AFP: Um Escalonador de Requisições de Microsserviços Guiado por Feedback

Mayco S. Berghetti[1] **Fabrício B. Carvalho**[1,2] **Ronaldo A. Ferreira**[1]

[1]FACOM – Universidade Federal de Mato Grosso do Sul (UFMS)
[2]FAENG – Universidade Federal de Mato Grosso (UFMT)

{mayco.berghetti,fabricio.carvalho,ronaldo.ferreira}@ufms.br

Abstract. *Applications that serve millions of users over the Internet, such as social networks and online games, are typically decomposed into microservices to scale and provide low response times. Many of these applications process requests with high service-time dispersion, which leads to the Head-Of-Line Blocking problem in which requests with high service times block other requests and increase the average and tail latencies of the applications' requests. This work presents AFP (Application Feedback Policy), a scheduling policy that mitigates this problem. Experimental results show that AFP processes up to 4.5× more workload than the best-known approach for SLOs of a few microseconds.*

Resumo. *Aplicações que atendem milhões de usuários na Internet, como redes sociais e jogos online, são tipicamente decompostas em microsserviços para escalar e prover baixos tempos de resposta. Muitas dessas aplicações processam requisições com alta dispersão de tempo de serviço, o que leva ao problema de Head-Of-Line Blocking em que requisições com altos tempos de serviço bloqueiam outras requisições e aumentam as latências médias e de cauda das requisições. Este trabalho apresenta AFP (Application Feedback Policy), uma política de escalonamento que mitiga esse problema. Resultados experimentais mostram que AFP processa 4,5× mais carga de trabalho que a melhor solução conhecida para SLOs da ordem de alguns microssegundos.*

1. Introdução

Avanços tecnológicos na Internet, nas redes de acesso e nos dispositivos pessoais permitiram o desenvolvimento de aplicações interativas, como redes sociais, geolocalização e jogos online, que realizam grande parte de seus processamentos em centros de dados remotos. Essas aplicações devem possuir baixos tempos de resposta para garantir uma boa experiência de milhões de usuários simultâneos [Ardelean et al. 2018].

Para atender à demanda de milhões de requisições por segundo com baixo tempo de resposta, de maneira escalável e tolerante a falhas, as aplicações são decompostas em microsserviços que são executados em vários servidores físicos com múltiplos núcleos de processamento [Zhang et al. 2022]. Na arquitetura de microsserviços, uma única requisição de usuário desencadeia inúmeras requisições menores que são processadas separadamente pelos microsserviços [Nishtala et al. 2013]. A aplicação gera a resposta da requisição original para o usuário agregando as respostas individuais dos microsserviços. Consequentemente, o tempo para completar o processamento de uma requisição é determinado pelo microsserviço mais lento a responder [Dean and Barroso 2013].

O tempo de resposta de uma requisição pode ser afetado por vários fatores durante a execução dos microsserviços, como competição por núcleos de CPU, largura de banda de acesso à memória RAM em um servidor e variação na carga de trabalho que pode induzir rajadas de requisições em pequenos intervalos de tempo [Fried et al. 2020, Dean and Barroso 2013, Ardelean et al. 2018, Schöne et al. 2015]. Além disso, aplicações que utilizam microsserviços, como Redis [Redis Ltd. 2023], possuem alta dispersão de tempo de serviço, ou seja, algumas requisições são processadas rapidamente em alguns microssegundos (*i.e.,* requisições curtas) enquanto outras podem levar vários milissegundos (*i.e.,* requisições longas). Essa disparidade nos tempos de processamento pode resultar no problema de *Head-Of-Line Blocking (HOL Blocking)* [Dean and Barroso 2013] em que requisições curtas são atrasadas pelo processamento de requisições longas, resultando em altos atrasos médios e de cauda (*i.e.,* atraso do percentil 99,9) e, consequentemente, baixa qualidade de serviço.

Com o objetivo de mitigar o problema de *HOL Blocking*, trabalhos recentes propõem diversas estratégias para o escalonamento de requisições, tais como: reserva de recursos intra servidor [Ousterhout et al. 2015, Didona and Zwaenepoel 2019, Demoulin et al. 2021] ou de servidores inteiros para processamento de requisições curtas [Delgado et al. 2016] ou interrupção de requisições longas para atender novas requisições curtas [Kaffes et al. 2019]. Infelizmente, essas estratégias não escalam, exigindo que os servidores operem com baixa utilização (*i.e.,* abaixo de 40%) para garantirem determinados objetivos de nível de serviço (SLO – *Service Level Objective*). Perséphone [Demoulin et al. 2021], por exemplo, reserva um núcleo de processamento somente para classificar as requisições e alguns núcleos de processamento para processarem exclusivamente requisições curtas para garantir o SLO de atraso de cauda dessas requisições (*e.g.,* atraso do percentil 99,9 inferior a 10 μs).

Este trabalho propõe uma nova política de escalonamento, denominada de AFP (*Application Feedback Policy*), que utiliza *feedback* da aplicação para determinar se uma requisição é de curta ou longa duração. Em AFP, requisições curtas são executadas do início ao fim sem interrupções, ao passo que requisições longas podem ser interrompidas para evitar que introduzam atrasos significativos nas demais requisições e mitigar o problema de *HOL Blocking*. Resultados de *microbenchmarks* em ambientes reais mostram baixa sobrecarga nos mecanismos utilizados por AFP para interromper e retomar o processamento de requisições longas. Além disso, resultados de simulação mostram que AFP processa 4,5$\times$ mais carga de trabalho do que a melhor solução conhecida [Demoulin et al. 2021] para SLOs da ordem de alguns microssegundos.

2. Conceitos Básicos e Motivação

Nos últimos anos, enlaces de rede têm atingido taxas de transmissão da ordem de centenas de gigabits por segundo. Por outro lado, a velocidade dos processadores está estagnada há vários anos, o que resulta na necessidade de se utilizar múltiplos núcleos de processamento (*i.e., CPU cores*) para a execução de uma aplicação de rede. Projetar software de rede para utilizar múltiplos núcleos[1] é desafiador, pois envolve o uso de dados compartilhados e a necessidade de se evitar condições de corrida.

Sistemas em produção [Nishtala et al. 2013, Foundation 2024] e trabalhos

[1] Para sermos concisos, utilizaremos a palavra núcleo como sinônimo de núcleo de processamento.

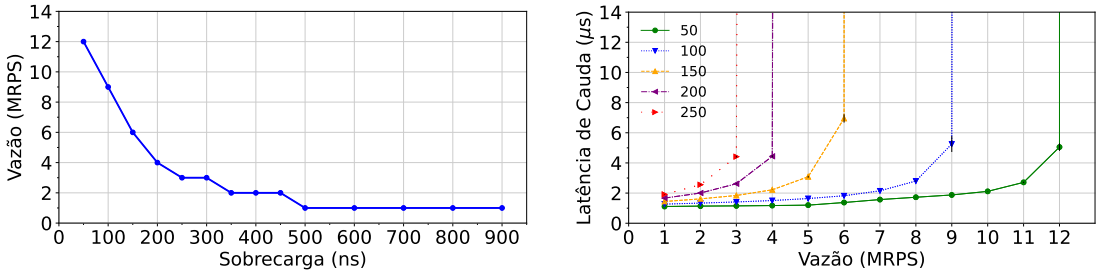
acadêmicos [Carvalho et al. 2022] utilizam a técnica de *sharding* para escalar aplicações de rede. Neste caso, o estado do software de rede é particionado e cada partição é atribuída a um único núcleo, evitando assim acessos concorrentes (estratégia conhecida como *share nothing* [Prekas et al. 2017]). O adaptador de rede (NIC – *Network Interface Card*) é programado para enfileirar os pacotes em núcleos específicos utilizando uma função de hash que garante que pacotes de um fluxo são processados pelo mesmo núcleo. Esse mecanismo de direcionamento de pacotes é conhecido como RSS (*Receive-Side Scaling*).

O uso de múltiplas filas com uma política de processamento *First Come First Served* (FCFS) resulta em uma política de escalonamento conhecida como *d-FCFS* (*Decentralized First Come First Served*). Muito embora *d-FCFS* permita o desenvolvimento de software escalável, ela não resolve o problema de *HOL Blocking* por pelo menos dois motivos. Primeiro, RSS pode não balancear perfeitamente as requisições entre os núcleos [Barbette et al. 2019] e muitas requisições ficarem concentradas em poucos núcleos enquanto os demais ficam ociosos, tornando o sistema sem conservação de trabalho (*non-work-conserving*) e aumentando os atrasos médio e de cauda das requisições. Segundo, redistribuir as requisições para outros núcleos para balancear a carga do sistema introduz acessos concorrente a estados compartilhados e exige o uso de locks para evitar condições de corrida, impactando no desempenho do sistema.

Para diminuir os atrasos médio e de cauda de requisições com alta dispersão de tempo de serviço, alguns trabalhos usam a política de escalonamento *c-FCFS (Centralized First Come First Served)*, em que todas as requisições são enfileiradas em uma única fila compartilhada por todos os núcleos [Kaffes et al. 2019, Demoulin et al. 2021]. Dessa forma, os núcleos processam as requisições da fila compartilhada a medida que ficam ociosos. A política *c-FCFS* mitiga o problema de *HOL Blocking* para algumas cargas de trabalho porque balanceia melhor as requisições entre os núcleos do que RSS. Porém, o recebimento de rajadas de requisições longas pode ocupar todos os núcleos e fazer com que as demais requisições tenham que esperar, incorrendo em *HOL Blocking*.

Esses trabalhos reservam um núcleo *despachante* para encaminhar as requisições para núcleos *trabalhadores* (*worker cores*) que efetivamente processam as requisições. O despachante, além de despachar as requisições para os trabalhadores, realiza outras tarefas visando reduzir a latência de cauda das requisições. Shinjuku [Kaffes et al. 2019], por exemplo, define um *quantum* de tempo que limita o tempo máximo que um trabalhador pode processar uma requisição. Caso o quantum expire, o trabalhador é interrompido pelo despachante para que outra requisição seja processada. Assim, Shinjuku evita que outras requisições sejam atrasadas por um tempo maior que um quantum. Entretanto, o despachante não pode despachar requisições durante o tempo de envio de uma interrupção a outro núcleo, impactando na escalabilidade do sistema. Perséphone [Demoulin et al. 2021], por sua vez, utiliza o despachante como um classificador de requisições. O despachante classifica as requisições e enfileira as curtas em uma fila distinta das longas. Além disso, Perséphone reserva núcleos para processar exclusivamente requisições curtas, minimizando assim o impacto das requisições longas nos tempos de processamento das requisições curtas.

Perséphone assume uma sobrecarga no despachante de 100 ns para classificar uma requisição ao passo que Shinjuku considera uma sobrecarga no despachante de 126 ns para interromper um trabalhador. Entretanto, essas sobrecargas podem variar



(a) Vazão em função da sobrecarga. **(b)** Latência em função da vazão.

Figura 1. Sobrecarga adicionada por despacho centralizado.

significativamente dependendo do trabalho de classificação ou de interferências no sistema durante o processamento de interrupções. Redis, por exemplo, gasta em média 660 ns para analisar e classificar uma requisição [Peter et al. 2015]. Com isso, a sobrecarga no despachante impacta diretamente na escalabilidade do sistema e na latência de cauda das requisições como ilustra a Figura 1. A Figura 1a mostra que a vazão do sistema em milhões de requisição por segundo (MRPS) diminui significativamente quando a sobrecarga no despachante aumenta. Similarmente, a Figura 1b mostra o impacto na latência de cauda (percentil 99,9) em função da carga de entrada para diferentes sobrecargas no despachante. Os experimentos das figuras simulam um sistema semelhante a Perséphone e Shinjuku com uma política *c-FCFS*, um núcleo despachante, 14 núcleos trabalhadores, tempo de serviço das requisições constante e igual a 1 μs, tempo de chegadas das requisições seguindo uma distribuição exponencial e um SLO de tempo máximo para a latência de cauda de 99,9% de 100$\times$ o tempo de serviço. É possível observar que poucos nanossegundos de sobrecarga no despachante impactam significativamente na vazão do sistema e na latência de cauda das requisições.

3. Política de Escalonamento AFP

AFP explora conhecimento do domínio do problema pelas aplicações e requer uma leve interação com elas para evitar retrabalho de classificação de requisições e decidir como estas serão escalonadas. A principal premissa de AFP é que as aplicações possuem conhecimento suficiente para determinar se uma requisição é curta ou longa e podem auxiliar o escalonador em seu processo de decisão. Uma aplicação como Redis, por exemplo, primeiro identifica o tipo da requisição (*e.g.,* leitura, escrita ou varredura) antes de executá-la. Com isso, a aplicação sabe de antemão se a requisição exigirá pouco (leitura) ou muito (escrita ou varredura) tempo de processamento e pode informar ao escalonador o tipo de requisição que irá processar.

Diferente de Perséphone, que dedica um núcleo despachante para classificar as requisições e distribuí-las para os núcleos trabalhadores, em AFP os núcleos trabalhadores recebem as requisições diretamente das filas do NIC via RSS, dispensando assim o uso de um núcleo despachante, que pode ser o gargalo do sistema dependendo da carga de trabalho, e evitando o retrabalho de classificar as requisições que já é feito pela aplicação, o que adicionaria atrasos no processamento das requisições. Entretanto, AFP reserva um núcleo para operar como temporizador e interromper requisições longas que atinjam um limite de tempo de processamento e evitar que elas atrasem o processamento das demais requisições. A aplicação executa as requisições curtas sem interrupção utilizando o modelo *run-to-completion*, ou seja, a aplicação notifica AFP somente quando irá executar uma requisição longa. AFP escalona as requisições longas nos núcleos trabalhadores em

duas etapas: *(i)* por meio de suas filas no NIC e *(ii)* por meio de uma fila de espera compartilhada entre os trabalhadores.

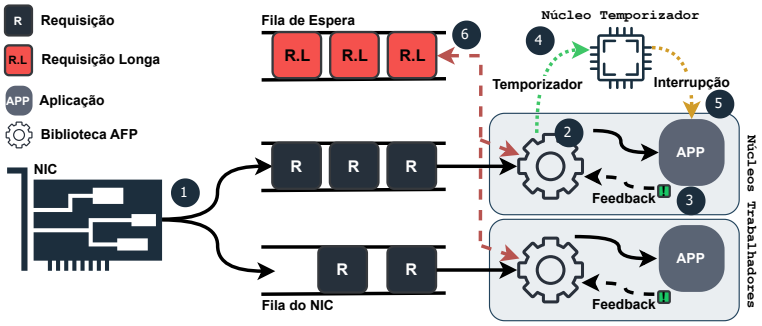


Figura 2. Visão geral de AFP.

A Figura 2 apresenta os principais componentes de AFP e os passos para o escalonamento das requisições. Em ❶, o NIC recebe as requisições e as distribui para as múltiplas filas utilizando RSS. Em ❷, cada núcleo trabalhador retira uma requisição de sua fila no NIC e executa a aplicação para fazer o processamento inicial e determinar se é uma requisição curta ou longa. Em ❸, a aplicação envia um *feedback* para AFP somente se a requisição atual for longa. Caso contrário, ela processa a requisição até o fim sem interrupções. Em ❹, o escalonador, ao receber o *feedback*, notifica o núcleo temporizador para atualizar o temporizador daquele núcleo trabalhador com um novo *quantum*. Em ❺, quando o temporizador expira, o núcleo temporizador determina se o trabalhador deve continuar o processamento da requisição atual ou ser interrompido para processar outra requisição. O núcleo temporizador envia uma interrupção ao trabalhador somente quando há novas requisições na fila do núcleo trabalhador no NIC. Nesse caso, o trabalhador interrompe o processamento da requisição atual e enfileira o contexto de execução na fila de espera compartilhada. Por fim, em ❻, quando não há requisições em sua fila, o trabalhador processa requisições que estão na fila de espera caso elas existam. As seções a seguir detalham os diferentes componentes do sistema.

3.1. Núcleo Trabalhador

O Algoritmo 1 mostra os passos de um núcleo trabalhador durante as etapas de escalonamento de requisições e processamento da aplicação. De forma contínua, os trabalhadores processam requisições que chegam pelo NIC ou retomam o processamento que está na fila ($\mathcal{Q}^{\text{wait}}$) compartilhada (laço *while* nas Linhas 3 a 11). Cada trabalhador i busca uma nova requisição em sua fila no NIC ($\mathcal{Q}_i^{\text{NIC}}$) caso o contador compartilhado `shared_count` for menor que o valor predefinido *THRESH* (Linha 4). Esse contador é atomicamente incrementado pelos trabalhadores, garantindo que a fila compartilhada seja visitada periodicamente e, consequentemente, evitando *starvation* das requisições longas.

Cada trabalhador i utiliza a variável compartilhada T_i para sincronizar suas ações com o núcleo temporizador e informá-lo quando deve ser interrompido. A cada requisição em $\mathcal{Q}_i^{\text{NIC}}$, o trabalhador i invoca a função `process_application` (Linha 6). Caso não haja requisições em $\mathcal{Q}_i^{\text{NIC}}$, o trabalhador busca e retoma a execução de uma requisição interrompida anteriormente em $\mathcal{Q}^{\text{wait}}$, se houver. Nesse caso, o trabalhador i chama a função `afp_feedback_start` (Linha 10) para que o núcleo temporizador insira uma nova entrada $T_i + quantum$ no temporizador. Note que o ponto de retorno quando a aplicação termina é na Linha 6, pois a função `resume` (Linha 11) nunca retorna.

Algoritmo 1 - *Worker Core* (w_i)

```
1:  shared_count ← ATOMIC_INIT_ONCE(0);
2:  T_i ← 0;
3:  while TRUE do
4:      if ATOMIC_READ(shared_count) < THRESH and request ← Q_i^NIC.try_pop() then
5:          ATOMIC_INC(shared_count);
6:          process_application(request);
7:      else
8:          ATOMIC_SET(shared_count, 0);
9:          if context ← Q^wait.try_pop() then
10:             afp_feedback_start()
11:             resume(context);                    ▷ restaura o contexto de execução da aplicação
12: function AFP_FEEDBACK_START()
13:     long_finished_i ← FALSE;
14:     ATOMIC_SET(T_i, now())
15: function AFP_FEEDBACK_STOP()
16:     ATOMIC_SET(long_finished_i, TRUE);
17:     ACQUIRE_SPIN_LOCK(L_i);
18:     T_i ← 0;
19:     RELEASE_SPIN_LOCK(Li);
20: function PROCESS_APPLICATION(r)
21:     application logic...
22:     if is_long_request(r) then                  ▷ parser da aplicação
23:         afp_feedback_start();
24:         application logic...
25:         afp_feedback_stop();
26:     else
27:         application logic...
28: function INTERRUPT_HANDLER()
29:     if long_finished_i = TRUE then
30:         return;
31:     Q^wait.push(current_context);
32:     yield_to_main();                            ▷ retorna ao laço principal na Linha 3
```

A aplicação utiliza as funções `afp_feedback_start` (Linha 12) e `afp_feedback_stop` (Linha 15) para enviar *feedback* para AFP no início e no fim do processamento de uma requisição longa. Essas funções atualizam T_i, para habilitar e desabilitar o temporizador, e `long_finished`$_i$, para evitar preempção no trabalhador i durante a desativação do temporizador. O spinlock L_i para atualizar T_i na função `afp_feedback_stop` é necessário para garantir que o trabalhador não receba uma interrupção após desativar o temporizador. Sem o spinlock, tal situação pode ocorrer devido ao tempo de propagação da interrupção no sistema. O uso do spinlock ocorre apenas no final do processamento de requisições longas, minimizando a sobrecarga no sistema, pois a maioria das cargas de trabalho consiste em requisições curtas.

A função `process_application` (Linha 20) categoriza a requisição r como curta ou longa na Linha 22. Para requisições longas, ela envia *feedbacks* para AFP indicando o início e o fim do processamento nas Linhas 23 e 25, respectivamente.

Por fim, a função `interrupt_handler` (Linha 28) define o fluxo de execução do trabalhador ao receber uma interrupção. O trabalhador verifica se a requisição longa atual está finalizada (Linha 29) para retornar ao ponto de retorno da aplicação (*i.e.,* Linha 6). Caso contrário, o trabalhador enfileira o contexto de execução corrente em $\mathcal{Q}^{\text{wait}}$ (Linha 31) e retorna ao início do laço principal (Linha 32). Quando o contexto é restaurado, o processamento continua a partir do ponto que foi interrompido.

3.2. Núcleo Temporizador

AFP dedica um núcleo para atuar como temporizador, o que permite a implementação eficiente de interrupções de outros núcleos utilizando *quantuns* de curta duração sem impactar significativamente na vazão do sistema e nos tempos de processamento das requisições. O núcleo temporizador possui três funções principais: *(i)* gerenciar o temporizador, *(ii)* avaliar quando um trabalhador deve ser interrompido e *(iii)* enviar interrupções aos trabalhadores.

Processadores atuais possuem um registrador *timestamp counter* (TSC) de alta precisão que incrementa a uma taxa constante e fornece uma opção eficiente para a implementação de temporizadores. O núcleo temporizador utiliza o TSC para determinar quando um quantum expirou e interromper um trabalhador. O Algoritmo 2 sumariza os passos executados pelo núcleo temporizador. Para cada trabalhador i, o núcleo temporizador tenta obter o spinlock L_i compartilhado com i (Linhas 3 e 10) e verificar se o trabalhador i está processando uma requisição longa (*i.e.,* $T_i \neq 0$) por mais de um *quantum* (Linha 4). Nesse caso, o núcleo temporizador verifica se há requisições em $\mathcal{Q}_i^{\text{NIC}}$ do trabalhador i para atualizar T_i (Linha 6) e interromper o processamento do trabalhador i (Linha 7). Caso não haja requisições em $\mathcal{Q}_i^{\text{NIC}}$, o núcleo temporizador atualiza T_i para que o trabalhador i continue o processamento da requisição atual por mais um *quantum* (Linha 9). Este passo visa minimizar o número de interrupções geradas e a sobrecarga associada para interrupção de um trabalhador.

Algoritmo 2 - *Timer core*

```
1:  while TRUE do
2:      for each worker core i do
3:          if TRY_ACQUIRE_SPIN_LOCK(L_i) then
4:              if T_i ≠ 0 and (now() - T_i > quantum) then
5:                  if Q_i^NIC.len() > 0 then
6:                      T_i ← 0;
7:                      send_interrupt(i);          ▷ interrompe o processamento do trabalhador i
8:                  else
9:                      T_i ← now();
10:             RELEASE_SPIN_LOCK(L_i);
```

3.3. Temporizador

Como microsserviços processam requisições na escala de microssegundos ou até mesmo nanosegundos, eles são paralelizados no contexto de um único processo, o que facilita o acesso a variáveis compartilhadas (*e.g.,* T_i nos Algoritmos 1 e 2) e a implementação de temporizadores eficientes como o descrito na Seção 3.2. Entretanto, temporizadores podem ser implementados em ambientes Unix com `alarm`, `setitimer` ou `timer_create`, mas essas funções invocam chamadas de sistema que possuem sobrecargas significativas e impactam negativamente o desempenho do sistema. Resultados de microbenchmark calculando o intervalo de tempo entre habilitar e desabilitar os temporizadores em um servidor com processador Intel(R) E5-2660 @2.60GHz e Kernel 4.4.185 mostram que `alarm`, `setitimer` e `timer_create` apresentam uma sobrecarga média de 639±0,12, 607±0,08 e 529±0,15 nanossegundos, respectivamente, enquanto AFP possui sobrecarga de apenas 40±0,15 nanossegundos (*i.e.,* 13× mais eficiente que as demais). Esses resultados representam a média de 500 mil execuções com intervalo de confiança de 95%.

3.4. Fila de Espera

Quando os trabalhadores não possuem requisições em suas filas no NIC, eles processam requisições longas interrompidas da fila compartilhada. Assim, AFP não deixa os processadores ociosos. Entretanto, em situações de alta carga, as requisições na fila compartilhada poderiam não ser processadas. Para evitar *starvation* dessas requisições, AFP limita em *THRESH* (Linha 4 do Algoritmo 1) a quantidade de requisições que são escalonadas sem que um trabalhador processe uma requisição da fila de espera. Os trabalhadores inserem e retiram requisições na fila compartilhada utilizando operações atômicas (*e.g., compare-and-swap*) e sem a necessidade de uso de locks.

3.5. Interrupção

AFP utiliza IPIs (*Inter Processor Interrupts*) para interromper os trabalhadores que estão processando requisições longas. Para minimizar as sobrecargas referentes às interrupções, AFP utiliza a função `smp_call_function_single` do Kernel do Linux para exportar a funcionalidade de IPI para espaço de usuário por meio de um novo módulo de Kernel denominado `AFP_kmod`. Muito embora existam diversos métodos para implementar IPI, cada um apresenta diferentes características e sobrecargas. A Tabela 1 apresenta os resultados de um microbenchmark em um servidor com processador Intel(R) E5-2660 @2.60GHz e Kernel 4.4.185 para as sobrecargas de envio, recebimento e retorno do tratador de interrupções. Os resultados mostram a média de 500 mil execuções de cada componente com intervalo de confiança de 95%.

Tabela 1. Sobrecarga de diferentes métodos de IPI.

IPI	Envio (ns)	Recebimento (ns)	Retorno (ns)
`signal`	$601 \pm 0{,}10$	$996 \pm 0{,}30$	$620 \pm 0{,}09$
`AFP_kmod`	$375 \pm 0{,}21$	$770 \pm 0{,}31$	$25 \pm 0{,}04$
Dune [Belay et al. 2012]	$851 \pm 0{,}29$	$482 \pm 0{,}37$	$97 \pm 0{,}11$

Por ter uma implementação genérica, `signal` (POSIX) apresenta uma sobrecarga significativa devido às chamadas de sistema, especialmente, a `rt_sigreturn` para restaurar o contexto da aplicação. Por outro lado, Dune [Belay et al. 2012] utiliza virtualização APIC (*Advanced Programmable Interrupt Controller*) [Intel 2023] para que processos em espaço de usuário acessem diretamente os recursos de hardware, removendo as trocas de modo de operação entre Kernel e usuário para o envio e recebimento de IPIs. Além disso, Dune implementa *posted interrupt*, uma tecnologia que permite que a entrega de interrupções seja mais eficiente em máquinas virtuais, apresentando o melhor resultado no recebimento de interrupções.

`AFP_kmod` diminui a sobrecarga de envio de interrupções em 1,6× em comparação com `signal` porque o temporizador envia uma interrupção diretamente para um trabalhador enquanto `signal` é genérico e, portanto, possui sobrecargas extras (*e.g.,* checar se o processo que irá receber o sinal está executando). Além disso, `AFP_kmod` salva e restaura o contexto da aplicação totalmente em espaço de usuário, reduzindo em mais de 2× a sobrecarga total (1.616 ns vs. 795 ns) no tratamento de interrupções (Recebimento + Retorno) em relação a `signal`. Muito embora `AFP_kmod` não apresente o menor tempo para recebimento de sinais/interrupções, o módulo de AFP permite o uso de novas tecnologias de hardware, como *User Interrupts* disponíveis em processadores Intel *Sapphire Rapids*, permitindo extensões para entregas de interrupções ainda mais rápidas diretamente no espaço de usuário [Sohil Mehta 2023].

4. Avaliação

Esta seção avalia AFP por meio de simulação estendendo o simulador de eventos discretos fornecido em [McClure et al. 2022] para incorporar as políticas de AFP, Perséphone (PSP) e d-FCFS usando RSS. Perséphone representa um dos trabalhos mais recentes e de melhor desempenho para o problema de *HOL Blocking* e é similar aos trabalhos anteriores, como Shinjuku. A implementação de RSS com múltiplas filas representa vários sistemas que usam *sharding*, mas que não lidam especificamente com o problema de *HOL Blocking*. Todos os resultados apresentados nesta seção mostram a média de 10 execuções com um intervalo de confiança de 95%. As simulações utilizam parâmetros extraídos dos trabalhos avaliados e dos microbenchmarks descritos na Seção 3.

A Tabela 2 apresenta os parâmetros que o simulador utiliza durante as avaliações. Para AFP e PSP, o simulador utiliza 14 núcleos trabalhadores, enquanto para RSS, o simulador utiliza 15 trabalhadores, uma vez que RSS não utiliza núcleo temporizador como AFP nem núcleo despachante como PSP. Para avaliar diferentes comportamentos referentes à chegada das requisições, o simulador utiliza as distribuições Exponencial e Lognormal, que são as distribuições mais observadas em centro de dados [Benson et al. 2010]. No que tange às aplicações, os experimentos consideram duas cargas de trabalho bimodal: Carga de Trabalho 1 e 2. A Carga de Trabalho 1 é composta por 99,5% das requisições com tempo de serviço de 0,5 μs, denominadas requisições curtas, e 0,5% das requisições com tempo de serviço de 500 μs, denominadas requisições longas. A Carga de Trabalho 2 é composta por 99% das requisições com tempo de serviço de 1 μs e 1% das requisições com tempo de serviço de 100 μs. Para AFP, o simulador considera parâmetros de *THRESH* de 1000, *quantum* de 1, 2, 4 e 8 μs e a sobrecarga de interrupção de 580 μs. Para PSP, o simulador considera a sobrecarga do despachante de 150, 300, 450 e 600 ns.

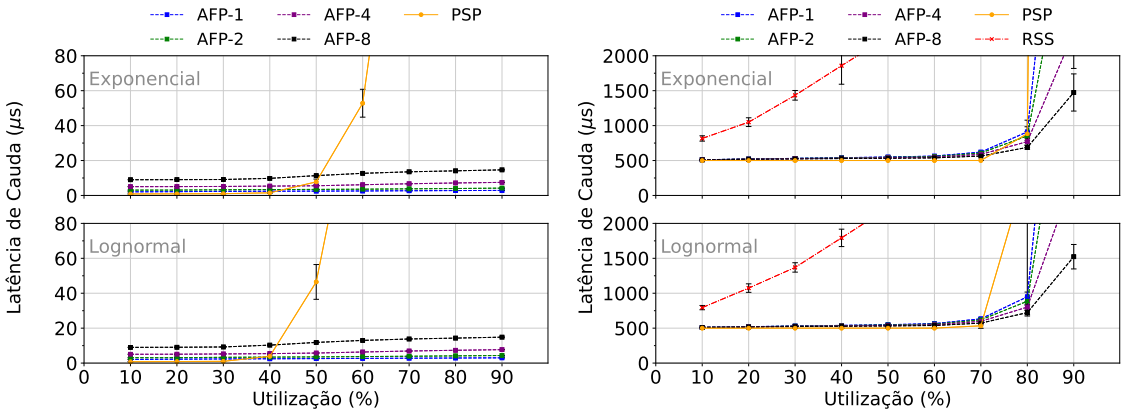
Tabela 2. Parâmetros de simulação.

Política	Parâmetro	Valor
RSS	Quantidade de núcleos trabalhadores	15
AFP e PSP		14
PSP	Quantidade de núcleos trabalhadores reservados	2
AFP, PSP e RSS	Distribuição de chegada das requisições	Exponencial e Lognormal
	Carga de Trabalho 1	99,5% (0,5 μs) e 0,5% (500 μs)
	Carga de Trabalho 2	99,0% (1 μs) e 1,0% (100 μs)
AFP	*THRESH*	1000
AFP	*Quantum*	1, 2, 4 e 8 μs
AFP	Sobrecarga da interrupção	580 ns
PSP	Sobrecarga do núcleo despachante	150, 300, 450 e 600 ns

Durante a simulação, para AFP e RSS, cada núcleo trabalhador recebe as requisições por meio de sua respectiva fila, simulando a distribuição aleatória das requisições pelo NIC. No caso de PSP, o NIC entrega todas as requisições ao núcleo despachante. Em PSP e RSS, os trabalhadores recebem e processam as requisições no modelo *run-to-completion*. No caso de PSP, o despachante inicialmente classifica as requisições e determina qual núcleo trabalhador receberá a requisição. Em AFP, os trabalhadores processam as requisições curtas no modelo *run-to-completion* e processam as requisições longas em etapas com duração de um *quantum*.

4.1. Latência de Cauda

Esta seção avalia a latência de cauda (*i.e.,* percentil 99,9) do tempo de processamento das requisições das políticas AFP, PSP e RSS sob as Cargas de Trabalho 1 e 2 em diferentes cenários. Os experimentos variam a carga oferecida em função da utilização dos núcleos trabalhadores. O simulador calcula a utilização em função da quantidade de trabalhadores e a distribuição da carga de trabalho. Por exemplo, para 14 trabalhadores, 99,5% das requisições de 500 ns, 0,5% das requisições de 500 μs e utilização de 50% resulta em $\approx$3,5 MRPS (Milhões de Requisições por Segundo).



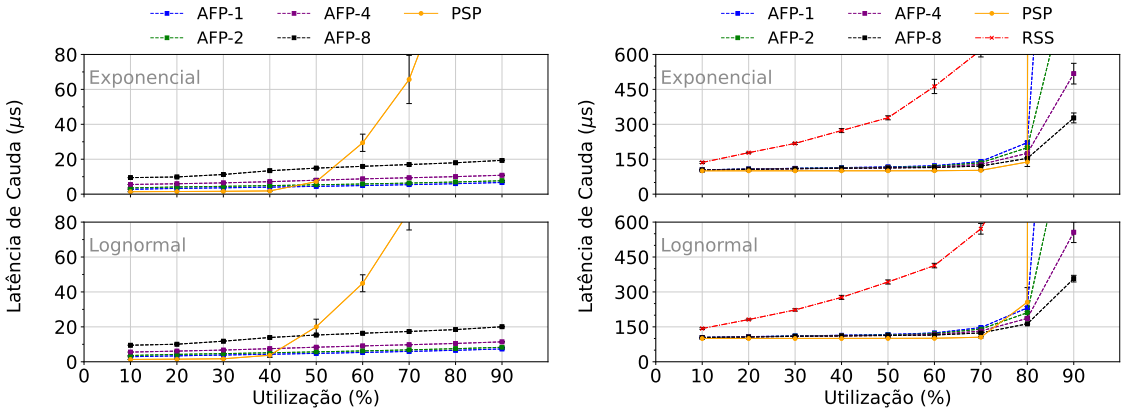
(a) Requisições curtas (0,5 μs) **(b)** Todas as requisições

Figura 3. Latência do percentil 99,9 para a Carga de Trabalho 1.

A Figura 3 apresenta os resultados de latência de cauda para as políticas AFP, PSP e RSS utilizando a Carga de Trabalho 1 e variando a utilização dos núcleos trabalhadores. O simulador varia a utilização de 10% a 90%. Nos sistemas atuais, servidores operam em taxas bem inferiores de utilização, normalmente na ordem de 30%, para poderem garantir SLOs de latência de cauda. Para avaliação de AFP, o valor do *quantum* foi variado em 1, 2, 4 e 8 μs, representados pelas curvas AFP-1, AFP-2, AFP-4 e AFP-8, respectivamente. Além disso, os gráficos superiores apresentam os resultados em que o intervalo de tempo entre as requisições segue uma distribuição exponencial enquanto os gráficos inferiores uma distribuição lognormal.

A Figura 3a mostra as latências de cauda das requisições curtas da Carga de Trabalho 1. AFP escala para todas as taxas de utilização sem degradar significativamente o desempenho, mantendo as latências de cauda praticamente estáveis. Por outro lado, PSP degrada a latência de processamento das requisições a partir de 50% de utilização para distribuição exponencial e 40% para lognormal. A política RSS apresenta uma latência de cauda média de 753 μs para as requisições curtas mesmo com utilização em 10%. Por isso, para a melhor visualização dos resultados, os gráficos não apresentam a curva de latências para requisições curtas do RSS. A Figura 3b apresenta as latências do percentil 99,9 para Carga de Trabalho 1 de todas as requisições (*i.e.,* curtas e longas). AFP é competitivo em relação a PSP até a taxa de 70% de utilização, em ambas as distribuições de tempo de chegada. Contudo, a partir de 80% de utilização, AFP apresenta as menores latências de cauda, especialmente para AFP-4 e AFP-8.

Similar ao experimento anterior, a Figura 4a apresenta os resultados de latência



(a) Requisições curtas (1 μs) **(b)** Todas as requisições

Figura 4. Latência do percentil 99,9 para a Carga de Trabalho 2.

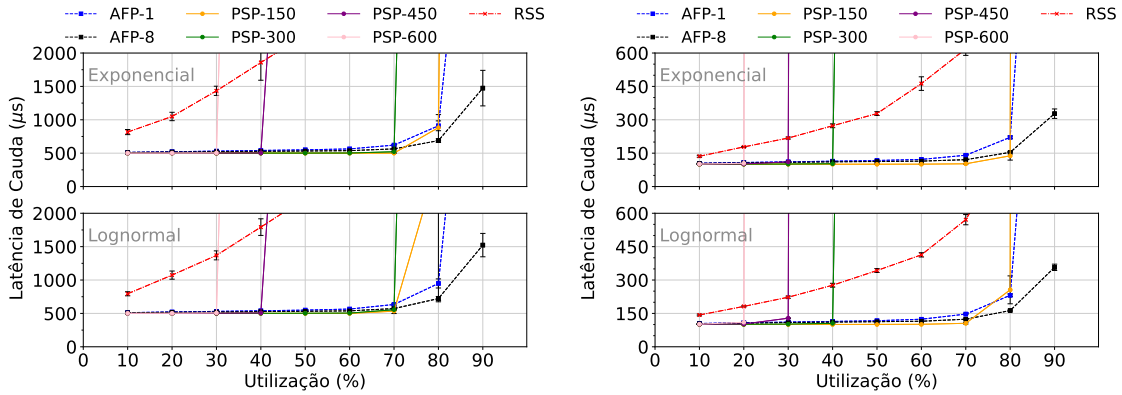
de cauda para a Carga de Trabalho 2. AFP e PSP são competitivos até a taxa de 40% de utilização. A partir desse ponto, PSP começa a degradar a latência das requisições curtas, enquanto AFP mantém as latências praticamente estáveis em todas as taxas de utilização avaliadas. Para a política RSS, a latência de cauda é de 115 μs a uma taxa de utilização de 10%. Por isso, para a melhor visualização dos resultados, o gráfico não apresenta a curva de latência para requisições curtas do RSS. A Figura 4b apresenta as latências de cauda para todas as requisições da Carga de Trabalho 2 e mostra que AFP é competitivo com PSP até a carga de 70% da utilização. Contudo, a partir de 80%, AFP apresenta menores latências de cauda no caso de distribuição lognormal em todos os valores de *quantum*. Entretanto, devido à utilização da fila centralizada, PSP degrada de forma severa as latências das requisições devido à utilização de um único núcleo despachante a partir de 80% de carga. Por outro lado, em ambas as distribuições, AFP-8 mantém baixas latências das requisições em todas as taxas de utilização (*e.g.,* garante SLO inferior a 400 μs).

Uma política de escalonamento padrão com RSS apresenta as piores latências de cauda, como mostrado nas Figuras 3 e 4, pela ausência de técnicas de balanceamento de carga ou classificação das requisições. Neste caso, as requisições curtas são atrasadas pelo processamento de requisições longas, resultando em altos atrasos de cauda. PSP, apesar de balancear a carga com a utilização da fila centralizada, sofre com a falta de escalabilidade imposta pelo núcleo despachante. Por outro lado, AFP além de escalar conforme a quantidade de núcleos trabalhadores, maximiza a utilização da capacidade de processamento disponível ao mesmo tempo que atenua o problema de *HOL Blocking*, sem impactar de forma significativa a latência das requisições curtas mesmo processando uma carga de trabalho 2$\times$ maior que PSP para SLOs da ordem de alguns microssegundos.

4.2. Escalabilidade

Os experimentos da Seção 4.1 consideram uma sobrecarga fixa de 150 ns do núcleo despachante de PSP, como reportado em [Demoulin et al. 2021]. Entretanto, o despachante possui atribuições além da classificação das requisições, tais como receber os pacotes de dados do NIC e enfileirar as requisições no núcleo trabalhador. Além disso, dependendo da aplicação, a classificação de requisições pode ser complexa, acarretando sobrecargas na ordem de $\approx$ 600 ns [Peter et al. 2015], por exemplo. Para avaliar AFP e PSP utilizando diferentes sobrecargas no núcleo despachante, esta seção apresenta

resultados de experimentos variando a sobrecarga do despachante de PSP e o *quantum* de AFP. A Figura 5 apresenta os resultados de latência de cauda desses experimentos utilizando os parâmetros definidos na seção anterior. As curvas PSP-150, PSP-300, PSP-450 e PSP-600 representam o núcleo despachante com sobrecargas de 150, 300, 450 e 600 ns, respectivamente, e as curvas AFP-1 e AFP-8 representam *quantuns* de 1 e 8 μs de AFP, respectivamente.



(a) Carga de Trabalho 1 **(b)** Carga de Trabalho 2

Figura 5. Latência de cauda para diferentes sobrecargas no despachante PSP.

A Figura 5a mostra as latências do percentil 99,9 de todas as requisições utilizando a Carga de Trabalho 1. Similar aos experimentos das Figuras 3 e 4, a política RSS apresenta os piores resultados devido à ausência de técnicas de balanceamento e classificação de requisições. Por outro lado, as diferentes sobrecargas no núcleo despachante faz com que PSP apresente diferentes comportamentos. Para PSP-150 e PSP-300, um único núcleo despachante não é o gargalo, fazendo com que PSP escale até 80% de utilização no melhor caso. Por outro lado, para PSP-450 e PSP-600, o despachante torna-se o gargalo, fazendo com que as latências sejam piores do que na política RSS a partir de 50% de utilização. Por sua vez, AFP apresenta as melhores latências de cauda, sendo escalável para as taxas de utilização avaliadas e um SLO de 2000 μs, tanto para AFP-1 quanto para AFP-8 e em ambas distribuições.

Similar à Figura 5a, a Figura 5b mostra os resultados utilizando a Carga de Trabalho 2. Nessa carga de trabalho em que a dispersão entre os tempos de processamento das requisições longas e curtas é menor, PSP-150 apresenta um comportamento competitivo em relação a AFP, escalando até 80%. Contudo, para sobrecargas maiores, PSP não escala a partir de 40% e em ambas distribuições. Por outro lado, AFP escala até 80% utilizando *quantum* de 1 μs e 90% utilizando *quantum* de 8 μs. Com um quanto *quantum* maior, AFP termina as requisições longas mais rapidamente, minimizando a quantidade de interrupções. Assim, AFP escala em todas as taxas de utilização para um SLO de 450 μs quando utilizando *quantum* de 8 μs.

Os resultados da Figura 5 mostram que AFP escala sem degradar a qualidade de serviço para diferentes parâmetros, cargas de trabalho e níveis de utilização.

5. Trabalhos Relacionados

***Threads* de Usuário**: Trabalhos recentes adotam *threads* de nível de usuário em que cada *thread* processa uma única requisição [Qin et al. 2018, Ousterhout et al. 2019,

Fried et al. 2020]. A criação e o gerenciamento das *threads* possuem baixo impacto no desempenho geral, fazendo com que o processamento de requisições não seja afetado significativamente. Assim como Shinjuku, AFP pode utilizar *threads* de nível de usuário para o processamento das requisições, minimizando o impacto das interrupções.

Balanceamento de Carga: Alguns trabalhos [Ousterhout et al. 2019, Prekas et al. 2017, Carvalho et al. 2022] redistribuem periodicamente a carga de processamento entre os núcleos de processamento para evitar que um núcleo fique sobrecarregado e aumente o atraso das requisições [Wierman and Zwart 2012]. Como trabalho futuro, AFP pode empregar a técnica de *work stealing* para melhorar a distribuição de requisições entre seus núcleos trabalhadores.

Interrupções: Shinjuku [Kaffes et al. 2019] e ZygOS [Prekas et al. 2017] utilizam interrupções durante o escalonamento de requisições. Shinjuku utiliza IPIs para interromper requisições longas. Porém, nenhuma requisição pode ser despachada enquanto uma interrupção está sendo enviada. ZygOS interrompe um trabalhador somente se outro trabalhador estiver ocioso e não encontrar requisições por meio de *work stealing*. Por outro lado, AFP permite que trabalhadores continuem o processamento de requisições da fila do NIC ou da fila de espera, enquanto outro trabalhador é interrompido pelo núcleo temporizador exclusivo.

6. Trabalhos Futuros e Conclusão

Este trabalho propõe AFP, uma nova política de escalonamento que se beneficia do conhecimento da aplicação para escalonar requisições e atenuar o problema de *HOL Blocking*. Resultados experimentais mostram que AFP é capaz de processar um número de requisições muito superior ao da melhor abordagem existente e ainda assim garantir SLOs na ordem de alguns microssegundos para requisições de curta duração. Como trabalho futuro, iremos integrar aplicações reais para avaliar o desempenho de AFP em ambientes e cargas de trabalho reais. Além disso, iremos explorar a instrumentação automática de aplicações a partir de técnicas de análise estática para se estimar o tempo de processamento de cada tipo de requisição, facilitando assim o uso de AFP por aplicações existentes sem a necessidade de instrumentação manual por programadores. O código-fonte do simulador e os *scripts* utilizados na avaliação estão disponíveis em `https://github.com/AFP-Project/AFP-Simulator`.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) – Código de Financiamento 001, do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) – Procs. 308101/2022-7 e 465446/2014-0, e da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) – Procs. 2023/00812-7, 2020/05183-0, 2015/24485-9 e 2014/50937-1.

Referências

Ardelean, D. et al. (2018). Performance Analysis of Cloud Applications. In *USENIX NSDI'18*.

Barbette, T. et al. (2019). RSS++ Load and State-aware Receive Side Scaling. In *ACM CoNEXT'19*.

Belay, A. et al. (2012). Dune: Safe User-level Access to Privileged CPU Features. In *USENIX OSDI'12*.

Benson, T. et al. (2010). Network Traffic Characteristics of Data Centers in the Wild. In *ACM IMC'10*.

Carvalho, F. B. et al. (2022). Dyssect: Dynamic Scaling of Stateful Network Functions. In *IEEE INFOCOM 2022*.

Dean, J. and Barroso, L. A. (2013). The Tail at Scale. *Communications of the ACM*.

Delgado, P. et al. (2016). Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *ACM SoCC'16*.

Demoulin, H. M. et al. (2021). When Idling is Ideal: Optimizing Tail-latency for Heavy-tailed Datacenter Workloads with Perséphone. In *ACM SOSP'21*.

Didona, D. and Zwaenepoel, W. (2019). Size-Aware Sharding for Improving Tail Latencies in In-Memory Key-Value Stores. In *USENIX NSDI'19*.

Foundation, L. (2024). The Linux Kernel. Disponível em `https://kernel.org/`.

Fried, J. et al. (2020). Caladan: Mitigating Interference at Microsecond Timescales. In *USENIX OSDI'20*.

Intel (2023). *Intel® 64 and IA-32 Architectures Software Developer's Volumes 3C: System Programming Guide*. Seção 26.5.5.

Kaffes, K. et al. (2019). Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *USENIX NSDI'19*.

McClure, S. et al. (2022). Efficient Scheduling Policies for Microsecond-Scale Tasks. In *USENIX NSDI'22*.

Nishtala, R. et al. (2013). Scaling Memcache at Facebook. In *USENIX NSDI'13*.

Ousterhout, A. et al. (2019). Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *USENIX NSDI'19*.

Ousterhout, J. et al. (2015). The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*.

Peter, S. et al. (2015). Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*.

Prekas, G. et al. (2017). Zygos: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *ACM SOSP'17*.

Qin, H. et al. (2018). Arachne: Core-Aware Thread Management. In *USENIX OSDI'18*.

Redis Ltd. (2023). Redis. Disponível em `https://redis.io/`.

Schöne, R. et al. (2015). Wake-up Latencies for Processor Idle States on Current x86 Processors. *Computer Science-Research and Development*.

Sohil Mehta (2023). x86 User Interrupts support. Disponível em `https://lwn.net/Articles/869140/`.

Wierman, A. and Zwart, B. (2012). Is Tail-Optimal Scheduling Possible? *Operations Research*.

Zhang, Z. et al. (2022). CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *USENIX ATC'22*.