foi medida a latência de resposta do servidor ao receber rajadas de milhares de pacotes enviados pelo cliente de maneira contínua. Conforme a latência média se mostra estável para uma dada quantidade de pacotes por segundo, o valor da média da latência é coletado e o número de pacotes enviados por segundo é aumentado progressivamente. Para este experimento foram utilizados aproximadamente 700 mil diferentes fluxos de pacotes, alocados de maneira dinâmica entre os 10 núcleos disponíveis nas máquinas utilizadas. A Figura 6 apresenta os resultados comparativos de desempenho entre os sistemas Calisto e Shenango durante a execução deste experimento.

O sistema Calisto teve desempenho inferior no início do recebimento dos pacotes para processamento como apresenta a Figura 6. Enquanto a aplicação cliente enviava 5 mil pacotes por segundo, o sistema Shenango apresentou latência significantemente menor. A razão principal desta grande diferença no início do recebimento dos pacotes está na política de realocação do sistema Calisto. Diferentemente do Shenango, o sistema Calisto precisa primeiro adicionar na tabela de indireção (*hash_to_core*) os novos direcionamentos entre *hash* e núcleos. Como a tabela está vazia para os primeiros fluxos de pacotes recebidos, são executadas duas operações a cada novo pacote: uma de pesquisa e outra de inserção na tabela. Estas operações não acontecem no sistema Shenango, e, por isto, esta sobrecarga inicial causa diferença na latência coletada até os primeiros 5 mil pacotes por segundo.

Quando avaliamos porém o desempenho quando a vazão dos sistemas chega a 8 mil pacotes por segundo, observa-se que o sistema Calisto é capaz de processar os pacotes com menor latência. Esta diferença se mantém linear durante o processamento intensivo de pacotes, chegando até os 14 mil pacotes por segundo mantendo a latência até 10% mais baixa quando comparada com o Shenango, como apresentado no gráfico comparativo da

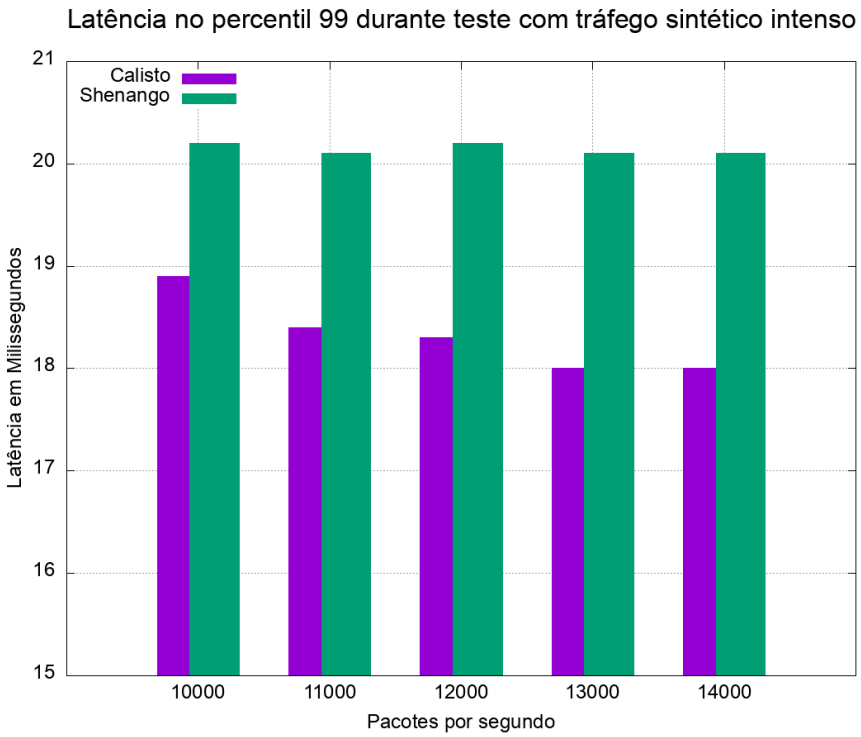


**Figura 7. Latência de cauda da execução com Synthetic em vazão de 10 mil a 14 mil pacotes por segundo**

Figura 7. Dessa maneira fica evidente que o sistema Calisto, após popular sua tabela de direcionamento e pagar este custo, se torna mais eficiente.

Conforme os pacotes de fluxos são dinamicamente alocados aos núcleos tendo sua afinidade de fluxo garantida pelo sistema Calisto, a sobrecarga de troca de contexto entre diferentes núcleos diminui e o sistema se mostra ainda mais eficiente conforme cresce o número de pacotes enviados por segundo.

## 6. Conclusão

Este artigo apresenta Calisto, um sistema que utiliza uma política de alocação de pacotes entre núcleos considerando sua afinidade de fluxo. Utilizando um nível de indireção para associar o *hash* de um pacote a um núcleo, o sistema Calisto se mostrou capaz de lidar com grandes quantidades de pacotes com latência até 10% mais baixa quando comparado ao sistema Shenango.

Utilizando uma estrutura de dados *Hash Table* com custo médio *O(1)* para operações de inserção e pesquisa, e um mecanismo que realiza realocação de pacotes entre núcleos de forma dinâmica, o sistema Calisto se mostrou capaz de servir de maneira eficiente aplicações em centro de dados onde a alocação de núcleos acontece de forma dinâmica.

O sistema Shenango é considerado estado da arte em eficiência no processamento rápido de pacotes em ambientes de centro de dados. Desta forma, o presente trabalho apresenta uma alternativa ainda mais eficiente em cenários de alta vazão, demonstrando a implementação do sistema Calisto, capaz de realizar processamento rápido de pacotes

com latência até 10% mais baixa.

O código fonte completo e os *scripts* para reproduzir os experimentos apresentados neste trabalho podem ser encontrados no link `https://github.com/andreybleme/calisto`.

## 7. Trabalhos Futuros

Para reduzir a sobrecarga inicial causada pelas múltiplas operações de pesquisa e inserção na tabela de indireção, a política de alocação de pacotes entre núcleos implementada no sistema Calisto pode ser alterada para conciliar a política que mantém a afinidade entre os pacotes com o *round-robin*. Nas execuções iniciais do sistema, ambas as políticas podem ser aplicadas de maneira alternada afim de reduzir a sobrecarga inicial de execução do sistema.

É possível também preparar a tabela de indireção a priori, fazendo com que, nas primeiras execuções do sistema Calisto, não sejam realizadas sempre duas operações: uma de pesquisa e outra de inserção, mas apenas uma de pesquisa. Realizar esta alteração executando novos experimentos pode apresentar resultados diferentes dos expostos neste trabalho, principalmente no resultado apresentado para o experimento com 5 mil pacotes por segundo.

O componente *IOKernel* modificado no Calisto faz uso de *POSIX threads*, estrutura que pode ser substituída por *lthreads* [Rushing's 2021], que são criadas em espaço de usuário e não requerem intervenção do *kernel*. Novos experimentos com esta estrutura podem apresentar resultados diferentes e seriam de grande valor no futuro.

Uma abordagem em que a tabela *hash* do Calisto possa ser previamente configurada de modo a já ter a política instalada no plano de dados poderia ser um experimento interessante na linha de pesquisa das redes definidas por *software* [Pantuza et al. 2014].

Outra possibilidade a ser explorada no futuro é a implementação de funções de rede *NFVs* eficientes utilizando o sistema Calisto, com experimentos realizados em interfaces de rede de velocidade 200 Gbps, submetendo o sistema a uma carga de trabalho ainda maior.

## 8. Agradecimentos

Agradecemos as agências de pesquisa CNPq, FAPESP projeto 2018/23085-5 e FAPEMIG pelo apoio na realização do presente trabalho.

## Referências

Barbette, T., Katsikas, G. P., Maguire, G. Q., and Kostić, D. (2019). Rss++: Load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 318–333, New York, NY, USA. Association for Computing Machinery.

Belay, A., Prekas, G., Primorac, M., Klimovic, A., Grossman, S., Kozyrakis, C., , and Bugnion, E. (2014). Ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14.

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.

DPDK (2021 (acessado em Janeiro de 2021)). *Data Plane Development Kit*.

Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. (2019). The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14.

Hood, R., Jin, H., Mehrotra, P., Chang, J., Djomehri, J., Gavali, S., Dennis Jespersen, K. T., and Biswas, R. (2010). Performance impact of resource contention in multicore systems. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–3.

Moon, Y., Lee, S., Jamshed, M. A., and Park, K. (2020). Acceltcp: Accelerating network applications with stateful TCP offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, Santa Clara, CA. USENIX Association.

Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. (2019). Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA. USENIX Association.

Pantuza, G., Bleme, L. A. C., Vieira, M. A. M., and Vieira, L. F. M. (2021). Danian: Tail latency reduction of networking application through an o(1) scheduler. In *26th IEEE Symposium on Computers and Communications (ISCC)*.

Pantuza, G., Sampaio, F., Vieira, L. F. M., Guedes, D., and Vieira, M. A. M. (2014). Network management through graphs in software defined networks. In *10th International Conference on Network and Service Management (CNSM) and Workshop*, pages 400–405.

Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., and Anderson, T. (2014). Arrakis: The operating system is the control plane. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14.

Prekas, G., Kogias, M., Kogias, M., and Bugnion, E. (2017). Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14.

Qin, H. (2019). The arachne distributed operating system. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14.

Rushing's, S. (2021 (acessado em Março de 2021)). *Coroutine library threads*.

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. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1).