Capítulo

3

DinD-Bench: Impacto de Contêineres Docker em Docker para a Programação Paralela[1]

Claudio Schepke - claudioschepke@unipampa.edu.br[2]
Felipe Bedinotto Fava - felipefava.aluno@unipampa.edu.br[3]
Diego Luis Kreutz - diegokreutz@unipampa.edu.br[4]

[1]Este capítulo foi parcialmente financiado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq (Processo número 407827/2023-4), Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul - FAPERGS (Edital PqG 07/2021 - Projeto Nº 21/2551-0002055-5) e Programa Hackers do Bem da Rede Nacional de Ensino e Pesquisa - RNP (GT - Malware DataLab)

[2]Claudio Schepke possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2005) e mestrado (2007) e doutorado (2012) em Computação pela Universidade Federal do Rio Grande do Sul, sendo este feito na modalidade sanduíche na Technische Universität Berlin, Alemanha (2010-2011). É professor associado da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete/RS desde 2012. Tem experiência na área de Ciência da Computação, com ênfase em Processamento de Alto Desempenho, atuando principalmente nos seguintes temas: interfaces de programação paralela, aplicações científicas e computação em nuvem.

[3]Felipe Bedinotto Fava é graduado em Ciência da Computação pela Universidade Federal do Pampa (UNIPAMPA), Campus Alegrete (2024) e técnico em Informática para Internet pelo Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense (IFSUL), Campus Santana do Livramento (2017). Atualmente é mestrando do Programa de Pós-Graduação em Engenharia de Software (PPGES) da UNIPAMPA. Possui experiência profissional no ramo do desenvolvimento de aplicações para múltiplas plataformas com ênfase em dispositivos móveis. Apresenta afinidade e interesse nas áreas de Computação Ubíqua, Computação em Nuvem, Computação Paralela, Computação de Alto Desempenho e Arquitetura de Computadores.

[4]Diego Kreutz possui doutorado pela Universidade de Luxemburgo (Luxemburgo) e mestrado em Informática, mestrado em Engenharia de Produção e graduação em Ciência da Computação pela UFSM. Realizou estágio de pós-doutorado na Universidade de Monash (Austrália) e possui experiências internacionais em pesquisa, desenvolvimento e inovação no LaSIGE/ULisboa/Portugal e CritiX/UNI/Luxemburgo. Durante essas experiências, atuou em projetos internacionais com pesquisadores e especialistas de instituições estrangeiras como CMU/USA, UPorto/Portugal, UCoimbra/Portugal, Portugal Telecom/Portugal, TUM/Alemanha, Infineon/Alemanha, Ether Trust/França, UPMC/França e SnT/Luxemburgo. Atualmente é professor associado na UNIPAMPA. Possui experiência na área de Ciência da Computação, com ênfase em Segurança Cibernética (ou Cibersegurança), Inteligência Artificial, Sistemas Distribuídos, Redes de Computadores, Tolerância a Falhas e Intrusões e Desenvolvimento de Software.

Resumo

Contêineres oferecem versatilidade permitindo o desenvolvimento de sistemas com arquiteturas monolítica e de microsserviços. Na primeira abordagem, um sistema inteiro opera dentro de um único contêiner, enquanto no segundo, um ou mais processos são encapsulados dentro de contêineres. Existem dois modelos predominantes para implementação baseada em contêineres : o modelo mestre-escravo e o modelo de contêiner aninhado. Neste minicurso introduzimos a ferramenta denominada DinD-Bench, publicamente disponível, para sistematizar a avaliação de desempenho entre o Docker padrão e o Docker dentro do Docker (DinD ou contêineres aninhados). A DinD-Bench incorpora componentes e códigos projetados especificamente para instanciar e executar benchmarks conhecidos, como SysBench, Stress, IOzone e iPerf, no Docker e sua variante de contêiner aninhado. Os recursos da DinD-Bench viabilizam uma execução automatizada, sistemática e reprodutível, podendo ser aplicada em outras arquiteturas computacionais e adaptada para outros sistemas operacionais, além de prover a coleta de valores de performance e permitir a incorporação de outros benchmarks. Para demonstrar a aplicação prática dos recursos introduzidos, apresentamos uma avaliação de desempenho considerando duas distribuições GNU Linux (Alpine e Debian) para os contêineres Docker, diversas cargas de trabalho e diferentes plataformas de computação, como nós da nuvem da Google e máquinas locais. Através das avaliações experimentais realizadas, podemos concluir que (a) os contêineres aninhados exigem até 7 segundos para serem inicializados, enquanto os contêineres padrão Docker exigem menos de 0,5 segundos para os sistemas operacionais Debian e Alpine; (b) Debian exibe desempenho superior de CPU e menor latência comparado ao Alpine; (c) Imagens Docker baseadas em Debian podem exigir 20% (ou mais) de tamanho de memória do que aquelas construídas em Alpine; e (d) As operações de disco e rede não tiveram diferenças significativas entre contêineres Docker aninhados e Docker. Além disso, vale ressaltar que algumas das disparidades decorrentes da combinação de contêineres e sistemas de hospedagem parecem ser influenciadas pela pilha de software em uso, incluindo diferentes versões de kernel, bibliotecas e outros pacotes de software essenciais.

3.1. Introdução

Contêineres revolucionaram o modo como aplicações podem ser instanciadas. Um contêiner é extremamente leve, provendo velocidade e eficiência de processamento, e garante isolamento e segurança na computação, sem afetar o software de gerenciamento de base de uma determinada arquitetura. Assim, por exemplo, um único servidor x86 pode hospedar facilmente centenas de contêineres, sendo, neste caso, a memória muitas vezes a principal restrição para o número de contêineres executados simultaneamente. Notavelmente, contêineres têm tempos de inicialização rápidos, demorando normalmente entre 1 a 2 segundos para serem instanciados.

Duas razões fundamentais destacam-se para o ressurgimento dos contêineres do ponto de vista técnico. Em primeiro lugar, as melhorias no suporte de *namespaces* e de *cgroups* do *kernel* Linux, desempenharam um papel fundamental no isolamento e limitação de recursos. Em segundo lugar, Docker, uma implementação específica de contêineres, transformou o cenário. Docker não apenas introduziu um formato de encapsulamento

atraente, mas também forneceu ferramentas indispensáveis e promoveu um ecossistema diversificado, tornando-se fundamental para este renascimento dos contêineres.

Uma alternativa para o desenvolvimento que tem sido explorada é o uso de contêineres aninhados, também conhecidos como *Docker in Docker* (DinD) ou *Docker Inside Docker*. DinD envolve a execução do Docker dentro de um contêiner Docker. Em vez de interagir com o *daemon* Docker do *host*, um novo mecanismo Docker é gerado dentro de um contêiner, fornecendo um ambiente isolado para gerenciar contêineres e imagens. DinD oferece uma solução elegante para criar contêineres Docker reprodutíveis e confiáveis. Ao lançar o Docker dentro do Docker, os desenvolvedores e administradores de sistema podem agilizar seus fluxos de trabalho de desenvolvimento, aumentar a segurança e simplificar seus pipelines de *Continuous Integration/Continuous Deployment* (CI/CD).

Em relação ao desempenho, pelo menos cinco pontos-chave podem ser destacados nos ambientes DinD para o ano de 2015 Amaral et al. (2015). Primeiro, as tarefas com uso intensivo de CPU em contêineres não apresentam diferença significativa de desempenho em comparação com configurações *bare-metal*. Em segundo lugar, os contêineres regulares são os mais rápidos, seguidos pelos contêineres aninhados e pelas máquinas virtuais, durante a criação de uma instância. Terceiro, a criação de contêineres aninhados envolve mais sobrecarga, mas ainda é mais rápida que as máquinas virtuais. Camadas extras de abstração impõem a execução de mais código, a fim de garantir a confiabilidade necessária, o que impacta na performance da aplicação. Quarto, a criação de vários pais com um único filho leva mais tempo devido a gargalos na inicialização do contêiner pai. Por último, configurações variadas de rede para contêineres aninhados (*host network*, *Linux bridge* e *Open vSwitch*) não apresentam impacto significativo no desempenho quando a rede física torna-se o gargalo.

No entanto, o panorama da tecnologia Docker evoluiu substancialmente de 2015 à 2024. Além disso, é importante destacar que avaliações passadas (e.g., (AMARAL et al., 2015)) concentraram-se exclusivamente em tarefas com uso intensivo de CPU, isto é, não exploraram cargas de trabalho levando em consideração quesitos como desempenho da memória e E/S de disco e de rede. Diferentemente de outras avaliações e soluções existentes, a DinD-Bench (FAVA et al., 2023, 2024) leva em consideração também os impactos potenciais de diferentes distribuições GNU/Linux e explora diversas configurações de servidores em ambientes de nuvem.

Considerando esses fatores, este capítulo introduz a DinD-Bench e apresenta uma avaliação de desempenho atualizada e que permite uma compreensão mais abrangente do comportamento do desempenho do Docker em contextos contemporâneos. Para além da DinD-Bench, é apresentada também uma metodologia de avaliação de desempenho de DinD e Docker em diferentes tipos de hardware e distribuições GNU/Linux usando *benchmarks* estabelecidos como SysBench[5], Stress[6], IOzone[7] e iPerf[8] (DIAZ et al., 2014; BACHIEGA et al., 2020; POKHARANA; GUPTA, 2023). O impacto de DinD foi considerado em plataformas de infraestrutura como serviço (IaaS) em nuvem, como *Google*

[5]<https://github.com/akopytov/sysbench>
[6]<https://github.com/resurrecting-open-source-projects/stress>
[7]<https://www.iozone.org/>
[8]<https://iperf.fr>

Compute Engine (GCE) e em ambientes de servidores locais tradicionais disponíveis.

O restante do capítulo está organizado da seguinte forma. A Seção 3.2 introduz Docker e DinD no cenário atual da computação. A DinD-Bench é apresentada na Seção 3.3, incluindo aspectos de arquitetura, implementação, funcionamento e utilização. A Seção 3.4 discrimina a metodologia dos experimentos, caracterizando os *benchmarks*, métricas e ambientes de execução. A Seção 3.5 mostra o resultados experimentais para CPU, memória, disco e rede, considerando quatro *benchmarks* distintos em cinco máquinas com diferentes configurações de hardware. Finalmente, as considerações finais são apresentadas na Seção 3.6.

3.2. Docker

Docker emergiu como uma tecnologia essencial em ambientes de desenvolvimento e produção, especialmente para implementação e implantação de arquiteturas baseadas em microsserviços (BOGNER et al., 2019; TAPIA et al., 2020; Karabey Aksakalli et al., 2021). No entanto, sua prevalência é enfatizada pela ampla adoção em diversos domínios e aplicações. Por exemplo, o Docker tornou-se uma ferramenta crucial na engenharia de software moderna, facilitando a instrumentação de testes automatizados e a implementação de pipelines de *Continuous Integration/Continuous Deployment* (CI/CD). Isto demonstra a sua versatilidade e eficácia no atendimento de demandas complexas do desenvolvimento de software contemporâneo.

Em termos de programação paralela, Docker oferece um desempenho similar às execuções em ambiente físico e superior ao uso de máquinas virtuais. Por outro lado, o grande tempo de inicialização de microsserviços, não é algo tolerável para aplicações com tempo de execução na ordem de poucos segundos. A grande vantagem está no fato de que contêineres permitem versatilidade de gerenciamento, o que provê uma forma de avaliar a instanciação de um grande número de nós, por exemplo. Uma outra vantagem é que existem aplicações com muitas dependências de software, como os modelos de previsão do tempo. O uso de uma versão distinta de um pacote ou a versão diferente de uma biblioteca pode gerar erros de compilação. Conteinerização pode ser a solução para que esses tipos de erros não ocorram.

Existem dois modelos predominantes para implementação baseada em contêiner Docker dentro da estrutura de microsserviços: o modelo mestre-escravo e o modelo de contêiner aninhado (DinD) (AMARAL et al., 2015). O modelo mestre-escravo compreende um contêiner mestre orquestrando outros como escravos, onde esses operam os processos da aplicação. O mestre é responsável por monitorar os contêineres subordinados e facilitar sua comunicação. Em contraste, a abordagem de contêiner aninhado envolve a criação hierárquica de contêineres subordinados (filhos) dentro de um contêiner principal (pai). O contêiner pai pode ser agnóstico e simplesmente existir, enquanto os filhos executam os processos do aplicativo, confinados dentro dos limites definidos pelo pai.

A abordagem DinD simplifica o gerenciamento em arquiteturas de microsserviços. Isso reduz o número de etapas de reprodução de experimentos e isola os ambientes de desenvolvimento e teste, aumentando a segurança e o isolamento, simplificado implementação de pipelines de CI/CD e gerenciamento de recursos em cenários de multi-locação, onde múltiplas equipes ou usuários exigem ambientes isolados em infraestrutura com-

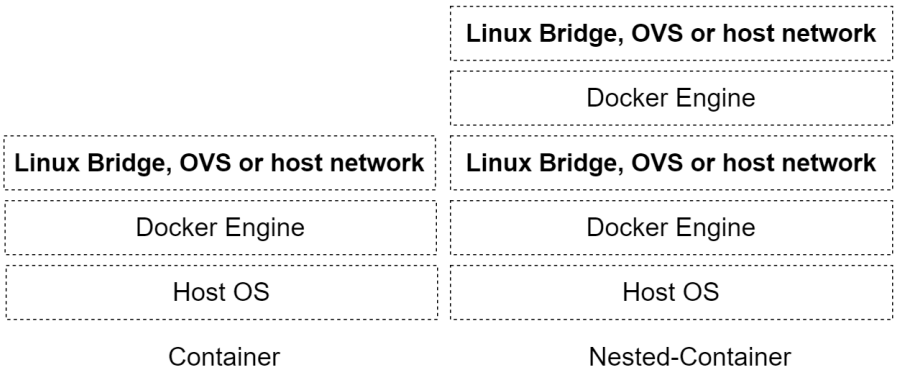

Figura 3.1. Duas camadas de *daemon* Docker (adaptado de (AMARAL et al., 2015), Figura 2)

partilhada (KUSHWAH, 2024; PETAZZONI, 2024). DinD facilita a comunicação entre processos (IPC), garante o compartilhamento de destino e permite o compartilhamento de memória, disco e recursos de rede. No entanto, esta abordagem pode introduzir custos adicionais devido às duas camadas do *daemon* Docker, conforme ilustrado na Figura 3.1.

Com relação aos aspectos práticos, a Figura 3.2 ilustra a interação entre os componentes Docker (*Docker Workflow*). O Dockerfile é considerado o arquivo padrão que especifica como a imagem Docker irá ser construída e qual será o seu conteúdo. Um contêiner Docker representa uma instância de uma imagem Docker em execução. O Docker Hub pode ser utilizado para armazenar de maneira centralizada e organizada as imagens Docker, que serão utilizadas tanto no Docker padrão quanto no DinD.

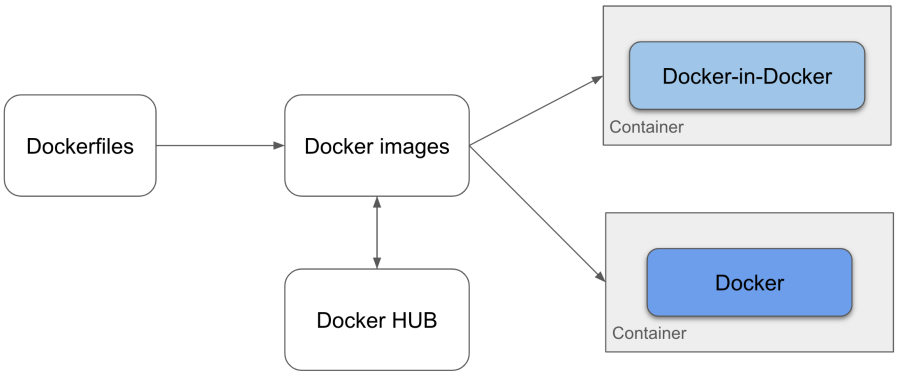

Figura 3.2. Relação entre Dockerfile, imagens, contêineres e hub docker

No contexto do Docker, o fluxo de trabalho pode ser definido através de 7 etapas (OZOR, 2023), como discriminado a seguir.

1. **Desenvolvimento e construção da aplicação:** O desenvolvimento de uma aplicação pode ser realizado em qualquer linguagem de programação. Posteriormente, são definidos o conjunto de instruções para construir uma imagem Docker, tipicamente em um arquivo de texto estruturado conhecido como Dockerfile. O Doc-

kerfile define a imagem base (e.g., Debian 12.0, Ubuntu 22.04), as dependências necessárias, o caminho do código da aplicação que será copiado para dentro da imagem e os comandos a serem executados automaticamente quando o contêiner for iniciado.

2. **Construção de uma imagem Docker:** Através da CLI (*Command Line Interface*) do Docker, ou de uma ferramenta de construção como *Docker Compose*, o usuário pode construir uma imagem Docker baseada na especificação contida no Dockerfile. A imagem Docker resultante é um pacote de software independente, encapsulando tudo o que é essencial para executar a aplicação, o que inclui o código-fonte/executável, o ambiente de execução e quaisquer dependências necessárias, como ferramentas de sistema, bibliotecas e configurações específicas.

3. **Execução de contêineres Docker:** A partir de uma imagem previamente criada, o usuário pode instanciar um contêiner, via CLI ou outras ferramentas, utilizando a *Engine* do Docker. Contêineres são instâncias de imagens Docker que podem ser iniciadas, interrompidas e gerenciadas de forma independente. Os contêineres garantem funcionalidade uniforme em diferentes ambientes, ao isolar o software do seu entorno. Pode-se executar contêineres localmente em uma máquina de desenvolvimento ou implantá-los em um ambiente de produção.

4. **Gerenciamento de contêineres Docker:** O *Docker Engine* fornece um conjunto de recursos e comandos para gerenciar contêineres em execução. Pode-se visualizar contêineres em execução, interrompê-los, iniciar contêineres interrompidos e remover aqueles que não forem mais necessários.

5. **Utilização do *Docker Compose* para aplicações em vários contêineres:** *Docker Compose* é uma ferramenta para definir e executar aplicativos Docker de vários contêineres. Os serviços são definidos através de arquivos de configuração YAML, onde são especificados recursos de rede e volumes necessários para a aplicação / serviço. É possível iniciar e finalizar diversos contêineres simultaneamente através do *Docker Compose*.

6. **Publicação e distribuição imagens Docker:** As imagens do Docker podem ser publicadas em repositórios privados ou públicos, como é o caso do *Docker Hub*. A publicação das imagens facilita o gerenciamento e a implantação e também potencializa a re-utilização por outras pessoas em outros ambientes e contextos.

7. **Atualização e iteração:** À medida que novas versões da aplicação continuam a ser desenvolvidas, pode-se iterar na imagem do Docker, atualizando o Dockerfile e reconstruindo a imagem. Isso permite distribuir e implantar facilmente novas versões do seu aplicativo.

3.3. DinD-Bench

A Figura 3.3 apresenta a visão geral da DinD-Bench. Ela é composta por cinco componentes principais: (1) módulo principal; (2) conjunto de *benchmarks*; (3) repositório de dados; (4) módulo de análise de dados; e (5) recursos gerais. A função do módulo

principal é permitir que o usuário selecione o *benchmark* que será executado. Um aspecto importante do módulo principal é o fato de ele estar preparado para incorporar novos *benchmarks*. A inclusão de um novo *benchmark* requer apenas poucas ações do usuário, como inclusão de um novo diretório e a configuração do arquivo padrão de inicialização e execução do *benchmark*. A partir desse ponto, o *benchmark* passa a estar automaticamente disponível na DinD-Bench.

O arquivo de configuração e execução do *benchmark* está programado para receber entradas (i.e., parâmetros de execução) e gerar saídas (e.g., arquivos com as estatísticas de saída) que estarão disponíveis no repositório de dados ao final da execução. Na configuração de um novo *benchmark*, o usuário necessita também realizar ajustes de parâmetros de entrada e saída para a execução e geração de dados de saídas compatíveis com a estrutura e organização da DinD-Bench.

Ao final da execução do(s) *benchmark(s)*, o usuário pode utilizar o módulo de análise de dados para gerar estatísticas e gráficos da(s) execução(ões). Para cada novo *benchmark*, o usuário pode também incluir novos recursos de processamento e análise de dados no respectivo módulo. Isto torna a solução flexível e ajustável a diferentes ferramentas e contextos.

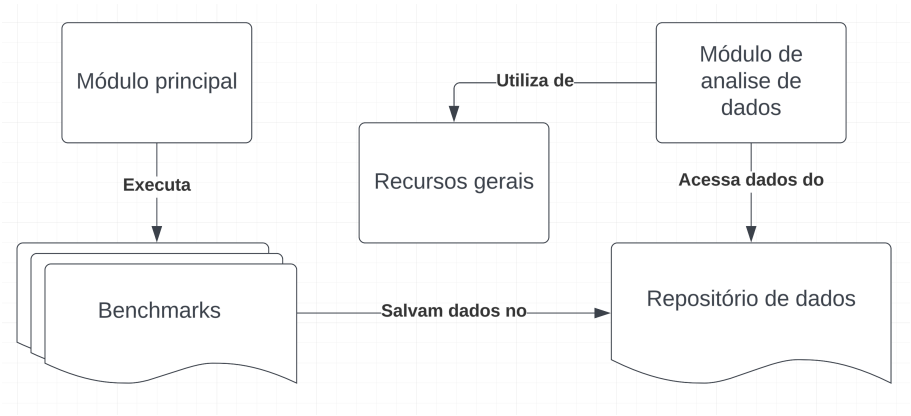

Figura 3.3. Arquitetura organizacional DinD-Bench

A Figura 3.4 ilustra o fluxo de execução da DinD-Bench. O usuário inicializa o módulo principal escolhendo o(s) *benchmark(s)* que serão executados e a instância que será executada (i.e., Docker ou DinD). No momento que o módulo principal inicializa o(s) *benchmark(s)* especificado(s) pelo usuário, é executada a instância definida pelo usuário, ou seja, Docker ou DinD. No primeiro caso é instanciado um contêiner Docker através do comando `docker run`. Já no segundo caso são preparadas as configurações que serão repassadas para o contêiner aninhado. A execução do DinD é realizada através da execução do comando `docker run –privileged`.

Para entender os detalhes da DinD-Bench, é importante compreender o que são *nested* contêineres e o que os difere de contêineres normais. Resumidamente, quando um contêiner é executado dentro de outro, cunha-se o nome de *nested* contêiner ou contêiner aninhado. Uma das estratégias mais conhecidas e utilizadas é a Docker-in-Docker (DinD). Na DinD é instanciado um contêiner Docker *daemon* dentro de outro contêiner

Docker. Para executar contêineres aninhados, é necessário utilizar a *flag* `-privileged`, que irá elevar os privilégios do contêiner para viabilizar o acesso a recursos da máquina hospedeira (e.g., diretório de dados, dispositivos de E/S).

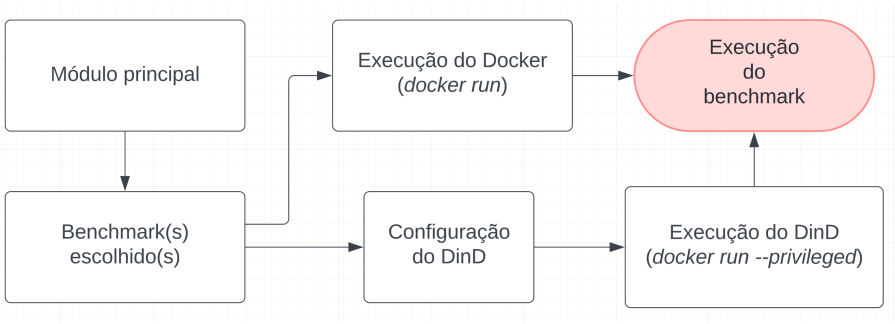

Figura 3.4. Fluxo de execução da DinD-Bench

A Figura 3.5 ilustra visualmente a organização dos contêineres quando utilizada a abordagem DinD. Na máquina hospedeira (*host*) é executado um contêiner Docker, dentro do qual é instanciado um segundo contêiner Docker, identificado na figura por *Docker-in-Docker*. As setas presentes na imagem indicam onde são executados os comandos passados pelo usuário para cada um dos ambientes.

A instanciação da DinD-Bench inicia com a execução de um comando na máquina hospedeira (física ou virtual). Na sequência, é executado um comando para a criação do contêiner aninhado dentro da primeira instância Docker. Finalmente, é executado o *benchmark* selecionado dentro do contêiner aninhado (DinD).

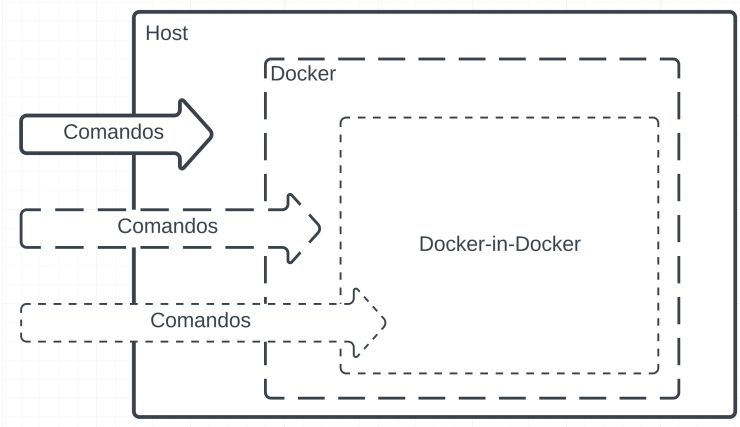

Figura 3.5. Ambiente e comandos no contexto da DinD-Bench

Observando a arquitetura e o fluxo de funcionamento, a DinD-Bench pode ser considerada também um automatizador da execução de aplicações dentro de contêineres normais ou aninhados (i.e., DinD). Na prática, a DinD-Bench é agnóstica às aplicações que são executadas dentro dos contêineres. Consequentemente, o usuário pode adaptar a solução para executar quaisquer outros tipos de aplicações.

O código da DinD-Bench, bem como dados coletados nas medições de desempenho apresentadas na Seção 3.5, pode ser encontrado no repositório Github[9]. Os *benchmarks* disponíveis estão configurados para execução em contêineres Docker baseados em Debian e Alpine. É importante destacar também que novas distribuições GNU/Linux podem ser facilmente acrescentadas para ampliar ainda mais as análises e avaliações. Atualmente, a avaliação de desempenho produzida pelos *benchmarks* já incorporados à DinD-Bench permitem a avaliação sistemática de CPU, memória, latência e taxa de transferência de E/S (para rede e disco).

3.4. Metodologia

Uma visão geral da metodologia é apresentada na Figura 3.6. Para avaliar o desempenho de quatro componentes cruciais do computador (CPU, memória, disco e rede) em dois ambientes de contêiner (Docker e Docker-in-Docker) em duas distribuições GNU/Linux (Debian 12.1 e Alpine 3.18), conduziu-se a execução de quatro *benchmarks* em cinco máquinas com configurações distintas. Cada *benchmark* é executado 20 vezes, resultando em um total de 1.600 execuções. A Tabela 3.1 resume as principais configurações. Para uma descrição detalhada e configurações dos experimentos, disponibilizamos as informações publicamente acessíveis no GitHub (FAVA et al., 2023).

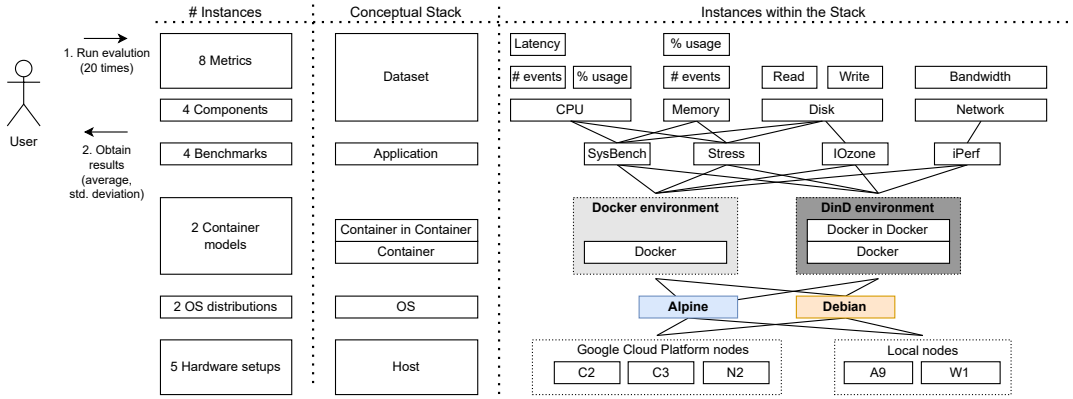

Figura 3.6. Metodologia aplicada (conforme (FAVA et al., 2024), Figura 2)

Conforme mostra a Tabela 3.1, o ambiente de execução compreende três nós do Google Cloud Platform (GCP): C3, N2 e E2, e dois nós locais, A9 (Intel i7) e W1 (AMD Ryzen). Especificamente, os nós GCP C3 (tipo c3-standard-4), N2 (tipo n2-standard-2) e E2 (tipo e2-medium) apresentam 2, 1 e 1 núcleos/threads, respectivamente. Em contraste, os nós locais A9 e W1 estão equipados com 4 e 8 núcleos físicos, respectivamente. Vale ressaltar que os nomes das máquinas locais não remetem nenhum padrão especifico de nomenclatura de nenhum ambiente de *cloud*.

Os *benchmarks* selecionados são amplamente reconhecidos nas comunidades de profissionais e pesquisadores para avaliar a performance de aspectos de hardware (UpCloud, 2018; QIAN, 2019; Dell Technologies, 2021; DIAZ et al., 2014; XAVIER et al., 2016; BACHIEGA et al., 2020; POKHARANA; GUPTA, 2023). Esses *benchmarks* são

[9]<https://github.com/DinDperf/DinD-Bench>

Tabela 3.1. Configurações dos benchmarks, contêineres, sistemas computacionais e computadores ((FAVA et al., 2024), Tabela 1))

Item	Descrição
SysBench	Emprego de uma sequência padrão de testes, incluindo threads, memória, CPU e FILEIO.
Stress	Um teste com tempo limite de 100s para cada parâmetro de CPU, HDD, E/S e VM.
IOzone	Utilização de 1 GB de dados para leitura e gravação em blocos de 4 KB.
iPerf	Uso dos parâmetros cliente-servidor padrões (locais) para medir a taxa de transferência e a latência.
Contêiner	Docker e Docker in Docker (DinD)
Sist. Oper.	Alpine 3.18 e Debian 12.1
C2 (GCP)	Debian 12, Docker Engine 24.0.7, Intel Xeon 3.10GHz, 32GB NVMe de Disco, 16GiB DIMM RAM
C3 (GCP)	Debian 12, Docker Engine 24.0.7, Intel Xeon Platinum 8481C, 32GB NVMe de Disco, 16GiB DIMM RAM
N2 (GCP)	Debian 12, Docker Engine 24.0.7, Intel Xeon 2.80GHz, 32GB PersistentDisk, 8GiB DIMM RAM
A9 (Local)	Debian 11, Docker Engine 24.0.7, Intel i7-9700 3.00GHz, 1TB WDC WD10EZEX-08W, 16GiB DIMM DDR4
W1 (Local)	Ubuntu 22.04.3 LTS, Docker Engine 24.0.5, AMD Ryzen 7 5800X, 512GB NVMe de Disco, 64GiB DIMM DDR4

abrangentes para mensurar processamento, memória e desempenho de E/S, abrangendo operações de rede e disco. A seguir, são descritas as especificações das ferramentas de referência escolhidas para a avaliação.

- **SysBench**: é uma ferramenta versátil de *benchmark* amplamente utilizada em sistemas do tipo Unix. SysBench avalia o desempenho do sistema em áreas como CPU, memória, threads, E/S de disco e bancos de dados (KOPYTOV, 2023). Esta ferramenta auxilia gerentes de sistema e desenvolvedores a testar e analisar o desempenho de sistemas e aplicações em diversos cenários.

- **Stress**: é uma ferramenta projetada para aplicar uma variedade de testes em um sistema. Stress pode avaliar CPU, memória, E/S (sincronização) ou estresse de disco (WATERLAND, 2023). A ferramenta é comumente utilizada por programadores de núcleo do sistema, permitindo-lhes avaliar a escalabilidade do sistema e examinar minuciosamente os recursos de desempenho percebidos. Além disso, a ferramenta ajuda os programadores de sistemas a revelar classes específicas de falhas que muitas vezes surgem apenas sob cargas pesadas, oferecendo percepções críticas sobre o desempenho do sistema sob condições de estresse.

- **IOzone**: é uma ferramenta essencial no domínio de avaliação de sistemas de arquivos (IOZONE, 2023). IOzone é um utilitário de *benchmark* versátil que gera e

avalia uma diversidade de operações de arquivos. Ele foi portado para várias máquinas e é compatível com vários sistemas operacionais. IOzone é inestimável para conduzir análises abrangentes de sistemas de arquivos em diversas plataformas de computador. O *benchmark* avalia rigorosamente o desempenho de E/S de arquivos em um espectro de operações, incluindo leitura, gravação, releitura, reescrita, leitura reversa, leitura circular, leitura aleatória, utilizando as funções *fread*, *fwrite*, *pread*, *aio_read* e *aio_write* e *mmap*.

- **iPerf**: é uma ferramenta de código aberto frequentemente utilizada para medir o desempenho de redes de computadores baseadas em IP (IPERF.FR, 2024). iPerf é eficaz para avaliações essenciais, incluindo testes de largura de banda, latência e perda de pacotes. A ferramenta designa um dispositivo como servidor, que escuta ativamente as conexões, operando em um modelo cliente-servidor. A segunda instância do iPerf inicia as conexões, funcionando como cliente, para realizar testes de desempenho.

3.5. Resultados Experimentais

Nesta seção são discutidos os principais resultados obtidos relativos a CPU, memória, disco e componentes de rede, em suas respectivas subseções. São apresentados uma série de gráficos obtidos pela execução dos *benchmarks*, para cada item considerado. Nos gráficos são apresentados resultados para as diferentes configurações de hardware, as duas distribuições GNU/Linux utilizadas nos experimentos (Alpine preenchido em azul e Debian preenchido em laranja) e os dois modelos de contêiner (DinD preenchido com cores mais escuras e Docker preenchido com cores mais claras).

3.5.1. CPU

Na Figura 3.7 são apresentados os resultados de uso da CPU pelo *benchmark* SysBench. O número de eventos e a latência estão sendo levados em consideração. Observando a tecnologia da CPU e a frequência de operação (consulte Tabela 3.1), percebe-se que o número de eventos da CPU se correlacionam diretamente com a capacidade da CPU. Em particular, o sistema W1 demonstra o mais alto desempenho, enquanto N2 apresenta o mais baixo. Como esperado, há uma relação quase inversa, ou seja, a melhor CPU apresenta a menor latência.

Os resultados também indicam que os contêineres Docker baseados em Debian superam consistentemente os resultados das configurações equivalentes baseados em Alpine, obtendo menor latência de CPU. Notavelmente, estudos recentes também destacaram disparidades de desempenho entre imagens Docker construídas em diferentes distribuições Linux para vários sistemas, como bancos de dados (por exemplo, Cassandra, Mongo e MySQL) e servidores web (por exemplo, Nginx) (IBRAHIM; SAYAGH; HASSAN, 2020). Uma distinção importante entre essas imagens Docker está no número variável de bibliotecas instaladas (por exemplo, de 13 a 119) em diferentes distribuições Linux para o mesmo sistema (por exemplo, MySQL). Além disso, o número de eventos e a latência da CPU mostrados na Figura 3.7 indicam que a influência do DinD sobre o Docker é insignificante para ambos os sistemas operacionais.

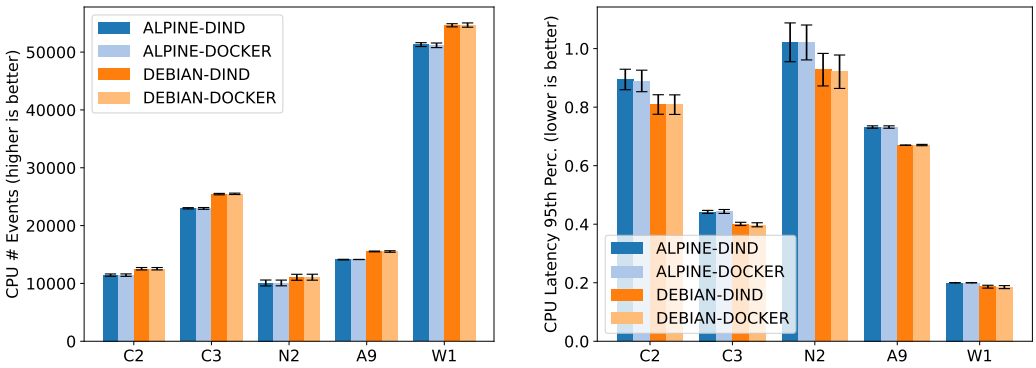

Figura 3.7. Estatísticas de CPU usando SysBench incluindo número de eventos e valores de latência (FAVA et al., 2024), Figura 3

Na Figura 3.8, são apresentados os dados sobre o número de eventos e porcentagem dos resultados de SysBench para o subsistema de testes usando threads. Notavelmente, as instâncias do Docker exibem uma contagem maior de eventos com latência menor. A maior latência do DinD permanece consistente em diversos sistemas de hospedagem. No entanto, é importante observar que essa latência varia entre as máquinas. Por exemplo, em cenários como W1, a latência DinD pode ultrapassar a latência do Docker em mais de 10%.

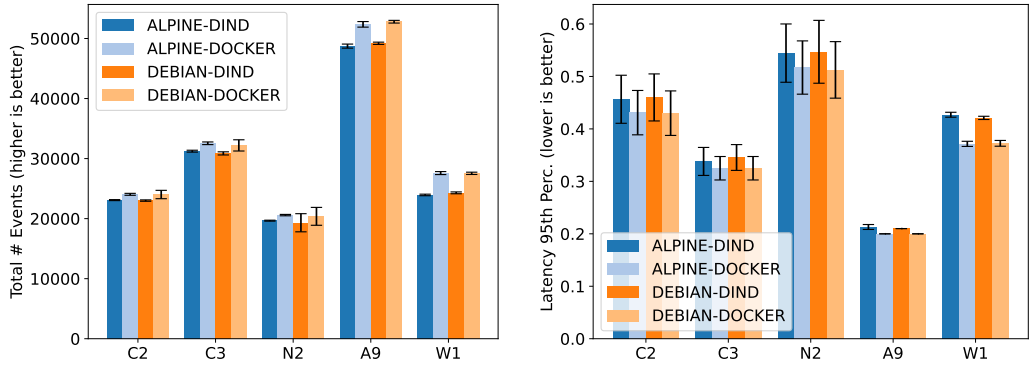

Figura 3.8. Estatísticas de threads usando SysBench, incluindo o número de eventos e porcentagem (FAVA et al., 2024), Figura 4

Ao examinar o comportamento de execução de Stress em CPU para DinD e Docker em 1024 amostras (segundos), conforme ilustrado na Figura 3.9 e na Figura 3.10, torna-se evidente uma notável latência de inicialização (*bootstrap*) em instâncias DinD. Por exemplo, embora o *benchmark* Stress inicie sua execução imediatamente no Alpine Docker, isso pode levar mais de 20 segundos em um sistema de hospedagem C2 ao usar DinD. Esse padrão de latência persiste em diferentes *benchmarks* e ambientes de execução, bem como também é observado consistentemente tanto em instâncias Debian como Alpine. Notavelmente, a latência de *bootstrap* de DinD pode variar significativamente de uma máquina para outra. O sistema N2 exige mais tempo de inicialização. Já W1 apresenta desempenho mais rápido.

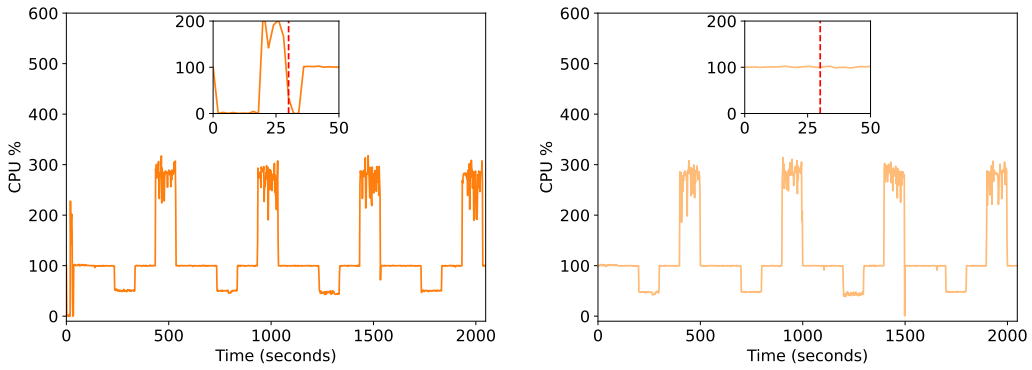

Figura 3.9. Comparação de DinD e Docker usando Stress para Debian no sistema C3

Para investigar melhor a latência de *bootstrap*, mostrada na Figura 3.9 e na Figura 3.10, foram analisados os tempos de inicialização e desligamento dos contêineres DinD, conforme valores apresentados na Figura 3.11. Enquanto os contêineres Docker iniciam em menos de 500 ms, DinD leva em média até 6 segundos para iniciar. Isso significa que DinD pode ser 12 vezes mais lento que os contêineres Docker para se tornar operacional para a execução de aplicações.

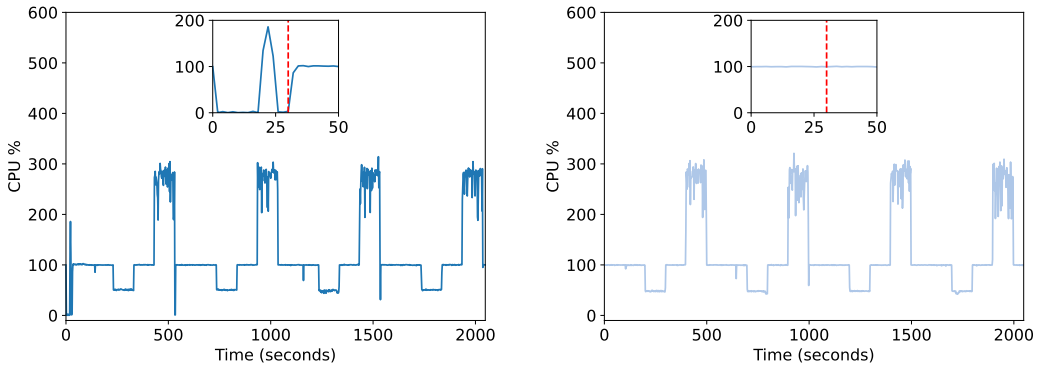

Figura 3.10. Comparação de DinD e Docker usando Stress para Alpine no sistema C3

De forma similar, embora os contêineres Docker terminam em menos de 100 ms, DinD pode levar mais de 1,5 segundos para parar completamente. Também é importante notar que não há diferença significativa nos tempos de inicialização e desligamento entre contêineres DinD baseados em Alpine e Debian. Curiosamente, o sistema A9 exibe um tempo de inicialização e desligamento um pouco mais lento, apesar de sua CPU Intel i7-9700 com clock de 3GHz. Atribui-se o alto desvio padrão à natureza não determinística inerente aos tempos de paralisação dos contêineres. No entanto, é importante observar que a pilha de software em uso pode potencialmente influenciar essa variação. Ao contrário dos outros sistemas de hospedagem que operam em Debian 12 ou Ubuntu 22.04, o sistema A9 roda em Debian 11, conforme indicado na Tabela 3.1. Esta distinção nos sistemas operacionais pode contribuir para as disparidades observadas no desempenho.

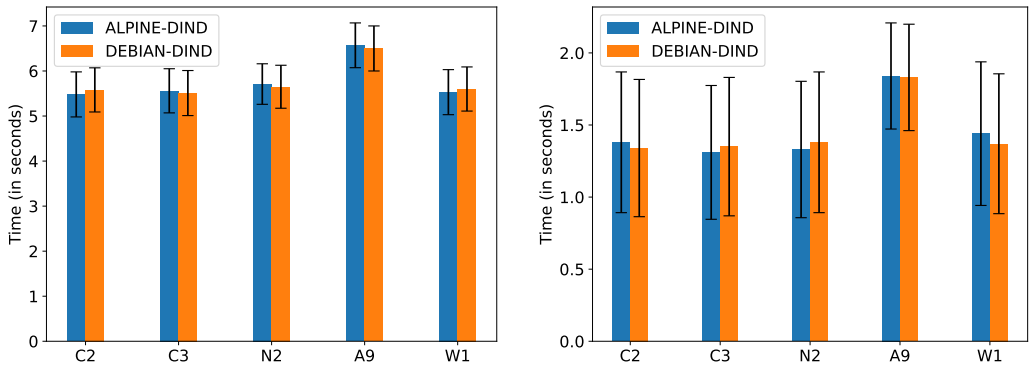

Figura 3.11. Tempo necessário para inicializar e parar contêineres DinD (FAVA et al., 2024), Figura 6

3.5.2. Memória

Na Figura 3.5.2, são apresentadas as contagens de eventos de memória para SysBench e a utilização de memória (por amostragem) para Stress. Os resultados tornam evidente que o número total de eventos de memória permanece consistente nas configurações DinD e Docker, os quais empregam a mesma distribuição Linux. Atribui-se essa consistência à metodologia uniforme de coleta de dados do SysBench em ambos os ambientes. No entanto, há uma disparidade notável nas contagens de eventos de memória, ao comparar as distribuições Debian e Alpine. Em particular, a imagem Docker baseada em Debian incorre em uma sobrecarga significativa, ultrapassando 20% do tamanho da memória em todos os ambientes de execução. Essa discrepância ressalta o impacto da distribuição Linux subjacente na geração de eventos de memória.

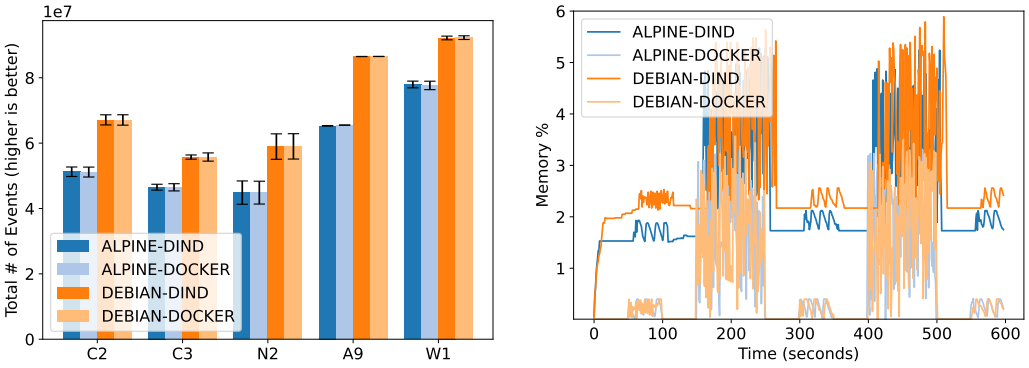

Figura 3.12. Consumo de memória de SysBench e Stress (FAVA et al., 2024), Figura 7

Além disso, pode-se fazer pelo menos três observações a partir da Figura 3.5.2. Em primeiro lugar, é evidente que o consumo geral de memória de DinD é significativamente maior do que o Docker em ambas as distribuições Linux. Esta medida abrange o consumo de memória inerente do *benchmark*, como a sobrecarga de memória adicional introduzida pelo DinD. Vale a pena enfatizar que as amostras de utilização de memória

são coletadas em tempo real, fornecido por *docker stats* para um contêiner específico em execução.

Em segundo lugar, surge uma diferença visível entre DinD baseado em Debian (representado pela linha amarela) e DinD baseado em Alpine (representado pela linha azul). Debian DinD exibe consistentemente um consumo de memória significativamente maior do que seu equivalente Alpine (no gráfico, é fácil visualizar considerando o par DinD). Mais uma vez, essa variação ressalta que as imagens Docker baseadas no Debian podem exigir 20% a mais memória do que aquelas construídas no Alpine.

Além disso, a Figura 3.5.2 revela um padrão cíclico no comportamento do *benchmark*. Esta ocorrência cíclica é comum em todos os programas ao capturar estatísticas de utilização de memória usando `docker stats`. Atribui-se às disparidades ilustradas aqui às diferenças nas imagens Docker enraizadas em distribuições Linux distintas (IBRAHIM; SAYAGH; HASSAN, 2020). Dependendo das bibliotecas específicas da distribuição usadas para suportar e executar os *benchmarks*, o consumo de memória pode variar significativamente, levando às discrepâncias observadas no consumo de memória.

3.5.3. Disco

Na Figura 3.13 são mostrados os resultados do *benchmark* IOzone em GB/s. Os testes incluíram as execuções nas opções *read*, *write*, *reread* e *rewrite*. Observa-se diferenças mínimas entre as quatro abordagens, tanto para DinD quanto para Docker usando Alpine e Debian, em nossos sistemas de hospedagem. Geralmente, as operações de releitura apresentam uma ligeira vantagem nos contêineres Docker em comparação ao DinD. Em ambientes específicos, como A9 e W1, as distinções entre contêineres Docker e DinD podem variar para *benchmarks* ou aplicativos com uso intensivo de disco de curto prazo. Seria interessante uma exploração mais abrangente dessas diferenças, através da execução de *benchmarks* de E/S prolongados e robustos, como a avaliação de uma aplicação real de cargas de trabalho.

Enquanto os sistemas C3, C2 e N2 usam Google PersistentDisk com especificações como revisão 1, em conformidade com SPC-4 e dispositivo de estado sólido, W1 alcança desempenho superior utilizando discos SSD (modelo: SM2P32A8-512GC1, versão de firmware: VC0S032V, NVMe Versão: 1.4). Por outro lado, A9 também deveria alcançar maior desempenho usando seus discos SSD de alta velocidade (modelo: SKHynix_HFS256GD9TNI-L2B0B, versão de firmware: 11710C10, versão NVMe: 1.3). No entanto, a pilha docker em A9 automaticamente reduz a taxa de transferência de E/S do disco de até 62 MB/s para entre 10 MB/s e 15MB/s.

3.5.4. Rede

A Figura 3.14 apresenta os resultados de execução de desempenho de rede para iPerf com medidas expressas em Gbits/s. Fica evidente pelos dados que existem variações insignificantes de desempenho entre DinD e Docker nos nós C2, C3 e N2, todos hospedados no Google Cloud Platform. Porém, em sistemas de hospedagem local, A9 e W1, os contêineres Docker apresentam desempenho até 7% superior em relação ao DinD.

Um dos principais fatores que contribuem para essa discrepância é o uso de diferentes versões de distribuições e sistemas operacionais Linux (ou seja, versões de *kernel*).

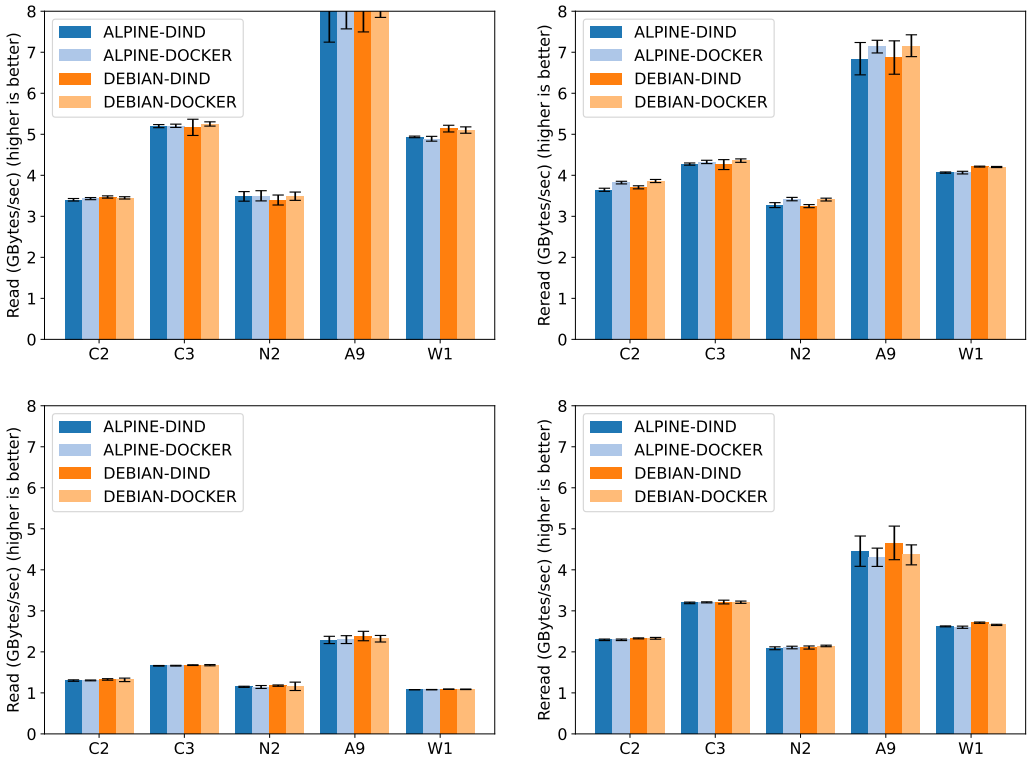

Figura 3.13. Resultados das operações read, reread, write e rewrite usando IOzone

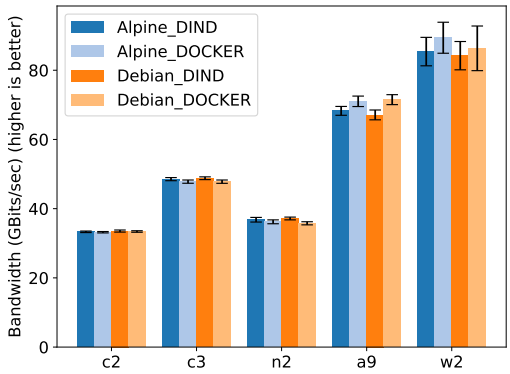

Figura 3.14. Valores de largura de banda da rede coletados por iPerf (FAVA et al., 2024), Figura 9

Embora todos os nós do GCP utilizem o Debian 12, o sistema A9 emprega o Debian 11 e o W1 opera no Ubuntu 22.04. Consequentemente, variações nas versões instaladas de pacotes e bibliotecas contribuem para a diferença de desempenho observada.

3.6. Considerações finais

Neste capítulo introduzimos a ferramenta DinD-Bench e avaliamos o desempenho de contêineres Docker aninhados (DinD) dentro de um contexto de arquiteturas baseadas em microsserviços. A DinD-Bench permite avaliar sistematicamente as capacidades do DinD em uma variedade de ambientes de hospedagem, através da utilização de *benchmarks* bem estabelecidos de CPU, memória, disco e E/S de rede, nomeadamente Sysbench, Stress, IOZone e iPerf. Implantamos esses *benchmarks* em imagens Docker criadas em distribuições Debian e Alpine Linux. Por fim, coletamos as estatísticas dos *benchmarks* em três nós da Google Computing Platform (GCP) e dois nós locais.

As comparações entre as distribuições de Linux distintas (Debian e Alpine), utilizado na execução dos *benchmarks*, revelam que a influência do DinD sobre o Docker em relação ao número de eventos e a latência da CPU são insignificantes para ambos os sistemas operacionais. Além disso, não há diferenças significativas entre DinD e Docker para E/S de disco e rede. No entanto, em termos de consumo de memória, a execução dos contêineres indicam diferenças não negligenciáveis entre Docker e DinD. Além disso, os contêineres DinD necessitam de um tempo de inicialização de até 7 segundos.

Estes fatores, nomeadamente, o consumo de memória e o tempo de inicialização, surgem como potenciais restrições em aplicações paralelas ou arquiteturas de microsserviços que operam sob restrições de tempo ou recursos. Vale a pena enfatizar que algumas das disparidades, como o maior consumo de memória, parecem ser resultado direto da pilha de software em uso, incluindo diferentes versões de *kernel*, bibliotecas e outros pacotes essenciais. Já um tempo de inicialização de 7 segundos pode ser considerado inaceitável para microsserviços de baixa latência e curta duração.

Para facilitar a transparência e a reprodutibilidade, a ferramenta DinD-Bench e os dados coletados estão disponíveis no repositório GitHub (FAVA et al., 2023). Este repositório garante a reprodutibilidade dos experimentos e também pode ser utilizado como um recurso valioso para futuras iniciativas de pesquisa ou experimentação prática.

Referências

AMARAL, M. et al. Performance Evaluation of Microservices Architectures Using Containers. In: *2015 IEEE 14th international Symposium on Network Computing and Applications*. Cambridge, MA, USA: IEEE, 2015. p. 27–34.

BACHIEGA, N. G. et al. Performance Evaluation of Container's Shared Volumes. In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Porto, Portugal: IEEE, 2020. p. 114–123.

BOGNER, J. et al. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Hamburg, Germany: IEEE, 2019. p. 187–195.

Dell Technologies. *How to Test Available Network Bandwidth Using 'Iperf'*. 2021. Alibaba Cloud ECS. <https://www.dell.com/support/kbdoc/en-us/000139427/how-to-test-available-network-bandwidth-using-iperf>.

DIAZ, C. O. et al. Performance Evaluation of an IaaS Opportunistic Cloud Computing. In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. Chicago, IL, USA: IEEE, 2014. p. 546–547.

FAVA, F. B. et al. Assessing the Performance of Docker in Docker Containers for Microsservice-based Architectures. In: *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. Dublin, Ireland: IEEE, 2024. v. 1.

FAVA, F. B. et al. *DinD-Bench*. 2023. <https://github.com/DinDperf/DinD-Bench>.

IBRAHIM, M. H.; SAYAGH, M.; HASSAN, A. E. Too many images on dockerhub! how different are images for the same system? *Empirical Software Engineering*, Springer, v. 25, p. 4250–4281, 2020.

IOZONE. *IOzone Filesystem Benchmark*. 2023. <http://iozone.org/>. Last access in February, 2024.

IPERF.FR. *IPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. 2024. <https://iperf.fr/>. Last access in February, 2024.

Karabey Aksakalli, I. et al. Deployment and Communication Patterns in Microservice Architectures: A Systematic Literature Review. *Journal of Systems and Software*, v. 180, p. 111014, 2021. ISSN 0164-1212.

KOPYTOV, A. *Scriptable database and system performance benchmark*. 2023. <https://github.com/akopytov/sysbench>. Last access in February, 2024.

KUSHWAH, S. *Docker Inside Docker*. 2024. Https://medium.com/@shivam77kushwah/docker-inside-docker-e0483c51cc2c.

OZOR, A. T. *Docker Workflow*. 2023. https://medium.com/augustineozor/docker-workflow-b9fe71d32184.

PETAZZONI, J. *Using Docker-in-Docker for your CI or Testing Environment? Think Twice*. 2024. https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/.

POKHARANA, A.; GUPTA, R. Using Sysbench, Analyze the Performance of Various Guest Virtual Machines on A Virtual Box Hypervisor. In: *2023 2nd International Conference for Innovation in Technology (INOCON)*. Bangalore, India: IEEE, 2023. p. 1–5.

QIAN, C. *Testing I/O Performance with Sysbench*. 2019. Alibaba Cloud ECS. <https://www.alibabacloud.com/blog/testing-io-performance-with-sysbench_594709>.

TAPIA, F. et al. From Monolithic Systems to Microservices: A Comparative Study of Performance. *Applied Sciences*, v. 10, n. 17, 2020. ISSN 2076-3417.

UpCloud. *Evaluating cloud server performance with sysbench*. 2018. <https://upcloud.com/blog/evaluating-cloud-server-performance-with-sysbench>.

WATERLAND, A. *stress - Tool to impose load on and stress test a computer system*. 2023. <https://github.com/resurrecting-open-source-projects/stress>. Last access in February, 2024.

XAVIER, M. G. et al. Understanding Performance Interference in Multi-Tenant Cloud Databases and Web Applications. In: *2016 IEEE International Conference on Big Data (Big Data)*. Washington, DC, USA: IEEE, 2016. p. 2847–2852.