

Universidade Federal de Campina Grande - UFCG  
Centro de Engenharia Elétrica e Informática - CEEI  
Departamento de Engenharia Elétrica - DEE

William Henrique Azevedo Martins

# **Avaliação de Desempenho para Sistemas Baseados em Container no Contexto de Indústria 4.0**

Campina Grande, Brasil

08 de Maio, 2025

William Henrique Azevedo Martins

**Avaliação de Desempenho para Sistemas Baseados em  
Container no Contexto de Indústria 4.0**

Universidade Federal de Campina Grande - UFCG  
Centro de Engenharia Elétrica e Informática - CEEI  
Departamento de Engenharia Elétrica - DEE

Orientador: George Acioli Júnior, D.Sc.

Campina Grande, Brasil

08 de Maio, 2025

William Henrique Azevedo Martins

**Avaliação de Desempenho para Sistemas Baseados em  
Container no Contexto de Indústria 4.0**

Trabalho aprovado em:        /        /

---

**George Acioli Júnior, D.Sc.**  
Orientador

---

**Eisenhauer de Moura Fernandes**  
Convidado

Campina Grande, Brasil  
08 de Maio, 2025

# Agradecimentos

Dedico este trabalho à minha mãe, Edineide, e à minha irmã, Kathleen, cujo amor e apoio foram fundamentais em minha trajetória.

Sou grato a todos que caminharam ao meu lado desde o início, nos bons e maus momentos. Em especial, agradeço a Francinildo, Alysson, Iury, Rafael e José. E aos eternos amigos Josivan, Humberto, Fabio, Maria, Gabriel, Lucas e Marconi, que, por tanto tempo, têm me suportado.

Agradeço também aos amigos do Laboratório de Instrumentação Eletrônica e Controle – Matheus Ferreira, Egydio Tadeu, Anna Aguiar, Mateus Figueiredo, Douglas Almeida, Bruno Oliveira, Raiff Santos, Samara Cardoso e Julia Ramalho — pelo companheirismo, conselhos e apoio ao longo dessa jornada.

Por fim, registro minha sincera gratidão aos professores Rafael Bezerra, Péricles Rezende, George Acioli e Thiago Euzébio, pelo conhecimento transmitido e pelas oportunidades oferecidas, que foram fundamentais para o meu crescimento acadêmico e profissional.



# Resumo

A necessidade por soluções escaláveis e flexíveis tem crescido com o advento da indústria 4.0. Nesse cenário, a virtualização por software (*containers*) vem ganhando espaço por permitir o *deployment* rápido em qualquer ambiente. Essa tecnologia já está presente em grandes *datacenters* e, mais recentemente, em dispositivos embarcados. A necessidade por explorar a containerização em soluções que exigem compromisso com o tempo de resposta levanta o questionamento: o uso de contêineres impacta a performance e segurança de qualquer aplicação ou dispositivo adotado? Para isso, foram propostos cenários que permitem avaliar o impacto da adoção dos contêineres em um sistema, simulando diversos níveis de carga de trabalho em sistemas ciberfísicos. Os testes propostos foram realizados em duas plataformas de *hardware* distintas, ambas com restrições de recursos computacionais: uma voltada para aplicações *Internet of Things* (IoT) e outra voltada para aplicações em sistemas de automação industrial. Com isso, foi possível inferir que o *overhead* introduzido no sistema ao adotar o uso de contêineres é similar ao desempenho nativo do sistema. Embora presente, esse impacto não compromete significativamente a aplicação.

**Palavras-chaves:** Virtualização; Container; Desempenho; IoT; Automação Industrial; *Raspberry*; *DCN Computer*;

# Abstract

The need for scalable and flexible solutions has grown with the advent of Industry 4.0. In this scenario, software virtualization (*containers*) has been gaining ground by enabling rapid *deployment* in any environment. This technology is already present in large *data-centers* and, more recently, in embedded devices. The need to explore containerization in solutions that require a commitment to response time raises the following question: does the use of containers impact the performance and security of any adopted application or device? To address this question, scenarios were proposed to evaluate the impact of container adoption on a system by simulating various levels of workload in cyber-physical systems. The proposed tests were carried out on two distinct *hardware* platforms, both with computational resource constraints: one aimed at *Internet of Things* (IoT) applications and the other aimed at applications in industrial automation systems. From these analyses, it was possible to infer that the *overhead* introduced by adopting containers is similar to the system's native performance. Although present, this impact does not significantly compromise the application.

**Key-words:** Virtualization; Container; Performance; IoT; Industrial Automation; *Raspberry*; *DCN Computer*;

# Lista de ilustrações

Figura 1 – Kit didático O-PAS da Fabricante Smar . . . . .	7
Figura 2 – Medição da latência da CPU utilizando o Cyclictest. . . . .	8
Figura 3 – Dados de histograma da latência da CPU utilizando o Cyclictest . . . .	8
Figura 4 – Medição de latência com período não ajustado à carga, com faixa de "sombra" não mensurável. . . . .	9
Figura 5 – Medição de latência com período ajustado à carga. . . . .	9
Figura 6 – Exemplo de uso do Hackbench. . . . .	10
Figura 7 – Consumo de recursos computacionais do Hackbench. . . . .	11
Figura 8 – Execução do comando <b>stress-ng</b> com múltiplos tipos de carga. . . . .	11
Figura 9 – Chamada do <b>stress-ng</b> para estresse de CPU nível definido. . . . .	12
Figura 10 – Comparação entre containers e VMs. . . . .	14
Figura 11 – <i>Dockerfile</i> para criação de <i>container</i> padronizado para execução dos testes. . . . .	15
Figura 12 – Primeiro conjunto de teste para definição de <i>benchmarking</i> para avaliação de latência. . . . .	17
Figura 13 – Segundo conjunto de testes para definição de <i>benchmarking</i> para avaliação de latência. . . . .	20
Figura 14 – Distribuição de latência normalizada para os casos de teste 1 (sem <i>container</i> ) e o caso de teste 2 (com <i>container</i> ). . . . .	23
Figura 15 – Distribuição de latência normalizada obtida segundo o caso de teste 3 (sem <i>container</i> ) e os casos de teste 4 e 5 (com <i>container</i> ), todos executados com carga de trabalho no sistema. . . . .	24
Figura 16 – Análise da Variação de Latência . . . . .	25
Figura 17 – Teste progressivo de estresse da CPU. . . . .	26
Figura 18 – Execução dos cenários 1 e 2 no DCN Smar. . . . .	27
Figura 19 – Execução dos cenários 3 e 5 no DCN Smar. . . . .	27
Figura 20 – Resultados do cenário A executados no DCN Smar. . . . .	28
Figura 21 – Resultados do cenário B executados no DCN Smar. . . . .	29
Figura 22 – Resultados do cenário C executados no DCN Smar. . . . .	29

# Lista de abreviaturas e siglas

CLP	Controladores Lógicos Programáveis
CPU	Unidade Central de Processamento <i>Central Processing Unit</i>
RTOS	Sistema Operacional de Tempo Real ( <i>Real Time Operating System</i> )
SO	Sistema Operacional
DCN	Nó Computacional Distribuído ( <i>Distributed Computer Node</i> )
VM	Máquina Virtual ( <i>Virtual Machine</i> )
O-PAS	<i>Open Process Automation Standard</i>
IoT	Internet das Coisas ( <i>Internet of Things</i> )

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>1</b>
<b>2</b>	<b>OBJETIVOS</b>	<b>3</b>
<b>3</b>	<b>TRABALHOS RELACIONADOS</b>	<b>4</b>
<b>4</b>	<b>MATERIAIS E MÉTODOS</b>	<b>6</b>
4.1	Medição de Latência	6
4.2	Simulação de carga	10
4.3	Container	13
4.4	Cenários de Teste	15
4.4.1	Primeiro Conjunto de Testes	16
4.4.2	Segundo Conjunto de Testes	18
<b>5</b>	<b>RESULTADOS</b>	<b>22</b>
5.1	Resultados Raspberry Pi 4	22
5.2	Resultados DCN Smar	24
<b>6</b>	<b>CONSIDERAÇÕES FINAIS</b>	<b>30</b>
	<b>REFERÊNCIAS</b>	<b>31</b>

# 1 Introdução

Com o surgimento da Indústria 4.0, há a necessidade por soluções escalonáveis e flexíveis capazes de lidar com grandes volumes de dados, adaptar-se a sistemas já existentes facilitando a integração tecnológica, e a demanda por soluções baseadas em *softwares*. Neste cenário, a virtualização ganha destaque, possibilitando uma série de soluções e habilitando recursos que agregam valor aos negócios em que são empregados, como serviços em nuvens, sistemas ciberfísicos, Internet das Coisas (IoT). As vantagens da tecnologia de virtualização e o desenvolvimento de sistemas baseados em virtualização por software tais como os contêineres, deram notoriedade para os cenários de aplicações que exigem compromisso com o tempo de resposta e robustez.

Em particular, o *container* é uma tecnologia de virtualização já consolidada na área de *IT* e tem ganhado destaque em aplicações de automação industrial, tornando-se um recurso promissor para uma nova geração de controladores industriais. Esse movimento ganhou ainda mais força com o surgimento do ***Open Process Automation Standard (O-PAS)***, um padrão aberto mantido pelo *The Open Group* que define uma arquitetura aberta, interoperável e segura para sistemas de automação de processos industriais, utilizando padrões existentes e emergentes sempre que possível — um padrão de padrões.

O uso de *container* na indústria, têm-se mostrado promissor, facilitando o uso de inteligência artificial em dispositivos de borda (BEÑO et al., 2024). Permite a virtualização escalonável e flexível de CLP, habilitando funcionalidades que vão além de apenas realizar o controle de malha fechada (MELLADO; NÚÑEZ, 2022). Além disso, viabilizam a padronização e distribuição de micros serviços voltados para a robótica, em qualquer escala de aplicação (BUSCH; REIHER; ECKSTEIN, 2024). Mesmo em aplicações mais restritas, os containers podem ser projetados para conter apenas o essencial para a execução das aplicações, possibilitando a criação de qualquer estrutura de micros serviços voltada para o setor de automação industrial, como os blocos de funções IEC 61499 (YANG; DAI, 2022).

Em comparação com as tecnologias tradicionais de virtualização, como as máquinas virtuais (VMs), o *container* é uma alternativa mais eficiente e responsivas. Enquanto uma VM inclui um sistema operacional completo com seu próprio *kernel*, os containers compartilham o *kernel* do sistema operacional do host, utilizando recursos para isolamento e gerenciamento de recursos. Por isso, os contêineres são leves, ocupam menos espaço de memória, possuem inicialização rápida, e contêm apenas as bibliotecas e dependências necessárias para executar uma aplicação (RANDAL, 2019).

Por estas razões, os contêineres foram principalmente adotados em áreas relaci-

onadas ao IT, e hoje estão difundidas em diversos cenários, como na computação em nuvem, indústria de automação, estruturação de micro-serviços, serviços distribuídos de alta disponibilidade, sistemas ciber-físicos, balanceamento de carga e entre outros. Mas sua adoção ainda é visto com certa resistência em algumas áreas que exigem maior compromisso de resposta, robustez e segurança cibernética da aplicação. No entanto, sua adoção ainda enfrenta resistência em algumas áreas que exigem maior previsibilidade na resposta, robustez e segurança (QUEIROZ et al., 2023). Além disso, outro fator limitante é a restrição de poder de processamento e memória disponível em plataformas de *hardware* com menor disponibilidade de recursos, como as utilizadas em IoT e automação.

Este trabalho apresenta um estudo sobre a demanda computacional da execução de aplicações containerizadas em um *Raspberry Pi* e em um dispositivo baseado no padrão *O-PAS*, ambos dispositivos com restrição de recursos computacionais. Avaliamos cinco testes de caso para obter métricas de desempenho, como uso de CPU e latência, sob diferentes cargas de trabalho. Como a adoção de contêineres em automação industrial e hardware de IoT tende a crescer, compreender seus requisitos computacionais é fundamental para um design de sistema eficiente e alocação adequada de recursos.

Foram realizados testes que visam coletar a latência do sistema em diferentes situações, em que pode-se avaliar o *overhead* inserido ao migrar a carga de trabalho do dispositivo *host* para um ambiente virtualizado e isolado como o *container*. Os resultados obtidos foram avaliados de forma objetiva, compreendendo o impacto dos *container* em sistemas de automação industrial.

## 2 Objetivos

### Objetivo Geral

- Propor cenários de teste para a avaliação do desempenho de *containers* em diferentes plataformas de *hardware*.
- Avaliar o impacto da adoção de *containers* em sistemas de automação industrial.

### Objetivos Específicos

- Comparar o desempenho dos testes propostos em diferentes dispositivos.
- Caracterizar as capacidades de processamento das plataformas de *hardware* avaliadas.
- Identificar como cargas alocadas em diferentes configurações (*container* ou *host*) afetam o desempenho do sistema.

## 3 Trabalhos Relacionados

(GIVEHCHI et al., 2014) explora o uso de virtualização para criar uma plataforma capaz de prover o controle da planta por meio de Controladores Lógicos Programáveis (CLPs) Virtuais disponibilizados como serviço nuvem dentro de uma rede industrial privada. Os serviços de controle proposto em nuvem apresentam desempenho comparável ao de um CLP físico acoplado no sistema, dando base para migrar a lógica de controle para o CLP virtual sem comprometer o funcionamento do sistema visando aplicações *soft real-time*, em que as restrições de tempo são mais flexíveis.

Na mesma linha de serviços de controle, (GOLDSCHMIDT et al., 2015) propõe uma arquitetura de controle em nuvem, destacando vantagens como a descentralização por meio da orquestração de serviços para aplicações *hard real-time* e a escalabilidade dos CLPs Virtuais para atender a cargas de trabalho maiores. O autor explora cenários com múltiplos sistemas conectados à rede industrial consumindo recursos do CLP Virtual, sendo uma característica ausente da análise realizada por Givehchi. O objetivo é determinar, de forma empírica, a capacidade de resposta do CLP Virtual para múltiplos dispositivos dentro da arquitetura proposta, analisando o comportamento do *round-trip time* (RTT) com base em um limiar de tempo de resposta que atenda à aplicação de controle.

O trabalho de (HEGAZY; HEFEEDA, 2014) explora amplamente o conceito de computação em nuvem, alocando a lógica de controle em servidores comerciais localizados a milhares de quilômetros de distância da planta de controle. Essa proposta aproveita a alta disponibilidade dos servidores, além de permitir a migração completa da computação para servidores auxiliares em caso de falhas. Embora essa abordagem apresente algumas desvantagens, como questões de segurança e atrasos na comunicação, o autor propõe uma solução para mitigar os efeitos desses atrasos dentro da malha de controle. Ele compara o desempenho do controle em nuvem proposto com um controle convencional, demonstrando que é possível distanciar a lógica de controle da planta sem que ocorram *overshoot* ou instabilidade devido à latência variável da comunicação. Embora tenha sido demonstrado que é possível alocar a lógica de controle em servidores distantes, é necessário modificar a malha de controle para compensar os efeitos do atraso de comunicação. Se comparado com Givehchi e Goldschmidt, a arquitetura proposta por Hegazy tende a ser mais barata de implementar e requer menos custo de manutenção, já que não necessita de infraestrutura própria. Apesar disso, o autor não explora os aspectos de segurança da informação e nem os problemas de desconexão na planta local, o que inviabilizaria a comunicação com qualquer servidor externo e conseqüentemente o controle da planta.

(GOLDSCHMIDT; HAUCK-STATTELMANN, 2016) propõe uma arquitetura de CLP baseada em *software* por meio da virtualização de componentes e emulação de sistemas legados. O objetivo é propor um *framework* que permita à indústria adotar facilmente opções mais modernas de dispositivos, possibilitando que aplicações originalmente projetadas para plataformas específicas possam ser executadas de forma semelhante por meio da emulação parcial ou completa do *hardware*. Além disso, o autor explora problemas na atual pirâmide de automação, adotando uma estratégia flexível e dinâmica voltada para os desafios do setor, em particular na configuração e reutilização de componentes. Nesse contexto, a metodologia proposta destaca os containers como ponto chave, visando à criação de componentes-base para comunicação, dados, engines, aplicações, além de aspectos de robustez e orquestração das aplicações.

Segundo (SOLLFRANK et al., 2020), apesar dos contêineres serem uma tecnologia muito mais rápida que as VMs, é usualmente explorado por aplicações de TI e de serviço *web*. O autor explora um cenário simples de comunicação cliente-servidor para sistemas de automação industrial distribuído, por meio do estudo de caso do controle de um pêndulo invertido. O trabalho avalia aspectos importantes para a virtualização em sistemas distribuídos de tempo real, como o tempo de computação, atrasos de rede, distribuição e detecção de *outliers*. O estudo realizado do controle em malha fechada evidencia a capacidade dos containers de serem aplicados em virtualização em Sistemas de Controle em Rede.

## 4 Materiais e Métodos

A viabilidade de um sistema baseado em contêineres exige uma análise de desempenho para determinar se o sistema é capaz de processar as informações dentro do tempo esperado. Pois, dispositivos industriais são projetados para ter alto grau de robustez, e em geral não apresentam uma grande quantidade de recursos tais como memória e poder de processamento, por isso o *overhead* da containerização deve ser avaliado para garantir que não haverá falhas críticas decorrentes da ausência destes recursos. Além disso, falhas que comprometam o tempo de resposta do sistema não devem ser toleradas. Para avaliar esses aspectos, foi estabelecido um conjunto de cenários que possibilitam inferir o *overhead* introduzido pelos contêineres e o comportamento do sistema sob condições de estresse.

Neste trabalho, será analisada a latência em dois dispositivos: o *Raspberry Pi 4* e o *DCN Computer* da fabricante *Smar*, um dispositivo computacional voltado para aplicações industriais que adere ao padrão *O-PAS*.

Um *Raspberry Pi 4 Model B* foi utilizado como plataforma de teste. O dispositivo é equipado com uma CPU *ARM Cortex-A72* quad-core de 64 bits, operando a 1,5 GHz, e pode funcionar em ambientes com temperaturas entre 0° C e 50° C. Este processador foi escolhido devido à sua eficiência no manejo de tarefas computacionalmente intensivas, aliada ao baixo consumo de energia, sendo amplamente utilizado em aplicações *IoT* ou em cenários que não exigem alto poder de processamento.

O *DCN Computer* da *Smar* é equipado com um processador *Intel x62000FE* de arquitetura *amd64*, que possui dois núcleos de processamento operando a 1,0 GHz e 1,5 MB de memória cache. Além disso, o dispositivo é capaz de operar em ambientes com temperaturas de até 85° Celsius. Na Figura 1, observa-se um kit educacional da fabricante *Smar*, no qual estão presentes dois *DCN Computers* anexados ao rack de cor amarela, responsáveis por executar aplicações como estratégias de controle, servidores de comunicação e serviços distribuídos. Em azul, estão os cartões de *I/O*, transmissores de temperatura, pressão e um conversor Modbus para 20-4 mA.

Tanto o *Raspberry Pi 4* quanto o *DCN Smar* possuem interface de conexão *Ethernet*, sendo este o principal recurso utilizado neste trabalho para operar os dispositivos por meio de conexão remota.

### 4.1 Medição de Latência

Para medir a latência do sistema, foi utilizada a ferramenta *Cyclictest*. Sendo empregada a construção de *benchmarkings* para avaliação de sistemas em tempo real, pois

Figura 1 – Kit didático O-PAS da Fabricante Smar



Fonte: Autor.

é capaz de medir precisamente a latência e fornecer estatísticas sobre o tempo de resposta de um sistema. O *Cyclictest* será utilizada para inferir a latência dos sistemas em tempo real diante dos cenários de testes propostos, objetivando a construção de um *benchmarking* para avaliação do *overhead* tecnologia de containers.

O *Cyclictest* mede a diferença entre o tempo de execução esperado de um grupo de tarefas periódicas de medição e o seu tempo de execução real. Além disso, apresenta grande versatilidade para a medição de latência em sistemas de tempo real, permitindo não apenas inferir a latência do sistema, mas também configurar um valor desejado de latência (limiar), notificando latências superiores ao definido pelo usuário. O *Cyclictest* também possibilita o rastreamento detalhado dos núcleos de processamento em que o limiar foi ultrapassado, sendo útil na detecção de falhas sistêmicas.

Por meio dos recursos do *Cyclictest*, é possível realizar testes como: avaliação dos piores cenários de latência, análise de desempenho de *Kernels* Linux de Tempo Real, e aproximação da latência de aplicações executadas em sistemas operacionais de tempo real (*RTOS*).

A Figura 2 contém um exemplo de como medir a latência do sistema em uma janela de tempo definida. Os argumentos passados por terminal são: `-i`, que se refere ao intervalo de medição em microssegundos (por padrão, 1000); `-l`, que define o número de repetições do teste (por padrão, 0, indicando execução contínua); e `-t`, que especifica o número de *threads* que serão atribuídas ao teste, distribuídas pelos núcleos do processador.

Na Figura 2, são exibidas a latência mínima, média e máxima resultantes ao final do teste (ou instantâneas, caso `-10`). Os dados podem ser melhor detalhados em formato de histograma, conforme pode ser observado na Figura 3. O número de *bins* é especificado

Figura 2 – Medição da latência da CPU utilizando o Cyclicttest.

```

rpi4@rpi4:~$ sudo cyclicttest -mloackall -p99 -i250 -l4000 -t4
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.01 0.04 0.01 1/389 6443

T: 0 ( 6440) P:99 I:250 C: 4000 Min: 3 Act: 4 Avg: 3 Max: 30
T: 1 ( 6441) P:99 I:750 C: 1323 Min: 3 Act: 4 Avg: 3 Max: 13
T: 2 ( 6442) P:99 I:1250 C: 789 Min: 4 Act: 4 Avg: 4 Max: 19
T: 3 ( 6443) P:99 I:1750 C: 559 Min: 4 Act: 4 Avg: 5 Max: 10

```

Fonte: Autor.

Figura 3 – Dados de histograma da latência da CPU utilizando o Cyclicttest

```

rpi4@rpi4:~$ sudo cyclicttest -mloackall -smp -p99 -i250 -l4000 -h10 -q
defaulting realtime priority to 2
# /dev/cpu_dma_latency set to 0us
# Histogram
000000 000000
000001 000000
000002 000000
000003 000000
000004 000269
000005 003365
000006 000260
000007 000046
000008 000028
000009 000020
# Total: 000003988
# Min Latencies: 000004
# Avg Latencies: 000005
# Max Latencies: 000025

```

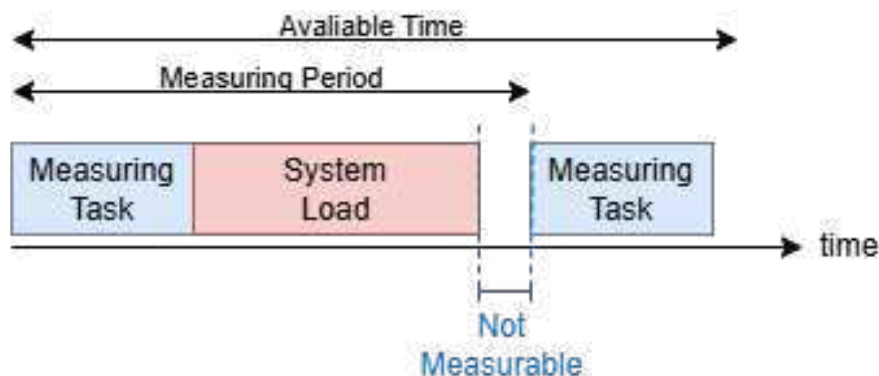
Fonte: Autor.

pelo argumento `-h` ou `-histogram`; no exemplo, é definido um total de 10 *bins* que vão de 0 a 9 microssegundos. Valores entre os *bins* são automaticamente arredondados e adicionados ao *bin* correspondente.

No entanto, um teste deve ser projetado tomando os devidos cuidados para que não haja erros de objetivo durante a medição. Tanto a configuração do teste quanto os parâmetros passados ao *Cyclicttest* devem ser escolhidos em função do cenário em que se deseja avaliar a latência. Se esses elementos não forem considerados criteriosamente, os resultados do *Cyclicttest* não representarão com precisão as latências que ocorrem naquela situação. Portanto, o *Cyclicttest* detectará uma latência se, e somente se, ela impedir que uma tarefa de medição seja executada no prazo. Este princípio deve orientar a seleção das opções do *Cyclicttest*, caso contrário, latências relevantes podem ser perdidas (Williams e Kacur (2025)).

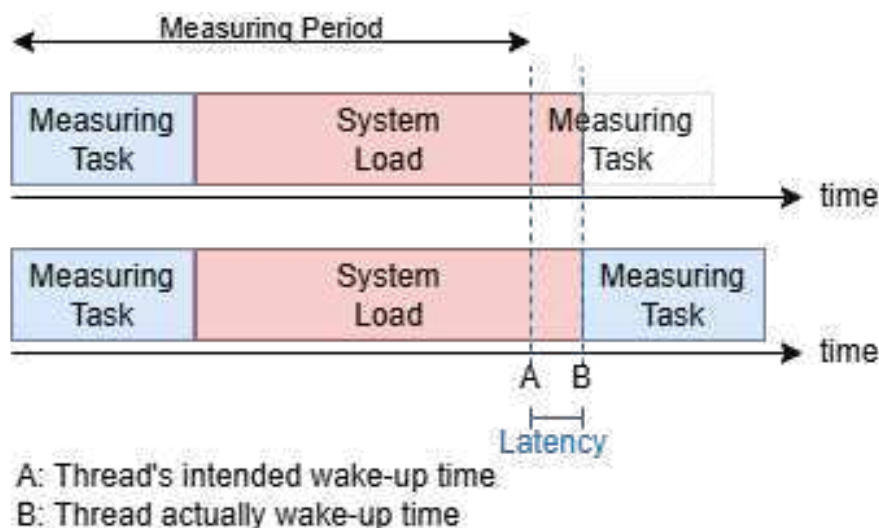
Por exemplo, se o período de execução da tarefa de medição for muito longo e as latências sempre ocorrerem e forem resolvidas entre os momentos em que a tarefa de

Figura 4 – Medição de latência com período não ajustado à carga, com faixa de "sombra" não mensurável.



Fonte: Autor.

Figura 5 – Medição de latência com período ajustado à carga.



Fonte: Autor.

medição deve ser executada, essas latências nunca serão percebidas pelo *Cyclictest*, pois nunca atrasam a execução da tarefa de medição. A Figura 4 contém um esquema em que o intervalo entre as threads de medição é maior em relação ao *slot* de tempo que a carga consome no processador, criando uma região em que o *Cyclictest* não consegue perceber incrementos ou decrementos da carga. Já a Figura 5 apresenta um esquema similar, no qual o período de amostragem é reduzido (ou a carga é aumentada), permitindo agora a medição da latência no sistema gerado pela carga.

Note que, na prática, o *overhead* do teste é muito inferior ao da carga do sistema e pode ser desprezado. As proporções nas Figuras 2 e 3 são apenas ilustrativas, visando uma melhor visualização.

Figura 6 – Exemplo de uso do Hackbench.

```
rpi4@rpi4:~$ hackbench -s 512 -l 200 -g 15 -f 25 -P
Running in process mode with 15 groups using 50 file descriptors each (== 750 tasks)
Each sender will pass 200 messages of 512 bytes
Time: 3.615
rpi4@rpi4:~$ hackbench --pipe --threads
Running in threaded mode with 10 groups using 40 file descriptors each (== 400 tasks)
Each sender will pass 100 messages of 100 bytes
Time: 0.237
```

Fonte: Autor.

## 4.2 Simulação de carga

Uma das etapas importantes na medição de latência de um sistema computacional é estabelecer uma carga similar àquela que o dispositivo estará executando quando empregado em uma aplicação real. A opção mais precisa envolveria instanciar a aplicação de tempo real no dispositivo, permitindo uma avaliação mais objetiva da latência do sistema. Embora essa seja uma opção ideal, focaremos em simular uma carga genérica. Para isso, algumas ferramentas permitem realizar essa simulação de diversas formas em sistemas Linux, como o *hackbench* e o *stress-ng*.

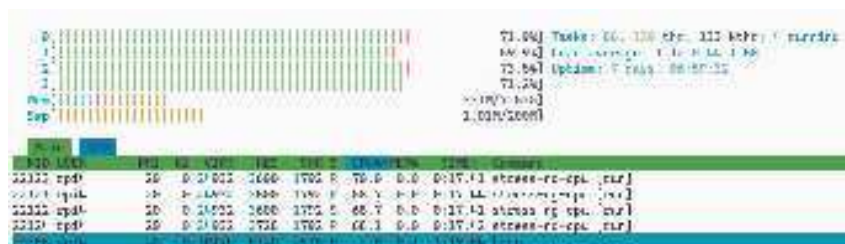
O *Hackbench* é tanto um benchmark quanto um teste de estresse para o escalonador do kernel Linux. Sua principal função é criar um número específico de pares de entidades escalonáveis (threads ou processos tradicionais) que se comunicam por meio de sockets ou pipes, cronometrando quanto tempo cada par leva para enviar dados e recebê-los (MCKENNEY, 2006).

Um exemplo de uso da ferramenta pode ser observado na Figura 6, onde o comando é executado em dois modos distintos. No primeiro modo, a ferramenta é configurada para enviar dados de até 512 bytes por pacote (-s 512). Cada thread remetente (sender) transmite 200 pacotes, e cada thread receptora (receiver) recebe a mesma quantidade (-l 200). São criados 15 grupos de remetentes e receptores (-g 15), sendo que cada thread sender/receiver cria 25 descritores de arquivos. Além disso, cada thread é gerada por uma thread pai (-P).

Na segunda execução, o Hackbench é iniciado em modo de comunicação via pipes (-pipe). Nesse modo, os dados são transmitidos entre remetentes e receptores utilizando pipes anônimos do sistema operacional — estruturas de comunicação unidirecional que permitem a troca de dados entre processos ou threads relacionadas. Esse tipo de comunicação simula um cenário mais próximo de aplicações reais que utilizam IPC (Inter-Process Communication).

Embora a ferramenta ofereça recursos para a realização de testes de carga, ela apresenta limitações em termos de flexibilidade para explorar outras características da arquitetura de hardware. O Hackbench se restringe basicamente à simulação da trans-



Figura 9 – Chamada do `stress-ng` para estresse de CPU nível definido.

Fonte: Autor.

A Figura 8 apresenta a execução do comando `stress-ng` com diversas opções de carga simultâneas. Esse comando é utilizado para submeter o sistema a diferentes tipos de estresse computacional com o objetivo de avaliar sua estabilidade e desempenho sob condições extremas. A chamada `stress-ng -cpu 4 -vm 2 -hdd 1 -fork 8 -timeout 2m -metrics` pode ser interpretada da seguinte forma:

- `-cpu 4`: ativa quatro trabalhadores que exercem carga intensiva de CPU, utilizando ciclos de cálculo matemático.
- `-vm 2`: cria duas tarefas que alocam e operam sobre blocos de memória, simulando uso intensivo de RAM.
- `-hdd 1`: inicia uma tarefa que realiza operações de escrita e leitura em disco, simulando atividade de I/O em disco rígido.
- `-fork 8`: gera oito processos filhos simultâneos, testando a capacidade de gerenciamento de processos pelo sistema operacional.
- `-timeout 2m`: define que o teste será executado durante dois minutos.
- `-metrics`: ao final da execução, exibe métricas detalhadas sobre o uso dos recursos e o comportamento do sistema durante o teste.

Essa combinação permite um teste mais completo, cobrindo simultaneamente CPU, memória, disco e gerenciamento de processos, oferecendo uma visão abrangente da resiliência do sistema sob carga mista. Este exemplo ilustra bem a versatilidade dos testes que podem ser executados com o `stress-ng`.

Ainda, é possível estabelecer um nível de carga passando o argumento `-cpu-load <Porcentagem de uso da CPU>`, isto permite estabelecer uma carga relativa ao uso de CPU, conforme disponível na Figura 9, em que o uso da CPU foi definido uma carga média de 70%.

O `stress-ng` deve ser usado com cuidado, pois foi projetado para fazer com que o sistema trabalhe de forma intensar e desencadeasse problemas de *hardware* e erros no

sistema operacional que só ocorrem quando o sistema está sobrecarregado. Alguns testes podem resultar em um sobreaquecimento do *hardware* (mal projetado), e também pode causar sobrecargas excessivas no sistema.

### 4.3 Container

Containers são unidades padronizadas de software que embalam o código e suas dependências, permitindo que a aplicação seja executada de maneira confiável em diferentes ambientes. Como já mencionado anteriormente, os *container* são uma forma de virtualização baseada em *software*, que é mais leve e rápida do que as máquinas virtuais.

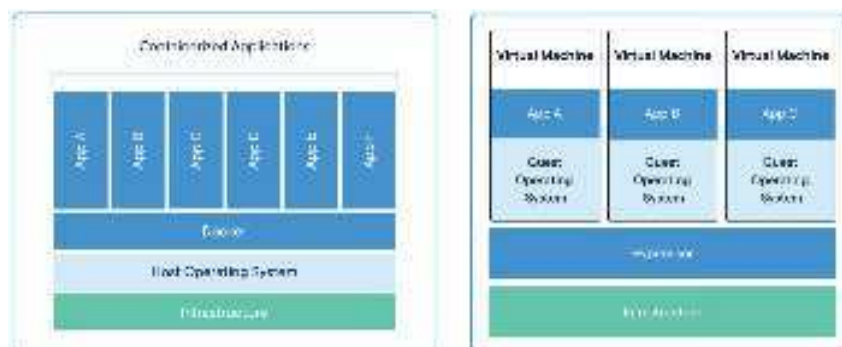
O *container* é uma tecnologia que permite isolar processos, executando-os em um ambiente separado e customizado com recursos e dependências específicas, utilizando funcionalidades do *kernel* do sistema operacional. O conceito inicial surgiu em 1979 com o *chroot*, que possibilita alterar o diretório raiz de um processo. Posteriormente, os *cgroups*, introduzidos no *kernel* do Linux em 2007, trouxeram controle mais refinado sobre recursos de sistema. Esses avanços deram origem a ferramentas modernas como o LXC (Linux Containers) no final dos anos 2000. Desde então, os contêineres foram amplamente aprimorados e são reconhecidos como ferramentas essenciais para abstrair e padronizar ambientes de desenvolvimento e produção, seja em dispositivos de borda ou em serviços na nuvem (KOZHIRBAYEV; SINNOTT, 2017).

Os containers ganharam destaque no mercado de *cloud computing* por oferecerem um ambiente padronizado, que soluciona problemas de dependências entre diferentes máquinas. Eles permitem o escalonamento ágil de aplicações na nuvem, possuem execução mais leve em comparação com máquinas virtuais e apresentam tempos de inicialização significativamente menores. Devido a algumas dessas características, o *container* têm sido associado como uma tecnologia facilitadora para aplicações de diferentes portes, habilitando a portabilidade de aplicações e sendo um auxiliar para melhorar a robustez dos sistemas.

A Figura 10 ilustra a diferença entre aplicações baseadas em containers e aquelas baseadas em máquinas virtuais. Enquanto o *container engine* é executado diretamente sobre o sistema operacional do hospedeiro, as VMs dependem de uma camada adicional chamada *hypervisor*. A principal diferença prática é que containers compartilham o *kernel* do sistema hospedeiro, enquanto cada VM possui seu próprio *kernel*, o que resulta em um consumo significativamente maior de recursos, como memória e poder de processamento.

Devido à popularidade dos containers no setor de TI, diversas plataformas surgiram com o objetivo de facilitar sua adoção e gerenciamento. Dentre elas, o Docker foi responsável por popularizar a tecnologia, ao oferecer uma solução open-source que abstrai e simplifica o uso de containers, permitindo desenvolver, instanciar e gerenciar aplicações

Figura 10 – Comparação entre containers e VMs.



Fonte: Docker.com, 2025.

de forma eficiente, em um ambiente padronizado e portátil.

Com a crescente adoção de containers, tornou-se necessário dispor de ferramentas capazes de gerenciar grandes volumes de containers distribuídos em ambientes escaláveis. Nesse cenário, o *Kubernetes* consolidou-se como a principal plataforma de orquestração de containers, amplamente adotada por sua capacidade de automatizar tarefas como escalonamento dinâmico de serviços, balanceamento de carga, monitoramento de aplicações e atualizações contínuas com alta disponibilidade.

Algumas dessas funcionalidades também são oferecidas pelo Docker por meio do *Docker Compose* e do *Docker Swarm*. O *Docker Compose* permite definir e executar aplicações multi-containers a partir de arquivos de configuração YAML, facilitando o gerenciamento em ambientes de desenvolvimento. Já o *Docker Swarm* é uma solução nativa de orquestração do Docker, que oferece recursos como criação de clusters, distribuição de containers entre nós e balanceamento de carga, embora com menor complexidade e flexibilidade em comparação ao *Kubernetes*.

Desde 2019, o Docker passou a utilizar o *containerd* como seu runtime padrão de containers. O *containerd* é um runtime leve e de alto desempenho, originalmente desenvolvido como parte do próprio Docker, mas que posteriormente foi separado em um projeto independente e mantido pela Cloud Native Computing Foundation (CNCF) ([containerd maintainers, 2024](#)). O que contribui para a confiabilidade e longevidade do ecossistema de containers. Com isso, o Docker mantém sua relevância e continua sendo uma ferramenta robusta tanto para desenvolvimento local quanto para uso em ambientes de produção.

A adoção do *containerd* trouxe diversos benefícios para o ecossistema Docker. Primeiramente, ela promove uma maior modularização da arquitetura do Docker, permitindo que componentes como o gerenciamento de imagens, a execução de containers e a interface de usuário sejam mantidos e evoluídos de forma mais independente. Isso torna o Docker mais leve, mais estável e mais interoperável com outras ferramentas do ecossistema de containers.

Além disso, o *containerd* é um runtime compatível com o padrão *Container Runtime Interface* (CRI), exigido por plataformas de orquestração como o Kubernetes. Essa compatibilidade torna mais fácil a integração entre o Docker e ambientes de orquestração de containers, além de garantir maior conformidade com os padrões da indústria.

Containers são instâncias de imagens, que por sua vez podem ser obtidas a partir de repositórios locais ou remotos (na nuvem), ou ainda podem ser criadas manualmente pelo usuário. No Docker, a criação de uma imagem personalizada é feita por meio de um arquivo de configuração chamado *Dockerfile*. Esse arquivo, escrito em formato de texto simples, define de forma estruturada as instruções necessárias para a construção da imagem, como a escolha da imagem base, os comandos a serem executados, as dependências a serem instaladas e os arquivos a serem incluídos no ambiente do *container*.

A Figura 11 apresenta a descrição de um *Dockerfile* baseado em uma imagem *Ubuntu*. Durante o processo de *build* do *container*, são executadas instruções para a instalação das ferramentas *stress-ng* e *cyclictest*. Alguns parâmetros e comandos, como *no-install-recommends* e *apt-get clean*, auxiliam na manutenção de um *container* mais leve, mantendo apenas os pacotes essenciais.

O *Dockerfile* pode ser compilado em uma imagem por meio do comando `docker build -t <tag para a imagem> <caminho raiz do build>`. Em um primeiro momento, o *Docker* irá baixar a imagem base do *Ubuntu 24.10* a partir do repositório oficial, executando em seguida as instruções especificadas no arquivo para construir a imagem final.

Figura 11 – *Dockerfile* para criação de *container* padronizado para execução dos testes.

```
Dockerfile.txt > ...
1 FROM ubuntu:24.10
2
3 ENV DEBIAN_FRONTEND=noninteractive
4
5 RUN apt-get update && \
6     apt-get install -y --no-install-recommends \
7         stress-ng \
8         rt-tests \
9     && apt-get clean && \
10    rm -rf /var/lib/apt/lists/*
11
12 CMD ["/bin/bash"]
```

Fonte: Autor.

## 4.4 Cenários de Teste

A seguir, os experimentos realizados nos dispositivos serão referenciados como *casos de teste*. Ao todo, foram realizados experimentos em oito cenários distintos com a

finalidade de avaliar o comportamento do sistema ao containerizar aplicações em ambientes industriais. Para simplificar a análise dos resultados, os cenários foram agrupados em dois conjuntos de teste.

O primeiro conjunto foi projetado para avaliar o desempenho do sistema ao inserir uma carga ou aplicação diretamente como processo no sistema nativo (*host*) ou, alternativamente, ao alocar essa carga dentro de um *container*. A ideia é construir um *benchmarking* a partir de diferentes arranjos de configuração, comparando o desempenho entre o processamento nativo e o containerizado.

O segundo conjunto de casos, por sua vez, visa avaliar de forma simultânea a influência que um *container* pode exercer sobre outro, bem como o impacto no próprio sistema operacional hospedeiro.

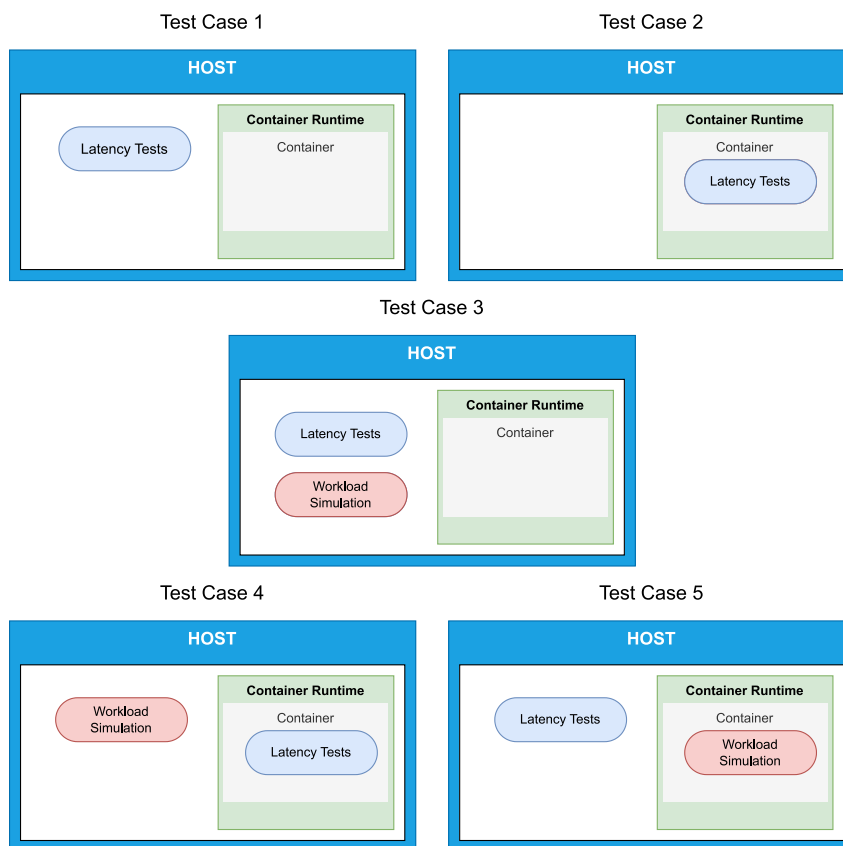
#### 4.4.1 Primeiro Conjunto de Testes

Os experimentos realizados consistem dos seguintes casos de testes, a Figura 13 contém a ilustração do que está descrito logo a seguir.

1. Execução da ferramenta Cyclictest no *host* sem cargas no sistema.
2. Execução da ferramenta Cyclictest dentro de uma instância do *container* sem cargas no sistema.
3. Execução da ferramenta Cyclictest no *host* com simulação de carga de trabalho no *host*.
4. Execução da ferramenta Cyclictest na instância de *container* e simulação de carga de trabalho no *host*.
5. Execução do Cyclictest no *host* e simulação de carga de trabalho dentro do *container*.

O Caso de Teste 1, embora não esteja diretamente ligado ao uso de contêineres na aplicação, serve como referência para a comparação de sistemas baseados em arquiteturas containerizadas. Esse teste permite avaliar a latência média do sistema operando sem cargas adicionais. Em outras palavras, o teste demonstra como o dispositivo e o conjunto de software fundamentais para seu funcionamento básico impactam o desempenho do sistema.

Para os demais teste, é inserido no sistema um *container* Docker para avaliar o impacto no sistema. O Caso de Teste 2 permite medir a latência base de uma instância de *container*. Esse teste pode ser expandido para comparar diferentes tecnologias de *container*, configurações e o impacto de gerenciadores automáticos de recursos. O Caso de Teste 3 segue a mesma ideia do Teste 1, com a diferença de que o sistema é submetido a uma

Figura 12 – Primeiro conjunto de teste para definição de *benchmarking* para avaliação de latência.

Fonte: Autor.

carga elevada para avaliar seu desempenho médio sob condições de carga de trabalho. O Caso de Teste 4 avalia o impacto na execução do *container* quando o *host* é submetido a uma carga elevada. Por fim, o Caso de Teste 5 complementa a ideia apresentada no Teste 4. Nesse caso, é possível analisar o impacto no *host* quando o *container* executa a carga de trabalho.

Para avaliar o desempenho do sistema, configuramos uma carga simulada que consome recursos da CPU para reproduzir a carga de trabalho projetada para o dispositivo utilizando a ferramenta **stress-ng**. A carga foi definida de modo a utilizar todos os núcleos do processador, atribuindo um *worker* para cada núcleo disponível, a taxa de utilização constante. Além disso, um teste adicional foi realizado variando a taxa de utilização da CPU de 0% (sem carga) e aumentada progressivamente em incrementos de 10%, a fim de observar o comportamento do sistema em diferentes pontos de operação.

Foi utilizado Docker para compilar uma imagem de *container* que contém as ferramentas necessárias para facilitar a configuração dos testes e execução e das ferramentas na plataforma de *hardware*. O Docker é uma ferramenta que dispõe de recursos para facilitar a criação de imagens e gestão do ciclo de vida de um *container*.

Os comandos de terminal necessários para a reprodutibilidade dos experimentos descritos neste texto estão disponíveis a seguir:

```
t=60 # tempo de amostragem [s]
T=$(expr $t + 10) # tempo de stress [s]

## Teste 1
sudo Cyclictest -h500 -p99 -q \
  -D${t}s > ./tmp/test1.txt

## Teste 2
docker run --cap-add=sys_nice -it --rm \
  --name rpi-instance rpi-stress bash -c \
  "Cyclictest -h500 -p99 -q -D${t}s" \
  > ./tmp/test2.txt
docker rm rpi-instance

## Teste 3
stress-ng --cpu 4 -t${T}s &
pid=$!
sudo Cyclictest -h500 -p99 -q -D${t}s \
  > ./tmp/test3.txt
echo "waiting stress-ng to finish"
wait $pid

## Teste 4
stress-ng --cpu 4 -t${T}s &
pid=$!
docker run --cap-add=sys_nice -it --rm --name rpi-instance rpi-stress
  bash -c "Cyclictest -h500 -p99 -q -D${t}s" > ./tmp/test4.txt
docker rm rpi-instance
wait $pid

## Teste 5
docker run --cap-add=sys_nice -d --rm \
  --name rpi-instance rpi-stress \
  bash -c "stress-ng --cpu 4 -t${T}s"
sudo Cyclictest -h500 -p99 -q -D${t}s \
  > ./tmp/test5.txt
docker wait rpi-instance

echo "FIM"
```

#### 4.4.2 Segundo Conjunto de Testes

Assim como no primeiro conjunto de experimentos, a proposta neste segundo momento é estabelecer um novo *benchmark*, desta vez por meio da modificação da estrutura

dos testes. O objetivo é analisar o impacto que uma aplicação — executada em ambiente *containerizado* ou diretamente no sistema — pode ocasionar tanto em outro container em execução quanto no próprio sistema operacional hospedeiro. Esta abordagem permite avaliar o grau de interferência mútua entre aplicações e a sobrecarga imposta ao ambiente de execução, fornecendo dados relevantes para decisões de alocação e isolamento de processos em sistemas virtualizados.

Este segundo conjunto é composto por três casos de teste, rotulados de A a C:

- **Caso de Teste A:** Um único *container* e o sistema operacional hospedeiro executam, simultaneamente, o *cyclictest*, com o intuito de medir a latência de temporização em ambos os ambientes.
- **Caso de Teste B:** Configuração semelhante ao Caso A, porém com a introdução de uma carga adicional executando diretamente no sistema hospedeiro. O objetivo é avaliar o impacto dessa carga externa sobre a precisão temporal do *cyclictest*.
- **Caso de Teste C:** Também baseado no Caso A, mas com a inclusão de um segundo *container* executando uma carga ou aplicação adicional. Esse cenário busca observar os efeitos de interferência entre múltiplos *containers* e seu reflexo nas medições de latência.

Os testes conforme disposto na Figura 13 foram executados com através do *script* a seguir:

```
## Test Case A

echo "Running Test Case A"
sudo cyclictest -h500 -p99 -q -D${t}s \
  > ./tmp/testA1.txt &
pid=$!
docker run --cap-add=sys_nice -it --rm --name \
  rpi-instance rpi-stress \
  bash -c "cyclictest -h500 -p99 -q -D${t}s" \
  > ./tmp/testA2.txt
wait $pid
docker wait rpi-stress

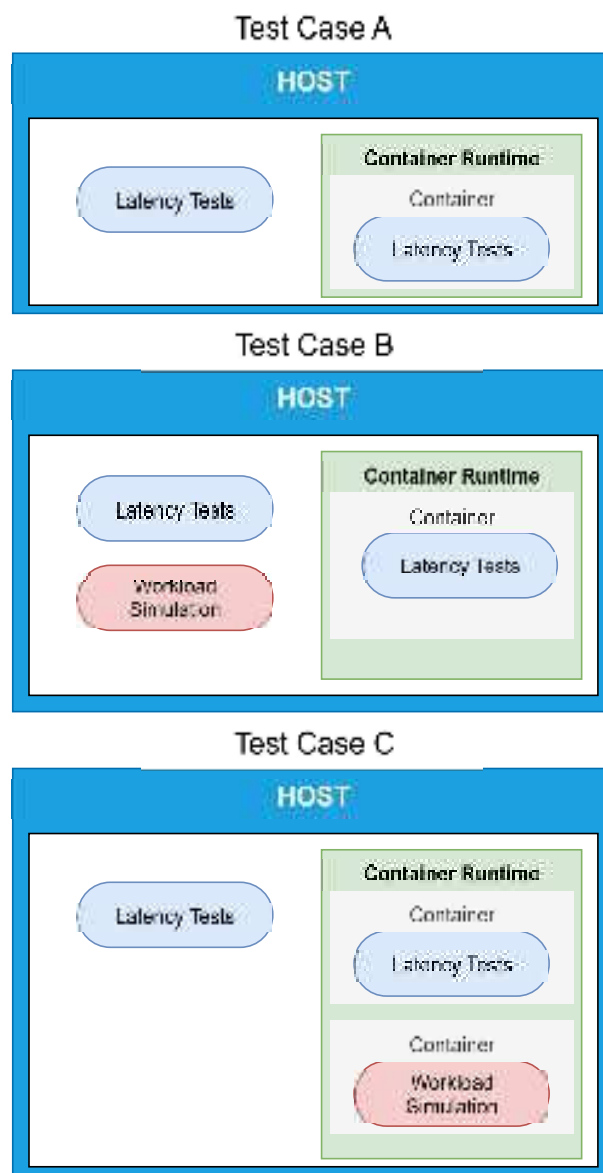
sleep 1

## Test Case B

echo "Running Test Case B"

chrt 97 stress-ng --cpu 1 --cpu-load 70 -t${T}s &
pid=$!
```

Figura 13 – Segundo conjunto de testes para definição de *benchmarking* para avaliação de latência.



Fonte: Autor.

```
sudo cyclictest -h500 -p99 -q -D${t}s \
    > ./tmp/testB1.txt &
docker run --cap-add=sys_nice -it --rm --name \
    rpi-instance rpi-stress \
    bash -c "cyclictest -h500 -p99 -q -D${t}s" \
    > ./tmp/testB2.txt
#docker rm rpi-instance
wait $pid
sleep 1

## Test Case C
echo "Running Test Case C"
```

```
sudo cyclictest -h500 -p99 -q -D${t}s \  
  > ./tmp/testC1.txt &  
  
docker run --cap-add=sys_nice -it --rm --name \  
  rpi-instance rpi-stress \  
  bash -c "chrt 97 stress-ng --cpu 1 --cpu-load 70 -t${T}s &  
  cyclictest -h500 -p99 -q -D${t}s" \  
  > ./tmp/testC2.txt  
  
echo "waiting stress-ng to finish"  
docker wait rpi-instance
```

## 5 Resultados

Os resultados estão organizados em duas seções: a primeira é dedicada aos resultados obtidos no *Raspberry Pi 4*, por meio da execução do primeiro conjunto de testes discutido anteriormente; a segunda seção apresenta os resultados referentes à execução dos dois conjuntos de testes no *DCN Smar*.

### 5.1 Resultados Raspberry Pi 4

Esta seção apresenta os resultados obtidos a partir da execução dos testes realizados em um *Raspberry Pi 4* Modelo B. Os testes permitiram a obtenção de dados relacionados ao atraso de resposta do sistema. Esses resultados estão apresentados na forma de gráficos e tabelas, os quais incluem estatísticas de latência para cada teste realizado, incluindo o comportamento médio e marginal do sistema.

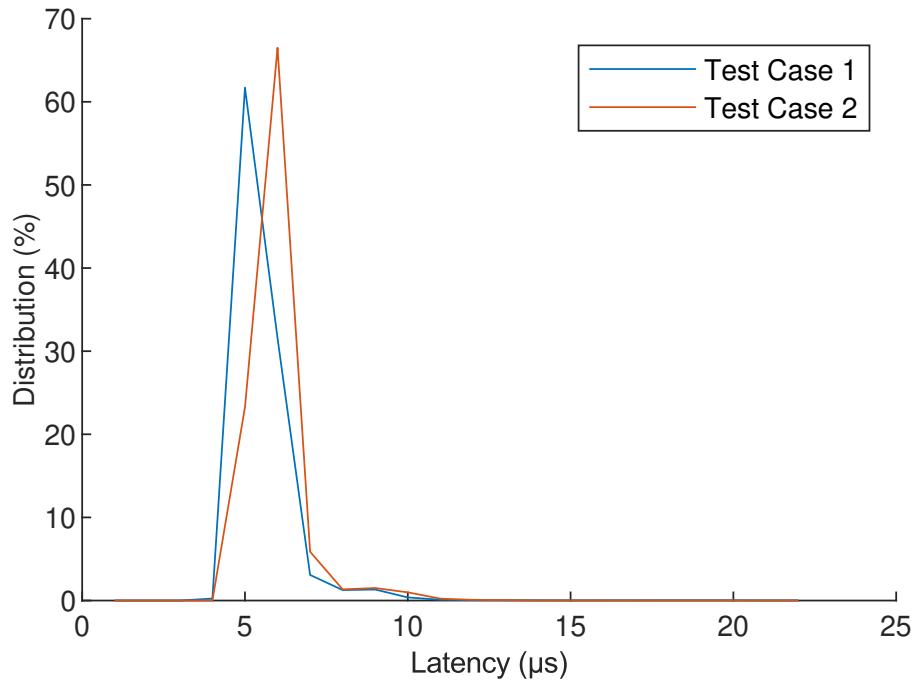
A Figura 14 apresenta a latência normalizada do sistema medida fora e dentro de um *container*, conforme descrito na Figura 13 pelos Casos de Teste 1 e 2, em que o sistema opera sem cargas significativas. O gráfico evidencia que a execução dentro do *container* introduz um atraso no tempo de resposta do sistema. Embora a latência média tenha permanecido inalterada, a latência mínima aumentou de 3  $\mu\text{s}$  para 4  $\mu\text{s}$  (33%), enquanto a latência máxima passou de 133  $\mu\text{s}$  para 172  $\mu\text{s}$  (29,32%), conforme indicado na Tabela 1. Deste modo, é possível observar a latência base do sistema operacional, assim como também avaliar o desempenho do *container*.

Tabela 1 – Latência média, mínima e máxima obtida para cada um dos casos de teste avaliados.

Caso de Teste	1	2	3	4	5
Latência mínima ( $\mu\text{s}$ )	3	4	5	5	4
Latência média ( $\mu\text{s}$ )	4,56	5	7,89	8,17	7,67
Latência máxima ( $\mu\text{s}$ )	133	172	155	214	134

A Figura 15 apresenta a resposta do sistema quando é adicionado uma carga ao sistema, conforme proposto nos Casos de Teste 3, 4 e 5. Para efeito de comparação, o Caso de Teste 3 é adotado como referência, pois exibe o comportamento padrão de aplicações que são executadas diretamente no sistema nativo sem o uso de contêineres. o Caso de Teste 4 ilustra como a carga fora do *container* influencia a latência observada pelo *container*, e o Caso de Teste 5 demonstra o efeito de introduzir uma carga de trabalho dentro do *container*. A partir da comparação das curvas no gráfico, é possível observar que há atrasos ao se utilizar o *container*, mas não chega a ser uma diferença significativa.

Figura 14 – Distribuição de latência normalizada para os casos de teste 1 (sem *container*) e o caso de teste 2 (com *container*).



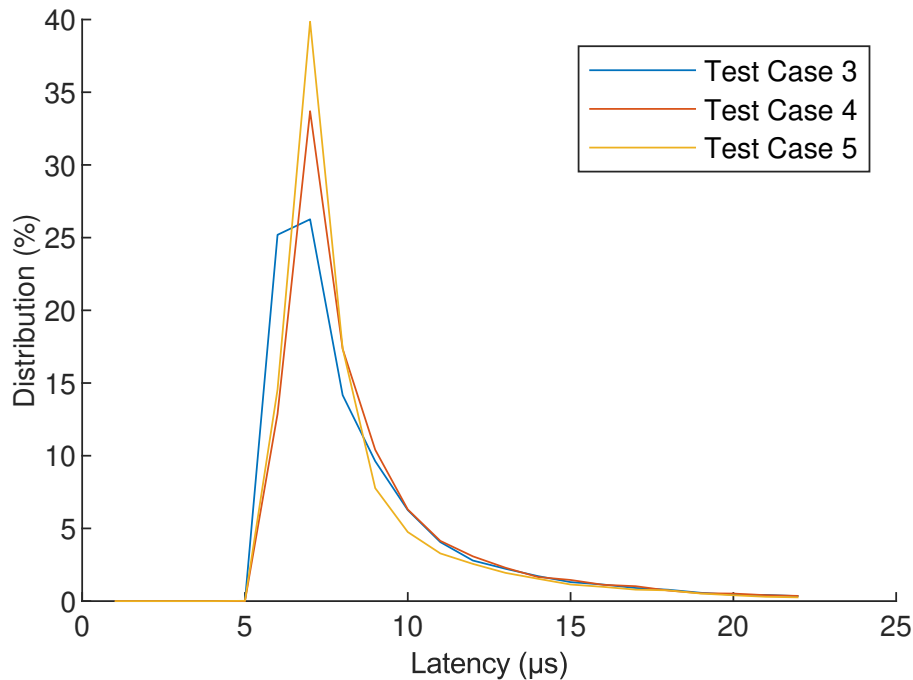
Fonte: Autor.

Dos dados extraídos (Tabela 1) em relação a referência observou-se que para o Caso de Teste 4, a latência mínima permaneceu constante, a latência média aumentou de  $7 \mu\text{s}$  para  $8 \mu\text{s}$  (14%), e a latência máxima aumentou de  $155 \mu\text{s}$  para  $214 \mu\text{s}$ . Já em relação ao Caso de Teste 5, a latência mínima diminuiu de  $5 \mu\text{s}$  para  $4 \mu\text{s}$  (-20%), a latência média manteve-se constante e a latência máxima diminuiu de  $155 \mu\text{s}$  para  $134 \mu\text{s}$  (-13%). Da comparação acima, é possível inferir que quando há uma sobrecarga no *host*, a latência vista dentro do *container* é maior face ao que é visto no *host*, e quando há uma sobrecarga dentro do *container* a latência não é tão sentida pelo *host*.

Vale ressaltar que os valores de máximo e mínimo apresentados na Tabela 1 referem-se a casos pontuais do sistema. Uma visualização estatística mais detalhada dos dados do experimento pode ser observada na Figura 16, onde o eixo vertical foi ajustado para melhorar a clareza. Pode-se notar semelhanças entre os Casos de Teste 3 e 5, destacando uma variância máxima menor quando ocorre uma sobrecarga dentro do *container*.

A Figura 17 mostra os resultados obtidos ao analisar a influência da gestão de economia de energia para aplicações IoT. Durante a execução de um teste de estresse progressivo dentro de um *container*, avaliamos dois cenários: o rótulo *On Demand* representa os resultados quando o sistema operacional ajusta dinamicamente a frequência da CPU com base na demanda da carga de trabalho. Além disso, os resultados quando a CPU opera com capacidade máxima são rotulados como *Performance*. Para uma apli-

Figura 15 – Distribuição de latência normalizada obtida segundo o caso de teste 3 (sem *container*) e os casos de teste 4 e 5 (com *container*), todos executados com carga de trabalho no sistema.



Fonte: Autor.

cação não containerizada, o teste de estresse foi executado fora do Docker. A diferença entre a aplicação containerizada e a não containerizada é mínima, com uma maior perda de desempenho ocorrendo quando a CPU está com 100% de uso. O ajuste dinâmico da CPU introduziu mais latência no sistema para cargas de trabalho baixas do que o uso de contêineres.

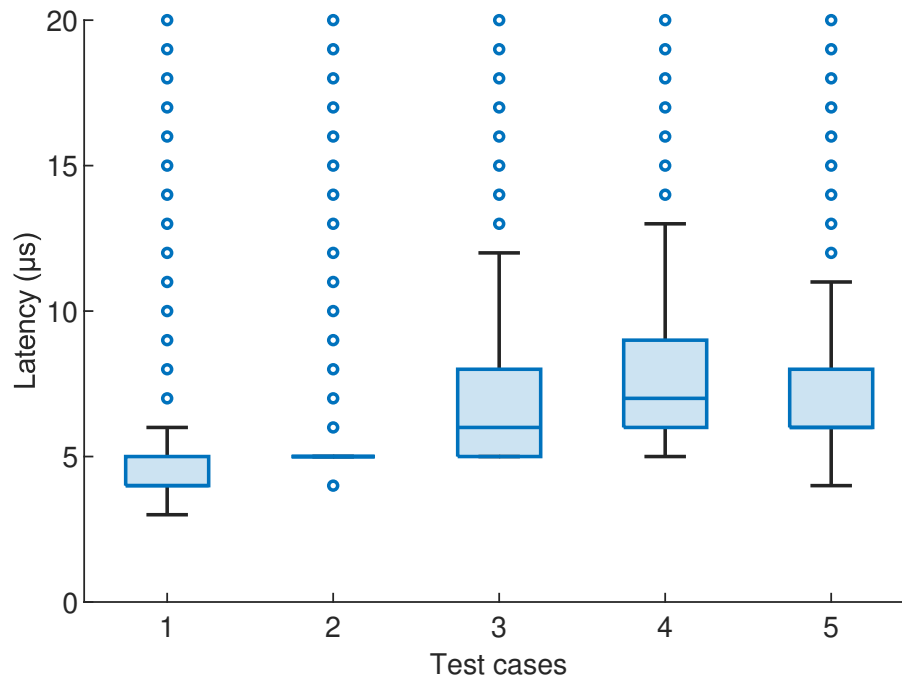
Além disso, para garantir resultados consistentes, a frequência do processador foi fixada. Em particular, os casos de teste foram realizados na capacidade máxima de processamento do *hardware*, permitindo explorar os limites de operação do dispositivo em cada cenário avaliado. Adicionalmente, recursos de economia de energia e ajustes dinâmicos de frequência de processamento foram temporariamente desabilitados com o mesmo objetivo.

## 5.2 Resultados DCN Smar

Para o *DCN* da Smar, foram avaliados os mesmos cenários que para o caso do *Raspberry Pi 4*. Embora sejam dispositivos distintos, os mesmos cenários podem ser seguidos para avaliar o desempenho do sistema ao adotar aspectos de containerização na aplicação.

Algumas adaptações no código disponibilizado para executar os cenários são ne-

Figura 16 – Análise da Variação de Latência



Fonte: Autor.

cessárias para ajustar as diferenças de *hardware* entre os sistemas. É preciso reduzir o número de *threads* gerados pelo *stress-ng* para estressar o dispositivo, pois a CPU do *DCN* possui apenas dois núcleos de processamento, e um deles está desabilitado.

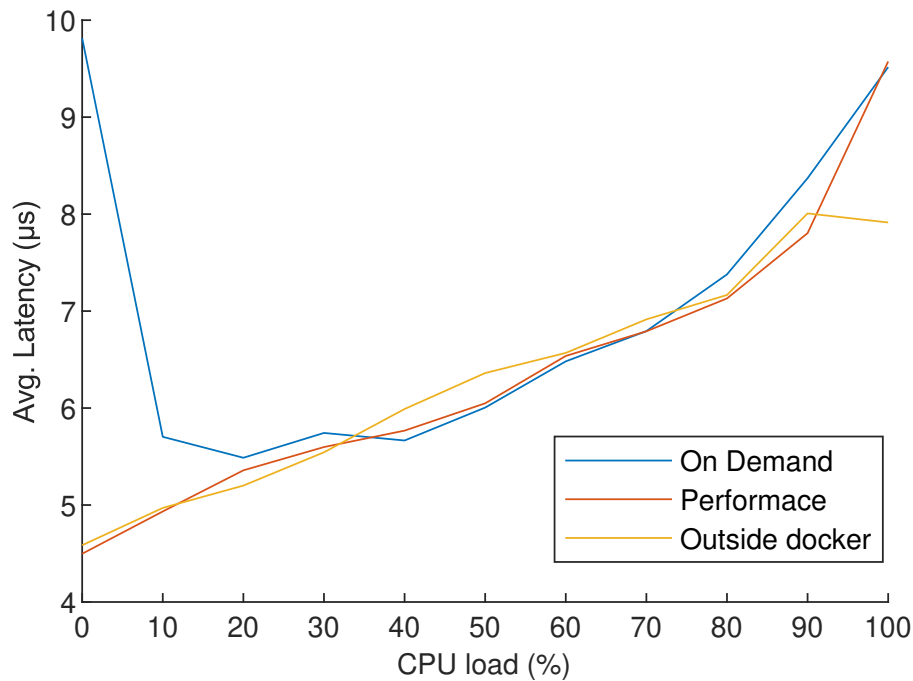
Outra alteração a ser levada em consideração é o quanto o teste irá estressar o dispositivo. Como o dispositivo possui apenas um núcleo, há a possibilidade de o estresse comprometer a integridade do sistema, gerando falhas na conclusão de outras tarefas de tempo crítico ou essenciais para o funcionamento do *SO*. Isso ocorre porque tarefas atribuídas como tempo real possuem maior prioridade que as demais, podendo ocupar todo o *slot* de tempo da CPU.

Dessa forma, foi reduzido o estresse máximo da carga para usar até 70% da capacidade total da CPU do *DCN*, e o número de núcleos foi reduzido para 1. Os resultados dos cenários 1 e 2 foram plotados na Figura 18, enquanto os resultados dos cenários 3, 4 e 5 foram plotados na Figura 19.

Na Figura 18 observamos a latência do sistema e do *container*, O Caso de Teste 1 apresentou uma latência média de  $6,16 \mu s$ , enquanto no Caso de Teste 2 foi observado uma latência média de  $7,93 \mu s$ . Desprezando a carga computacional devido a execução do *cyclictest*, podemos inferir que o *container* nesses dois cenários elevou o *overhead* em  $1,77 \mu s$  (+28,73%).

A latência máxima observada nos dois primeiros cenários não foi muito significativa. Em algumas poucas ocorrências, foram obtidas latências de  $35 \mu s$  e  $44 \mu s$  para os

Figura 17 – Teste progressivo de estresse da CPU.



Fonte: Autor.

cenários 1 e 2, respectivamente. Na maioria dos casos, a latência máxima relativa não ultrapassou os  $18 \mu\text{s}$  para ambos os cenários.

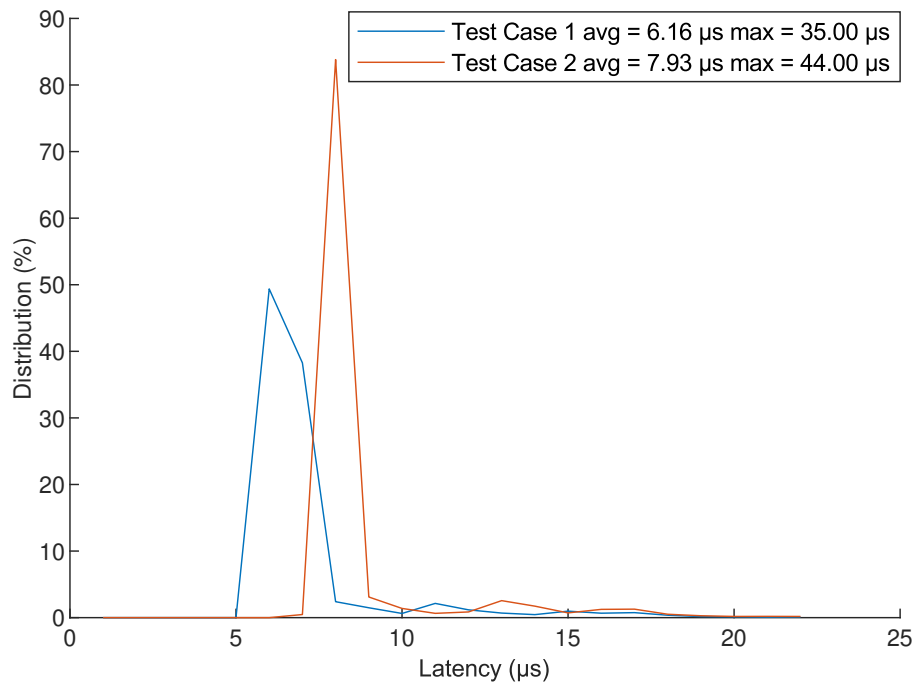
Na Figura 19 são apresentados os resultados obtidos a partir da execução dos cenários 3, 4 e 5. No terceiro cenário, os resultados mostraram-se bastante consistentes, sendo similares aos observados no *Raspberry Pi 4*, com latência média de aproximadamente  $6,91 \mu\text{s}$  e latência máxima de  $33 \mu\text{s}$ . Nos cenários quatro e cinco, as latências observadas exibem uma característica típica de distribuição bimodal, indicando a presença de dois comportamentos distintos no sistema. As médias ponderadas das latências nesses cenários foram de  $13,11 \mu\text{s}$  e  $12,16 \mu\text{s}$ , respectivamente.

Ao compararmos as latências observadas nos casos 3 e 5, verifica-se que a latência aumentou de  $6,91 \mu\text{s}$  para  $12,16 \mu\text{s}$  com a migração da carga do *host* para o *container*, representando um acréscimo de  $5,25 \mu\text{s}$  (ou  $75,97\%$ ) na latência do sistema. Um aumento similar é observado no caso 4, que também envolve a utilização de *containers*. Esse incremento é consideravelmente superior ao observado entre os casos 1 e 2, nos quais a adoção do *container* resultou em um aumento de apenas  $28\%$  na latência.

Ao analisarmos os casos 4 e 5, nota-se que a execução de cargas no *host* impacta negativamente a latência no *container*. Em contrapartida, a execução de uma carga adicional em um *container* não provoca impacto perceptível na latência do sistema *host*.

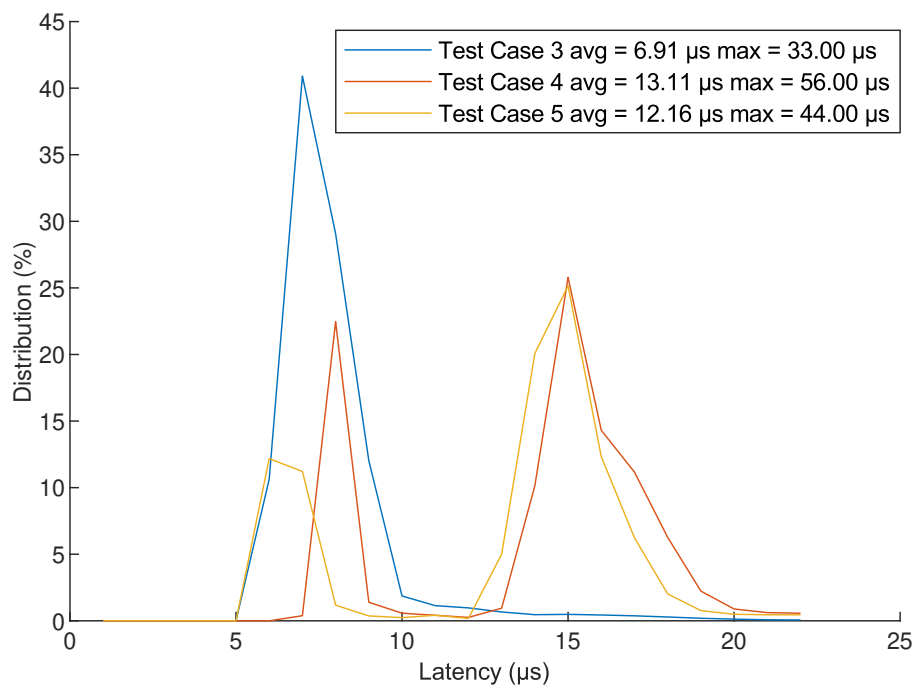
Por fim, os resultados do segundo conjunto de testes são apresentados a seguir. A Figura 20 exhibe os resultados de latência obtidos no cenário A; a Figura 21 apresenta os

Figura 18 – Execução dos cenários 1 e 2 no DCN Smar.



Fonte: Autor.

Figura 19 – Execução dos cenários 3 e 5 no DCN Smar.

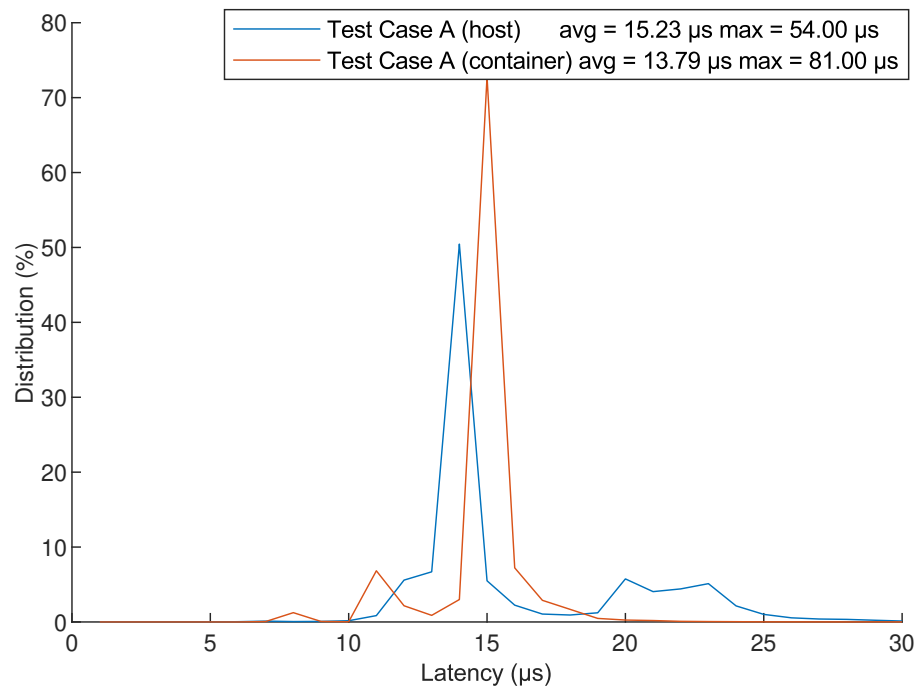


Fonte: Autor.

resultados correspondentes ao cenário B; e a Figura 22 mostra os resultados referentes ao cenário C.

Observa-se que o cenário A corresponde à combinação dos cenários 1 e 2 do primeiro conjunto de testes. No entanto, os resultados apresentados na Figura 20 evidenciam um comportamento divergente em relação ao esperado. Não foi encontrada uma explicação plausível para essa discrepância, especialmente considerando que o comportamento se manteve consistente em múltiplas execuções dos testes.

Figura 20 – Resultados do cenário A executados no DCN Smar.



Fonte: Autor.

Nos cenários seguintes, foi inserida uma carga simulada no sistema, correspondendo a aproximadamente 70 % do consumo computacional do *DCN*. No cenário B, a carga foi alocada diretamente no *host*, e o *cyclictest* registrou uma latência média maior no *host* em comparação ao *container*. Apesar disso, a diferença observada é relativamente pequena, representando uma variação percentual de  $-5,68\%$  em relação à latência média medida no *host*.

No cenário C, a carga foi migrada para um *container*. Uma análise direta em comparação com o cenário B demonstra uma redução na latência média tanto no *host* quanto no *container*, indicando que a execução da carga dentro de um *container* contribuiu para a mitigação de latências elevadas no sistema. Observou-se uma redução de  $14,45\%$  na latência medida no *host* e de  $10,85\%$  na latência do *container*.

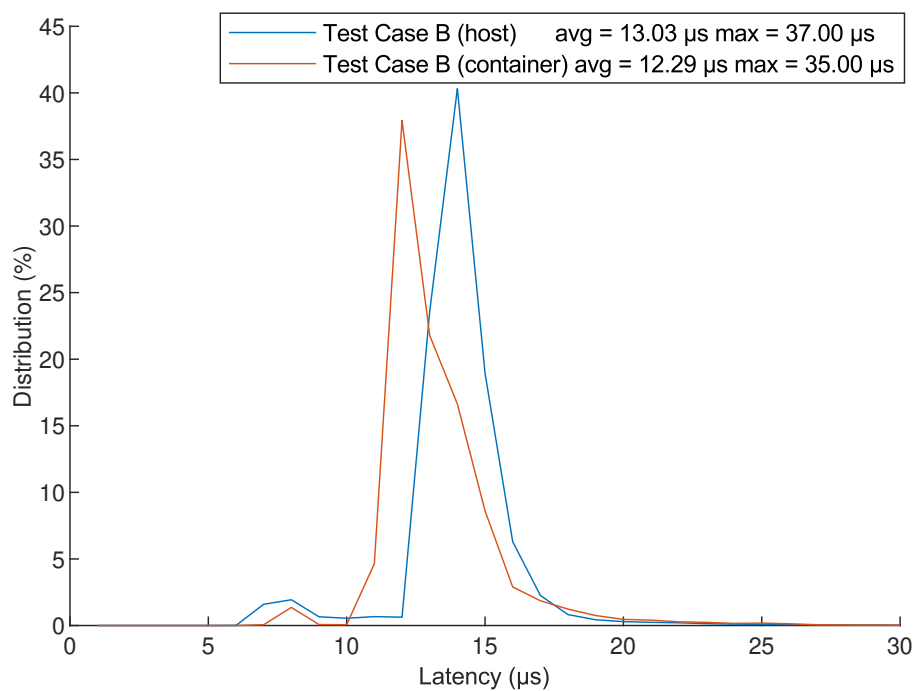


Figura 21 – Resultados do cenário B executados no DCN Smar.

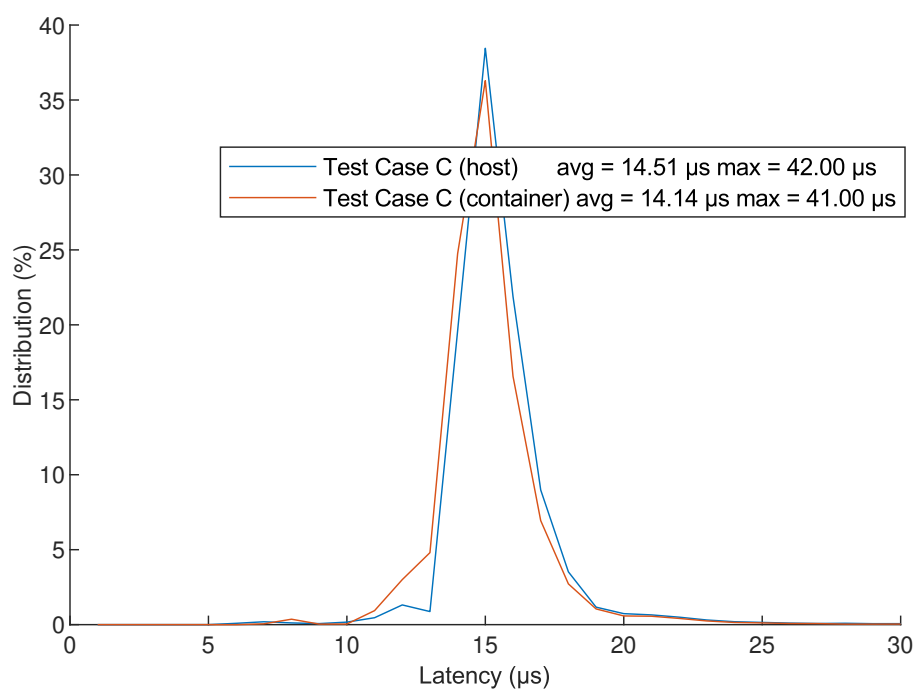


Figura 22 – Resultados do cenário C executados no DCN Smar.

## 6 Considerações Finais

Este trabalho abordou os principais tópicos relacionados à análise de desempenho computacional. Por meio da execução de testes generalistas, foi possível atingir os objetivos estabelecidos para caracterizar o impacto da adoção de *containers* em plataformas de *hardware* com recursos computacionais limitados, voltadas para aplicações de automação industrial e IoT.

Os resultados obtidos demonstraram que a utilização de *containers* impõe um pequeno impacto ao desempenho, atribuído às camadas adicionais de isolamento de processos. No entanto, não foram observadas violações críticas de desempenho que comprometessem, a priori, a execução de tarefas usuais em sistemas de controle de processos, nem em outros cenários onde as restrições computacionais sejam mais flexíveis.

Dessa maneira, os testes propostos ao longo deste trabalho podem ser realizados em qualquer plataforma de *hardware*, auxiliando na análise de desempenho e confiabilidade tanto dos *containers* quanto das aplicações. Assim, é possível inferir se as aplicações respondem dentro de uma faixa admissível de latência para as necessidades específicas de um sistema.

# Referências

- BEÑO, L. et al. Transforming industrial automation: voice recognition control via containerized plc device. *Scientific Reports*, Nature Publishing Group UK London, v. 14, n. 1, p. 29387, 2024. Citado na página 1.
- BUSCH, J.-P.; REIHER, L.; ECKSTEIN, L. Enabling the deployment of any-scale robotic applications in microservice architectures through automated containerization\*. In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. [S.l.: s.n.], 2024. p. 17650–17656. Citado na página 1.
- containerd maintainers. *containerd - An industry-standard container runtime*. 2024. Acessado em: 21 de abril, 2025. Disponível em: <<https://containerd.io/>>. Citado na página 14.
- GIVEHCHI, O. et al. Control-as-a-service from the cloud: A case study for using virtualized plcs. In: IEEE. *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)*. [S.l.], 2014. p. 1–4. Citado na página 4.
- GOLDSCHMIDT, T.; HAUCK-STATTELMANN, S. Software containers for industrial control. In: IEEE. *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. [S.l.], 2016. p. 258–265. Citado na página 5.
- GOLDSCHMIDT, T. et al. Cloud-based control: A multi-tenant, horizontally scalable soft-plc. In: IEEE. *2015 IEEE 8th international conference on cloud computing*. [S.l.], 2015. p. 909–916. Citado na página 4.
- HEGAZY, T.; HEFEEDA, M. Industrial automation as a cloud service. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 26, n. 10, p. 2750–2763, 2014. Citado na página 4.
- KING, C. I. *stress-ng: a tool to load and stress a computer system*. 2024. <<https://github.com/ColinIanKing/stress-ng>>. Acessado em: 2025-04-23. Citado na página 11.
- KOZHIRBAYEV, Z.; SINNOTT, R. O. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, Elsevier, v. 68, p. 175–182, 2017. Citado na página 13.
- MCKENNEY, P. E. *Hackbench: Scheduler and IPC benchmark for Linux*. 2006. <<https://manpages.ubuntu.com/manpages/xenial/man8/hackbench.8.html>> (Accessed: 2024-11-15). Citado na página 10.
- MELLADO, J.; NÚÑEZ, F. Design of an iot-plc: A containerized programmable logical controller for the industry 4.0. *Journal of Industrial Information Integration*, Elsevier, v. 25, p. 100250, 2022. Citado na página 1.
- QUEIROZ, R. et al. Container-based virtualization for real-time industrial systems—a systematic review. *ACM Computing Surveys*, ACM New York, NY, v. 56, n. 3, p. 1–38, 2023. Citado na página 2.

- RANDAL, A. The ideal versus the real: Revisiting the history of virtual machines and containers. *CoRR*, abs/1904.12226, 2019. Disponível em: <<http://arxiv.org/abs/1904.12226>>. Citado na página 1.
- SOLLFRANK, M. et al. Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation. *IEEE Transactions on Industrial Informatics*, IEEE, v. 17, n. 5, p. 3566–3576, 2020. Citado na página 5.
- WILLIAMS, C.; KACUR, J. *Cyclictest*. 2025. <<https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start>>. Última modificação em 19 de janeiro de 2025. Citado na página 8.
- YANG, D.; DAI, W. A lightweight container design for microservice-based industrial edge applications. In: IEEE. *2022 IEEE 17th Conference on Industrial Electronics and Applications (ICIEA)*. [S.l.], 2022. p. 858–863. Citado na página 1.