Danian: Redução da latência de calda de aplicações em rede através de um escalonador O(1)

Gustavo Pantuza [1], Lucas A. C. Bleme[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

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

Abstract. *Core allocation for application threads is a problem of reasonable complexity and computacional cost inside Unix systems. Caladan scheduler is a solution aiming to reduce the cost of how threads and core are allocated in microsecond scale. The system Danian optimizes through memoization the thread picking algorithm that picks the best thread for a given core. This operation cost dropped from O(n) to O(1), the CPU time reduced 7%, the tail latency reduced 3% on Caladan Synthetic experiment and 5% on the Netperf experiment.*

Resumo. *A alocação de núcleos para threads de programas é um problema complexo e de grande custo computacional dentro de ambientes Unix. O escalonador Caladan é uma solução que otimiza na escala dos microssegundos a forma como threads e núcleos são escalonados. O sistema Danian proposto, melhora o algoritmo de escolha de threads para um núcleo a ser escalonado através de memoization. O custo dessa operação foi reduzido de O(n) para O(1), o tempo de CPU reduzido em 7%, a latência de calda reduziu em 3% para o experimento Synthetic e 5% para o experimento Netperf.*

1. Introdução

Caladan é um escalonador de CPU eficiente que consegue alcançar melhores qualidades de serviço (*QoS*) como latência de calda através de um agendador de *threads* centralizado [Fried et al. 2020]. Esse escalonador foi desenvolvido dentro do ambiente Shenango [Ousterhout et al. 2019]: um sistema para melhorar o desempenho da comunicação entre um pacote que chega a uma interface de redes até a *thread* da aplicação que irá computá-lo.

O Shenango tenta balancear a quantidade de núcleos dedicados a uma aplicação enquanto evita a Congestão de Computação: um algoritmo proposto por esse trabalho para maximizar o uso de núcleos, enquanto estiverem ociosos aguardando algum resultado de I/O, alocando-os para outras *threads*.

O impacto da latência de calda [Dean and Barroso 2013] no desempenho global de aplicações distribuídas em redes é um problema conhecido. Contudo, ainda em aberto para novas abordagens de mitigar ou minimizá-la em escala.

Algumas propostas tentam resolver esse problema através de transferência (*offload*) de carga de trabalho computacional para o núcleo de sistema operacional como por exemplo, no gancho XDP [Høiland-Jørgensen et al. 2018] da pilha TCP/IP do *kernel* do

Linux. Outras abordagem transferem a carga de trabalho diretamente para interfaces de rede inteligentes [Yang et al. 2020]. Em outros casos, transfere-se o código para ambos: *kernel* e interface de redes [Moon et al. 2020] através de soluções como as oferecidas pelas primitivas da linguagem P4 [Bosshart et al. 2014].

Nesse contexto, o Caladan é um escalonador de *threads* em CPUs que visa reagir a congestionamentos na escala dos microssegundos. O presente trabalho investiga o problema da escolha de *threads* dentro desse escalonador. O Danian escolhe uma *thread* inativa que tenha sido executada recentemente no núcleo a ser escalonado. Essa abordagem reduz o tempo de carga de uma *thread* em um núcleo, reduz a troca de contexto e reduz a latência de calda em aplicações distribuídas dentro de centros de dados.

O Danian tem como contribuição melhorar o desempenho do Caladan na escolha de *threads* através do uso de *memoization* na estrutura de dados dos processos. A otimização implementada aumenta o uso de memória do escalador e reduz o tempo de computação do algoritmo. Nos experimentos em rede o Danian mostrou reduzir em quase 8% a latência de calda e 5% o uso de CPU. Através de *memoization* o algoritmo do Danian reduz de $O(n)$ para $O(1)$ o custo para escolha de *threads*.

Esse documento está dividido da seguinte forma: Primeiro discute-se o estado da arte através da comparação dos trabalhos relacionados com o este trabalho. Em seguida, descreve-se o problema de escalonamento e algumas soluções já desenvolvidas para solucioná-lo. Posteriormente, apresenta-se a como o Caladan implementa seu algoritmo de escolha de *threads*. Depois, explica-se a solução proposta pelo Danian e como ela otimiza a escolha de *threads* por núcleo a ser escalonado. Em seguida, são apresentados o ambiente de experimentação e as ferramentas de avaliação. Depois, são demonstrados os resultados e análises, os potenciais trabalhos futuros e uma conclusão.

2. Trabalhos relacionados

O artigo RSS++ [Barbette et al. 2019] apresenta uma política de balanceamento de carga que modifica dinamicamente a indireção da busca na tabela do RSS (*Receive Side Scaling*). Essa política distribui a carga entre os núcleos (CPUs) de maneira ótima. O RSS++ pode ser implementado usando DPDK e executado em sistema operacional Linux tradicional em espaço de usuário (*user space*). Ele se mostra eficiente para fluxos que tem duração parecida. Nesses casos o RSS++ distribui a carga entre os núcleos de processamento de maneira eficiente. A proposta do presente trabalho não verifica as filas de pacotes na interface de redes. No entanto, na esfera do escalonador Caladan, implementa uma política que associa *threads* recentemente executadas em um dado núcleo de modo a minimizar a troca de contexto.

Algumas soluções para processamento rápido de pacotes utilizam a máquina virtual eBPF [Vieira et al. 2019] disponível no núcleo do sistema operacional Linux para fazer transferência (*offload*) de carga de trabalho para o núcleo do sistema operacional. Este trabalho vai na direção contrária: ao invés de transferir para o núcleo, faz-se o *kernel bypass* através da ferramenta DPDK [DPDK 2021].

O Caladan [Fried et al. 2020] é o escalonador objeto do presente trabalho de modo que através das otimizações para o algoritmo que associa *threads* a núcleos, investiga-se uma abordagem que possa reduzir a latência de aplicações em rede.

O Shenango [Ousterhout et al. 2019] é uma solução baseada em interrupções nas interfaces e que implementa um algoritmo de detecção de congestionamento de computação. Dessa foma o Shenango realoca núcleos para processos que estejam sob congestionamento. O presente trabalho não modifica essa camada de detecção de congestionamento de computação mas se aproveita dessa realocação de núcleos.

O trabalho denominado *Dune* [Belay et al. 2012] investiga a possibilidade de utilizar os recursos VT-x dos processadores da Intel às quais habilitam otimizações no *hardware* para otimizar o desempenho de processos no sistema operacional. Para tal, o autores descrevem um módulo para o kernel do Linux que faz a interface entre os processos e os núcleos através do uso de *Hypercalls*. Diferentemente, o presente trabalho não depende das chamadas de sistemas de *hypervisors*. Utiliza-se o DPDK para implementar o *kernel bypass*.

Aplicações em redes que são intensivas em dados como redes sociais, buscas textuais, *streaming* de vídeo são normalmente distribuídas em múltiplos servidores. Tentar minimizar a latência de calda é uma tarefa complexa e que impacta a experiência dos usuários na ponta final dessas comunicações conforme demonstrado em [Dean and Barroso 2013]. O presente trabalho se baseia nesse fato para avaliar as soluções de escalonamento propostos em função da latência de calda das aplicações medidas de modo a minimizá-la.

Otimizações na esfera dos algoritmos são formas complementares de otimizar soluções [Cormen et al. 2009]. A técnica de evitar computações repetitivas na forma de *memoization* é um recurso comumente utilizado em soluções baseadas em programação dinâmica [Bellman 1954]. Dessa forma, o presente trabalho implementa uma solução utilizando *memoization* para otimizar um algoritmo que apresenta computações repetitivas para escolher a melhor *thread* para um dado núcleo.

3. Escalonamento de *threads*

Escalonamento de recursos em sistemas *multi-thread* é um problema complexo. A decisão de qual aplicação/cliente priorizar durante a tarefa de multiplexação nem sempre é justa.

O sistema *Lottery* [Waldspurger and Weihl 1994], propõe um mecanismo de escalonamento de recursos randomizado. Assim como em uma loteria, cada cliente recebe um número finito de bilhetes. Em cada loteria, o recurso é alocado para o cliente sorteado. Essa abordagem garante uma alocação proporcional ao número de bilhetes que um cliente possui.

Outra solução avalia e discute o problema de que a gestão de *threads* não é eficiente nem no espaço de *kernel* nem no espaço de usuário. Não exclusivamente em cada um desses ambientes. O artigo propõe e avalia o *scheduler activation* [Anderson et al. 1991], um contexto de execução que disponibiliza vetores de controle do *kernel* para o espaço de endereçamento de uma aplicação em espaço de usuário fazendo uso de eventos do *kernel*. Ou seja, um par entre núcleo e espaço de usuário para maximizar o escalonamento de *threads*.

As propostas acima não mitigam o problema de reagir em escala de microssegundos quando a latência de calda é impactada por rajadas momentâneas de requisições para

um mesmo serviço. Para esses cenários, o escalonador do Caladan se apresenta como solução eficiente. E é nesse cenário que o Danian otimiza o Caladan para reduzir ainda mais a latência das aplicações em rede.

4. Caladan

Em uma máquina rodando o sistema Caladan, um núcleo é dedicado a fazer, exclusivamente, a tarefa de *scheduler*. Esse núcleo continuamente lê sinais de controle em uma frequência de microssegundos de modo a reagir a problemas antes mesmo que a qualidade do serviço possa degradar-se.

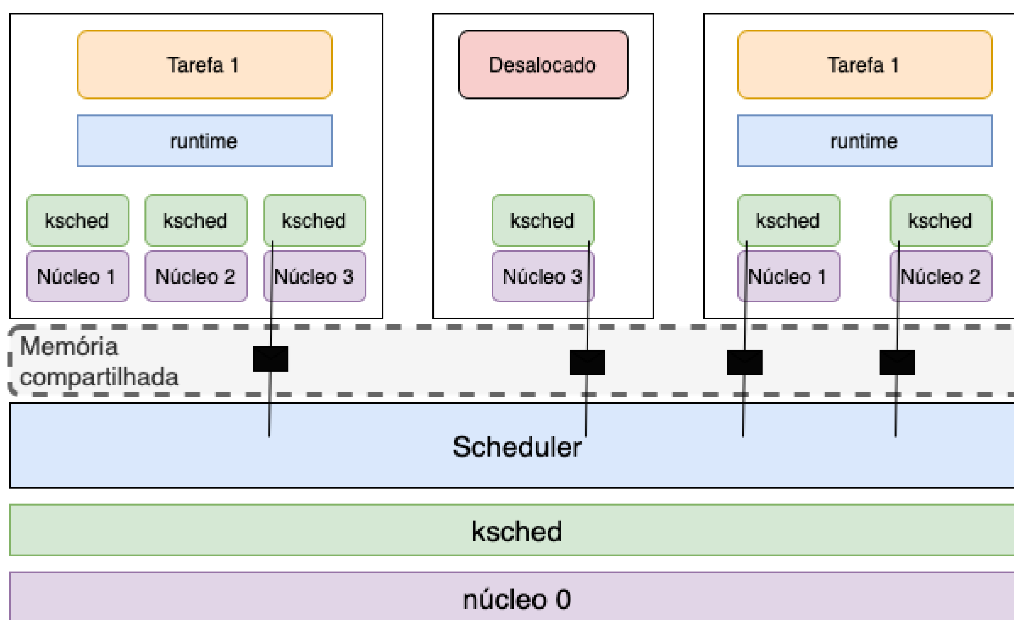

Figura 1. Imagem que apresenta a arquitetura geral do scheduler centralizado

A Figura 1 mostra a arquitetura do Caladan. O núcleo centralizado e chamado de *Scheduler* é responsável por ler sinais de controle dos *runtimes* das *threads* e das regiões de memória compartilhada. Assim, o *Scheduler* consegue ajustar a alocação de núcleos para *threads* que precisam ser escalonadas.

O Caladan descreve um módulo para o *kernel* do Linux chamado *KSCHED*. Em uma escala de microssegundos, esse módulo consegue fazer o agendamento de núcleos para *threads*. Uma instância desse módulo está associada a cada núcleo da máquina disponível para o Caladan. O agendamento é assíncrono, pois assim o agendador pode executar outras tarefas enquanto aguarda um núcleo completar uma tarefa.

O Caladan utiliza uma estrutura de dados de lista encadeada para guardar quais as *threads* estão em estado *idle* (disponíveis) dentro do escopo de um processo:

```
struct proc {
  pid_t pid;
  ...
  struct list_head idle_threads;
  ...
}
```

Listing 1. Lista duplamente encadeada original do Caladan

Nessa abordagem, toda vez que o escalonador precisa escolher uma *thread* ele itera sobre essa lista duplamente encadeada em busca da *thread* disponível e que tenha sido executada recentemente no núcleo a ser utilizado. Essa solução executa em $O(n)$, onde n é o número de *threads* executadas para um dado processo. Quando nenhuma *thread* em *idle* está disponível para ser escolhida, o algoritmo escolhe a última *thread* nessa lista duplamente encadeada.

O Código em linguagem C abaixo descreve a função *sched_pick_kthread* utilizada pelo Caladan para fazer a escolha da *thread* para um dado núcleo e processo passados como parâmetros dessa função.

```
static struct thread *
sched_pick_kthread(struct proc *p, unsigned int core)
{
  struct thread *th;

  /* first try to find a thread that last ran on this core */
  list_for_each(&p->idle_threads, th, idle_link) {
    if (th->core == core) {
      printf("From:Parent, Core:%d, Thread:%d\n", core, th->tid);
      return th;
    }
  }

  /* then try to find a thread that last ran on this core's sibling */
  list_for_each(&p->idle_threads, th, idle_link) {
    if (th->core == sched_siblings[core]) {
      printf("From:Siblings, Core:%d, Thread:%d\n", core, th->tid);
      return th;
    }
  }

  /* finally pick the least recently used thread (to avoid thrashing)
   */
  return list_tail(&p->idle_threads, struct thread, idle_link);
}
```

Listing 2. Função do Caladan para escolher uma thread para um núcleo

A escolha de uma *thread* em *idle* de um processo para um dado núcleo é o objeto de pesquisa do presente trabalho. A seção 5 descreve os novos algoritmos para otimizar a escolha de *threads* para um dado núcleo.

5. Danian

Para melhorar o desempenho desse algoritmo, o presente trabalho propôs o Danian: uma solução que adiciona um *array* de tamanho N dentro da estrutura de dados de um processo. N é o número de núcleos disponíveis na máquina em questão. Cada posição desse *array* representa um núcleo.

O Código 3 mostra a estrutura de dados adicionada ao *struct proc*. Esse *struct* representa um processo Caladan. Cada processo então passa a ter um *array* de *memoization* em que cada elemento do *array* é um ponteiro para uma *thread*, na forma de um *struct thread **, do processo em questão.

```
struct proc {
  pid_t pid;
  ...
  struct thread *last_run[NCPU];
  ...
}
```

Listing 3. Estrutura de dados de memoization por processo

Todas as posições do *array* são inicializadas com o valor $NULL$. Quando uma *thread* é escalonada para ser executada em um núcleo, atualiza-se a posição correspondente àquele núcleo para apontar para a *thread* em execução. Por exemplo, se a *thread* 5 de um dado processo é colocada para ser executada no núcleo 8, atualiza-se a posição 8 do *array last_run* com o ponteiro da *thread* escalonada. Dessa forma, sempre atualiza-se no *array* de *memoization* qual a *thread* daquele processo rodou recentemente naquele núcleo. O índice do *array* é o núcleo e o conteúdo é a lembrança (*memoization*) da *thread* em questão.

O código 4 mostra a implementação da função que escolhe uma *thread* baseada na busca *lookup* dentro do *array* de *memoization* de um dado processo. Essa operação de *lookup* executa em $O(1)$ em seu caso médio assim como em seu pior caso. Considere-se o pior caso quando a *thread* não está disponível para ser escolhida, pois ainda está ativa em algum núcleo. Nesse caso, retorna-se a última *thread* adicionada à lista duplamente encadeada utilizada pelo Caladan originalmente conforme citado anteriormente na seção 4. Buscar a calda dessa lista encadeada é uma operação também feita em $O(1)$.

```
static struct thread *
sched_pick_last_kthread(struct proc *p, unsigned int core)
{
  struct thread *th;

  th = p->last_run[core];

  if (!th->active) {
    return th;
  }

  return list_tail(&p->idle_threads, struct thread, idle_link);
}
```

Listing 4. Código que escolhe a thread para o núcleo usando memoization

Com isso, esse *array* de *memoization* por Processo em execução permite ao escalonador encontrar as *threads* disponíveis de um processo que melhor se enquadram na política de escalonamento. A saber, a *thread* daquele processo que recentemente foi executada naquele núcleo. Quando a *thread* não puder ser escalonada por não estar disponível, a solução proposta pega a calda da lista encadeada de modo a evitar desperdício (*trashing*) por parte do escalonador e **sempre** retornar uma *thread* para o escalonador.

Esse algoritmo implementa uma política LRU (*last recent used*) em que escolhe-se a *thread* que mais recentemente foi executada em um dado núcleo. Essa estratégia visa reduzir a troca de contexto entre *threads* sendo carregadas em um núcleo.

6. Experimentos

Para avaliar a solução proposta o presente trabalho executou dois experimentos: com o serviço *synthetic* e com o serviço *netperf*, ambos disponíveis no código fonte do Caladan implementados utilizando o *iokernel* (escalonador) mostrado na figura 1 como *KSCHED*. Esses programas funcionam na forma de um par de um cliente e um servidor. Ambas partes dos programas dependem do ambiente Caladan.

Os experimentos foram executados no CloudLab [Duplyakin et al. 2019] em uma topologia com duas máquinas conectadas por um *switch*: uma atuando como servidor e outra como cliente. A figura 2 exemplifica a topologia do experimento.

Figura 2. Topologia do experimento no CloudLab

Foram medidos percentis de latência entre o cliente e o servidor assim como tempo de execução da função que faz a escolha das *threads*. A latência de calda e o percentual de CPU foram computados durante os experimentos. Para o experimento *netperf* variou-se o número de *threads* utilizadas pelo cliente de modo a verificar o impacto dessa variação na latência global de uma aplicação distribuída em redes.

As máquinas *servidor 0* e *servidor 1* são computadores com 16 núcleos e interfaces de redes *Gigabit*, 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

7. Resultados

Na análise comparativa entre o escalonador do Caladan nativo e o escalonador com *memoization* de *thread* por CPU, tem-se que para todas as operações dinâmicas da estrutura de dados a complexidade de tempo de execução de $O(1)$. Já para o Caladan nativo, somente a operação de inserção executa em $O(1)$. As demais operações são em $O(n)$, onde n é o número de núcleos (CPUs) disponíveis na máquina. A tabela 2 apresenta a comparação entre as implementações:

Como ambas estruturas de dados dependem diretamente do número de núcleos disponíveis na máquina, elas tem complexidade de espaço em $O(n)$. Em função disso pode-se afirmar que, para otimizar o tempo de execução do algoritmo, foi necessário utilizar mais espaço em memória principal. A saber, cada processo armazena um novo

Programa	Complexidade de Tempo					Complexidade de Espaço
	Inicialização	**Inserção**	**Remoção**	**Atualização**	**Busca**	
Caladan	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Danian	O(n)	O(1)	O(1)	O(1)	O(1)	O(n)

Tabela 2. Tabela comparativa de complexidades de tempo e espaço das implementações do Caladan e do Caladan otimizado com memoization

array de tamanho n, onde n representa o número de núcleos disponíveis para o Caladan. Nesse caso, se tivermos 30 processos, nas máquinas do experimento (com 16 núcleos), teremos um acréscimo de alocação de $30*n*sizeof(struct\ thread*)$, onde 30 representa o número de processos, n o número de núcleos e $sizeof(structthread*)$ representa o espaço necessário pelo código em C para alocar um ponteiro de uma estrutura de dados *thread*.

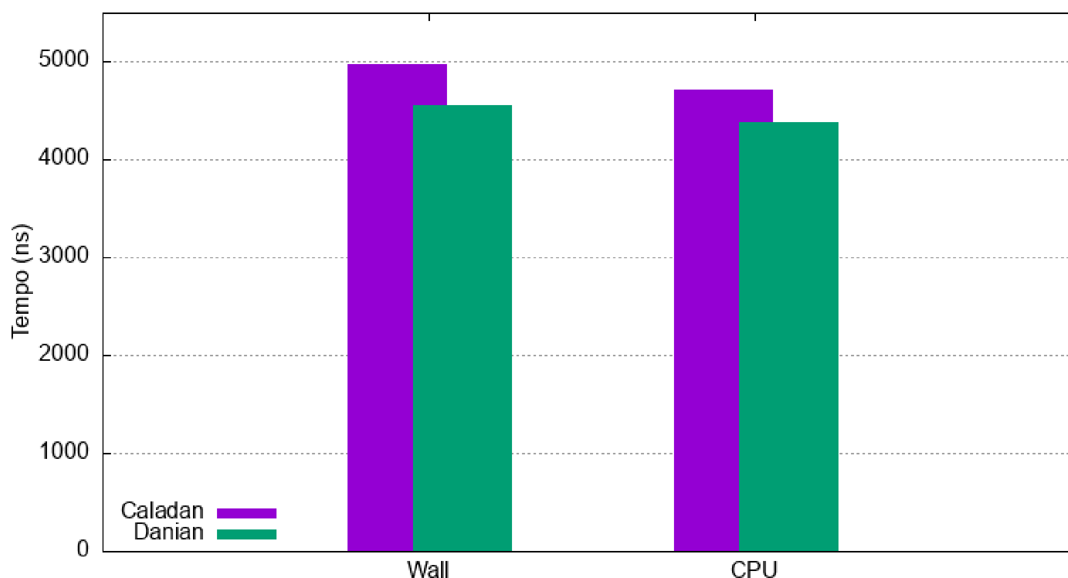

Figura 3. Tempos de CPU e Wall contabilizados para o Caladan e Danian

Notou-se, para o escopo do experimento com o programa *synthetic* uma redução no tempo de execução com a função *sched_pick_last_kthread* comparada à *sched_pick_kthread*. Foram contabilizados os tempos de CPU e *Wall*. Verificou-se uma redução de aproximadamente 7% no tempo de execução. A figura 3 mostra o gráficos dos tempos contabilizados em cada versão do escalonador.

Durante o experimento avaliou-se comportamento da latência de calda do serviço *synthetic* entre o cliente e o servidor. A figura 4 apresenta o comparativo das latências medidas em ambos os programas para o mesmo experimento. À cada número de coleta o programa *synthetic* aumenta linearmente o volume de dados transferidos. A implementação proposta mostrou uma redução de 3% no percentil 99 da latência durante a execução do

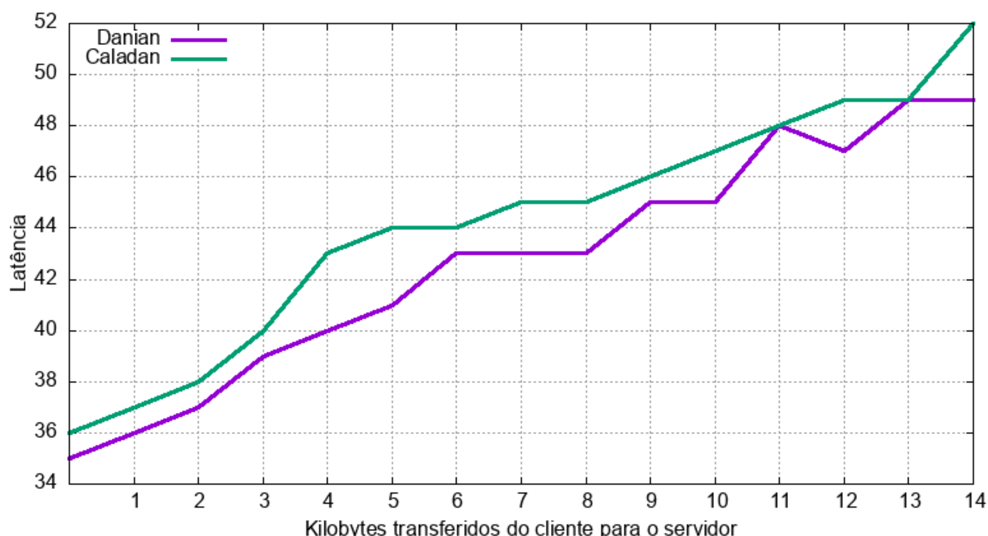

Figura 4. Latência de calda no percentil 99 do experimento Synthetic

synthetic.

Para o experimento *netperf* foram executadas diversas baterias de experimentação onde variou-se o número de *threads* na máquina cliente de 2 *threads* até 12 *threads*, aumentando de duas em duas *threads*.

A figura 5 mostra como à medida que o número de *threads* aumenta, a latência do experimento reduz. Esse comportamento ocorreu em ambos os casos: Caladan *baseline* e com a otimização de *memoization*. No entanto, o último manteve uma latência média menor em todos os cenários do experimento.

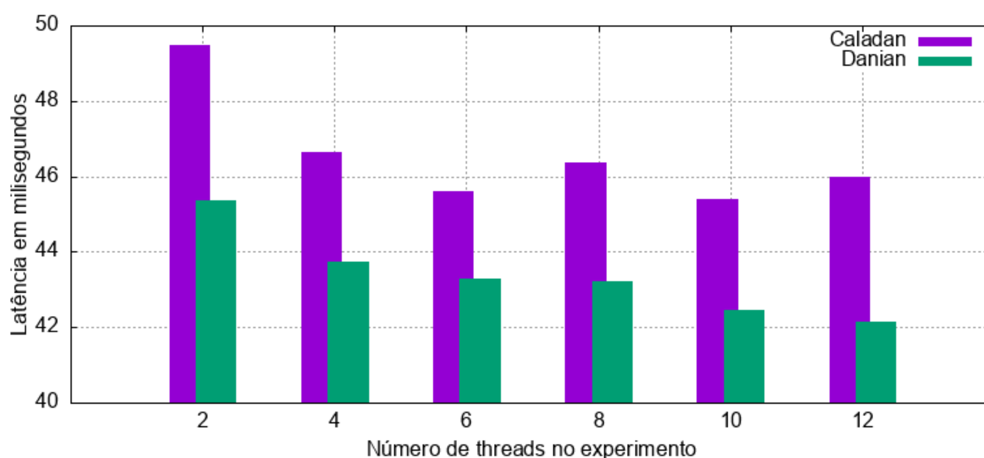

Figura 5. Latência média em função do número de threads no cliente

Durante o mesmo experimento variando o número de *threads* analisou-se o per-

centual de uso de CPU. A figura 6 mostra a dispersão dos usos de CPU computados para cada volume de pacotes enviados do cliente para o servidor.

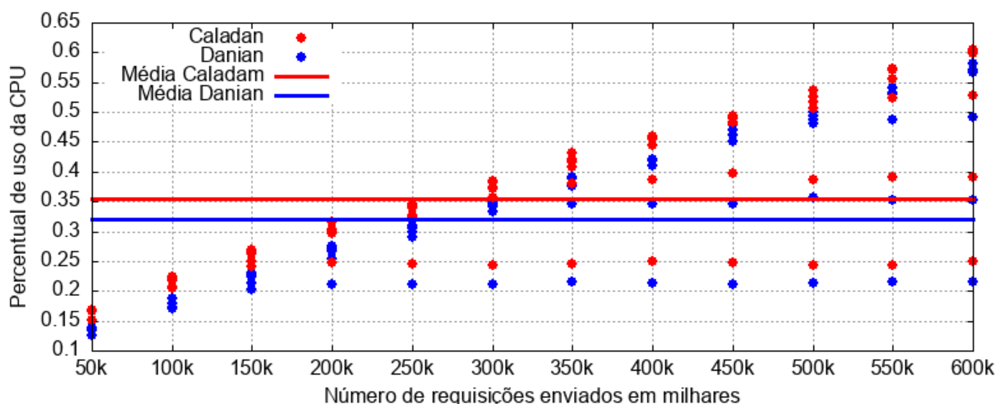

Figura 6. Médias de percentuais de CPU utilizada durante o experimento

Como pode ser notado nas linhas horizontais desse gráfico mostrando as médias globais dos percentuais de CPU, o algoritmo de otimizado com *memoization* reduz em quase 5% o uso de CPU pelo programa *netperf*.

A latência de calda do experimento *netperf* foi medida no percentil 90 comparando as latências entre o Caladan *baseline* e o otimizado com *memoization*. A figura 7 mostra esse resultado e é possível observar a redução na latência à medida que o número de pacotes enviados por coleta do experimento era executado.

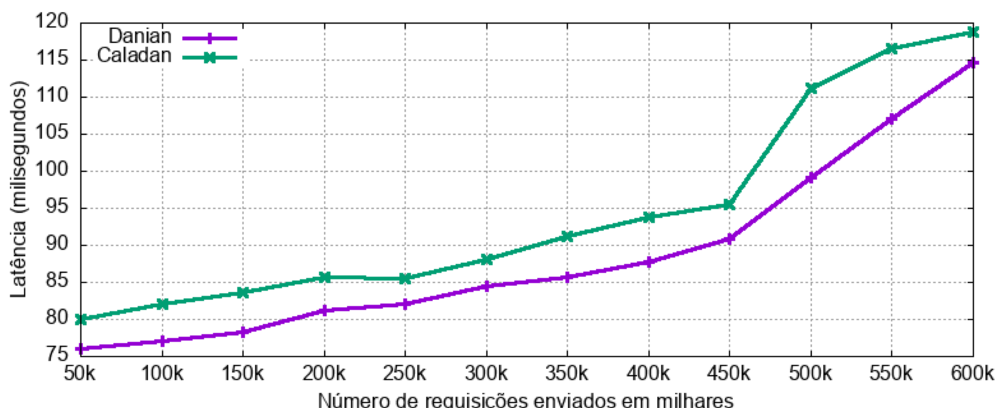

Figura 7. Latência de calda medida no experimento netperf

Através desses experimento foi possível comparar e medir o desempenho da função apresentada no código 4 ao qual a função *sched_pick_last_khread()* faz uso do *array* de *memoization* por processo para lembrar e buscar qual foi a última *thread* executada para um dado núcleo em uma política de LRU.

Durante o experimento Netperf, observou-se uma melhoria na vazão de pacotes à medida que o número de *threads* foram aumentadas. Uma melhoria média de 3% pode ser verificada na figura 8 comparando a solução Danian ao Caladan. Note que as curvas de vazão de requisições convergem ao final do experimento (12 *threads*) para 600 mil. Esse é o limite de I/O do sistema experimentado.

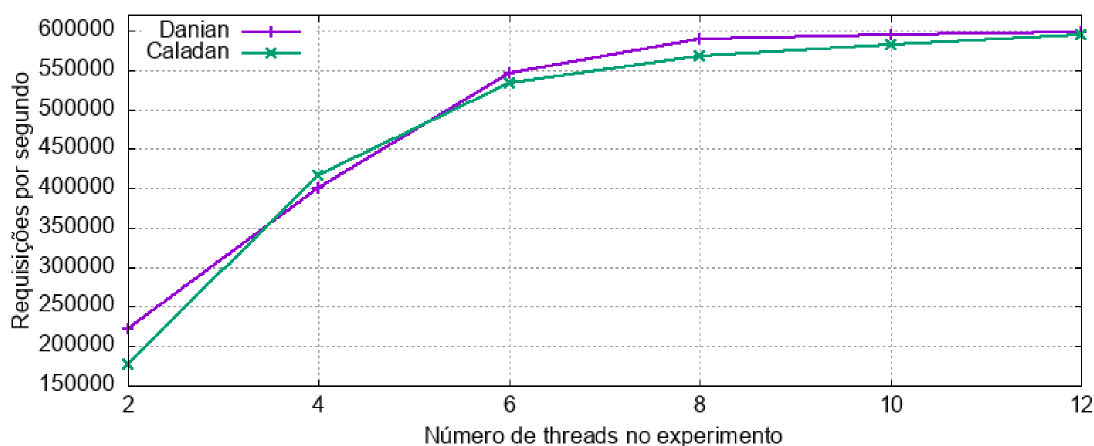

Figura 8. Requisições por segundo em funções do número de threads

8. Trabalhos Futuros

Em um cenário com múltiplas tarefas sendo acomodadas pelo escalonador Caladan, a latência e utilização dos ciclos de CPU pode se comportar de maneira diferente em relação ao experimento apresentado neste trabalho. Tendo por exemplo uma tarefa que processa pacotes provenientes de um banco de dados *memcached*, e outra tarefa com um *garbage collector* sendo executada em *background*, o escalonador Caladan pode realizar ainda mais operações de escolha de *thread*.

Desta maneira, a presente implementação proposta com *memoization* no mecanismo de escolha entre *threads* pode se mostrar ainda mais eficiente em cenários onde há interferência entre tarefas, se apresentando assim como interessante oportunidade de análise futura.

O Caladan tem potencial de ser utilizado para implementar uma solução em SDN, em que pode-se separar o plano de dados e o plano de controle [Pantuza et al. 2014]. Em especial para um plano de dados eficiente e de baixa latência.

O Caladan pode ser utilizado como escalonador para aplicações utilizando *lthreads* [Rushing's 2021]. Seria possível criar funções de rede (NFVs) eficientes e fazer experimentos comparativos verificando o desempenho dessas funções.

Uma avaliação detalhada do algoritmo em diversos outros cenários e programas seriam interessantes para estressar os limites da solução proposta, que, utilizando uma estrutura de dados com complexidade de tempo constante *O(1)* em *lookup* otimiza a utilização de CPU e reduzir latência no processamento de pacotes.

9. Conclusão

O presente trabalho melhora o desempenho do escalonador de CPU Caladan na forma como ele escolhe *threads* a serem alocadas em um dado núcleo do computador. A solução proposta reduz de $O(n)$ para $O(1)$ a complexidade de tempo de execução do algoritmo que escolhe uma *thread* de um processo para ser executada em um núcleo. Todas as demais operações dinâmicas utilizadas na estrutura de dados proposta também foram reduzidas a $O(1)$. Em contrapartida, aumentou-se o espaço de memória alocado por processo dentro do escalonador.

Através do uso de *memoization* com a política LRU, o Danian reduziu em 7% o tempo de CPU e 3% a latência de calda medida através do percentil 99 do experimento *Synthetic* enquanto manteve a capacidade do escalonador de alocar novos núcleos de processamento em microssegundos.

Além disso, para um experimento de desempenho da rede (*netperf*) reduziu-se a latência média em 5% e o percentual de uso de CPU em 15%. Uma análise do comportamento da latência mostra que à medida que o número de *threads* aumenta, a solução proposta reduz a latência global de *round trip* (RTT) da aplicação.

O Shenango e o Caladan são o estado da arte em eficiência na maximização do uso de CPU em um computador rodando aplicações *multi-thread*. O presente trabalho é uma contribuição para tornar essas ferramentas mais eficientes e capazes de processar pacotes de aplicações em rede que fazem uso intensivo de dados e exigem cada vez mais baixa latência.

Referências

Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. (1991). Scheduler activations: Effective kernel support for the user-level management of parallelism. *SIGOPS Oper. Syst. Rev.*, 25(5):95–109.

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., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. (2012). Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA. USENIX Association.

Bellman, R. (1954). The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503 – 515.

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.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Dean, J. and Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, 56:74–80.

DPDK (2021 (acessado em Março 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA. USENIX Association.

Fried, J., Ruan, Z., Ousterhout, A., and Belay, A. (2020). Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association.

Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. (2018). The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA. Association for Computing Machinery.

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., 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.

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., 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.

Waldspurger, C. A. and Weihl, W. E. (1994). Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, page 1–es, USA. USENIX Association.

Yang, X., Eggert, L., Ott, J., Uhlig, S., Sun, Z., and Antichi, G. (2020). Making quic quicker with nic offload. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ '20, page 21–27, New York, NY, USA. Association for Computing Machinery.