Universidade Federal do Ceará

Campus Quixadá
Curso de Redes de Computadores

Rafael Costa da Silva

Docker versus KVM: Uma análise de desempenho de disco para pequenos arquivos

Quixadá
Julho, 2016

Rafael Costa da Silva

Docker versus KVM: Uma análise de desempenho de disco para pequenos arquivos

Monografia apresentada ao Curso de Redes de Computadores do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial para obtenção do Título de Tecnólogo em Redes de Computadores.

Área de concentração: Computação
Orientador: Prof. Msc. Michel Sales Bonfim
Coorientador: Prof. Msc. Paulo Antonio Leal Rego

Quixadá
Julho, 2016

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S583d Silva, Rafael Costa da.
Docker versus KVM: Uma análise de desempenho de disco para pequenos arquivos / Rafael Costa da Silva. – 2016.
56 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Redes de Computadores, Quixadá, 2016.
Orientação: Prof. Me. Michel Sales Bonfim.
Coorientação: Prof. Me. Paulo Antônio Leal Rego.

1. Computação em nuvem. 2. Medição de desempenho. 3. Sistema operacional. 4. Virtualização. I. Título.
CDD 004.6

Rafael Costa da Silva

Docker versus KVM: Uma análise de desempenho de disco para pequenos arquivos

Monografia apresentada ao Curso de Redes de Computadores do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial para obtenção do Título de Tecnólogo em Redes de Computadores.

Área de concentração: Computação
Orientador: Prof. Msc. Michel Sales Bonfim
Coorientador: Prof. Msc. Paulo Antonio Leal Rego

Aprovada em: __/__/____

BANCA EXAMINADORA

Prof. Msc. Michel Sales Bonfim (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Msc. Paulo Antônio Leal Rego
(Coorientador)
Universidade Federal do Ceará (UFC)

Prof. Msc. Antonio Rafael Braga
Universidade Federal do Ceará (UFC)

Este trabalho é dedicado a todos que acreditam no próprio potencial e trabalham duro para aflorá-lo.

Agradecimentos

Aos meu pais Raimunda Costa da Silva e Raimundo Soares da Silva, que me deram o melhor que um filho pode receber de seus pais. Amor, carinho, educação e valores que levarei comigo por toda vida. As minhas irmãs por serem tão compreensivas e amorosas.

A minha namorada Julie Anne, pelo amor, compreensão, carinho e apoio durantes os momentos difíceis dessa jornada. E a sua família como um todo pelo incrível apoio que vocês me deram.

A todos os meus amigos, que me motivaram, incentivaram e me mantiveram sempre na luta pelos meus objetivos, sem desacreditar. Em especial, Carlos Carvalho, Rogério Carvalho, Erick Bhrener e Hinessa Caminha que aguentaram toda minha chatice, tensão, insegurança e medos durante todo esse tempo de graduação. Obrigado pelos singulares que vivi com vocês.

A todos os professores e servidores da UFC - Campus Quixadá. Especialmente a professora Atslands por quem cultivo imensa admiração, Ticiane Linhares, Michel Sales e Paulo Rego.

Aos meus colegas de turma com quem compartilhei muitos momentos, alegrias tristezas, decepções e vitórias.

Obrigado a todos que de alguma maneira fazem parte da minha vida, sem vocês nada disso seria possível.

"Aprende que nunca se deve dizer a uma criança
que sonhos são bobagens, poucas coisas são tão humilhantes
e seria uma tragédia se ela acreditasse nisso."
William Shakespeare

Resumo

A computação em nuvem utiliza extensivamente a virtualização, pois a mesma oferece a possibilidade de compartilhar recursos físicos através de máquinas virtuais (MVS) com cargas de trabalho diferentes de maneira isolada e segura. Todavia a virtualização adiciona uma camada extra de abstração, o hipervisor. E, consequentemente, uma redução de desempenho em relação ao desempenho nativo (não virtualizado). *Docker* provê uma virtualização leve a nível de sistema operacional (baseada em contêineres). Que remove a camada de abstração adicionada pelos hipervisores, melhorando o desempenho e a velocidade de implantação de ambientes virtualizados, ferramenta amplamente utilizada por provedores de nuvem pública. Por sua vez, o KVM *Kernel-Based Virtual Machine* é uma ferramenta que utiliza a virtualização assistida por *hardware* e possui desempenho muito próximo ao nativo, ótimo isolamento de recursos e pequena carga extra adicionada pelo hipervisor. Executamos nesse trabalho uma análise de desempenho de disco voltada para arquivos pequenos. Avaliamos e comparamos as tecnologias *Docker* e KVM. Identificamos em quais cenários as tecnologias avaliadas são mais apropriadas e qual a sua diferença de desempenho em relação ao desempenho nativo. Utilizando-se da variação do número de contêineres e MVs, avaliando assim, o desempenho de ambas as tecnologias em relação ao disco além do impacto causado pelos contêineres e MVs sobre seu hospedeiro.

Palavras-chaves: análise de desempenho. docker. kvm. virtualização.

Abstract

Cloud computing uses virtualization extensively, because it offers the ability to share physical resources through virtual machines (VMS) with different workloads isolated and safe way. However virtualization adds an extra layer of abstraction, the hypervisor. And consequently a performance reduction compared to native performance (non-virtualized). Docker provides a lightweight virtualization operating system level (based on containers). Which removes the abstraction layer added by hypervisors, improving performance and speed implementation of virtualized environments, widely used by public cloud providers tool. In turn the KVM Kernel-based Virtual Machine it is a tool that uses hardware-assisted virtualization and has very close to native performance, great insulation features and small extra charge added by the hypervisor. We perform this work one geared disk performance analysis for small files. We evaluate and compare the Docker and KVM technologies. Identified in which the scenarios evaluated technologies are most appropriate and what their performance difference compared to native performance. Using varying the number of containers and VMs, thus evaluating the performance of both technologies relative to the disk and the impact caused by containers and VMs on your host

Key-words: Docker. KVM. performance analysis. virtualization.

Lista de ilustrações

Lista de tabelas

Sumário

1 INTRODUÇÃO

A computação em nuvem *(cloud computing)* emergiu a alguns anos como um novo paradigma para hospedagem e entrega de serviços e recursos através da Internet. Segundo Xavier et al. (2015), a computação em nuvem tem oferecido uma grande quantidade de recursos computacionais em uma escala sem precedentes. Essa oferta de recursos tem permitido que os usuários sejam eles de pequeno, médio ou grande porte tenham alternativas para a contratação de recursos em nuvens públicas.

Para um ambiente de computação em larga escala com mais confiabilidade, segurança, disponibilidade e escalabilidade reduzindo os custos de manutenção de uma infraestrutura privada no caso de grandes infraestruturas ou mesmo para pequenas empresas e *startups* que podem beneficiar-se das vantagens da computação em nuvem.

Ainda de acordo com Xavier et al. (2015) recursos como pagamento pelo uso, alocamento de recursos de acordo com as necessidades, redução do investimento pré operacional, elasticidade, escalabilidade horizontal e vertical tem aumentado a popularidade da computação em nuvem e tem levado os fornecedores a um cenário em que o número de clientes está aumentando constantemente. O mesmo tem acontecido para as cargas de trabalho dos ambientes em nuvem que estão cada vez mais diversificadas.

A virtualização é componente fundamental para a computação em nuvem, de acordo Felter et al. (2014), uma vez que grande parte das cargas de trabalho na nuvem estão atualmente em execução em máquinas virtuais (MVs), o desempenho das MVs é um componente crucial do desempenho global em nuvem.

Diversas técnicas de virtualização têm se tornado bastante populares nos últimos anos. Contudo o conceito em si não é recente[1], as primeiras implementações de virtualização em nível de *hardware* foram realizadas pela IBM em *mainframes* na década de 1970. Dentre os principais recursos que podem ser virtualizados, temos: virtualização de redes, armazenamento, software e sistemas operacionais.

Apesar do conceito existir a um tempo considerável a virtualização tornou-se uma solução viável para empresas a partir do ano de 1998 quando a,

> *(VMWARE, 2007)*, descobriu como virtualizar a plataforma x86, que se pensava ser impossível, e criou o mercado de virtualização x86. As economias que dezenas de milhares de empresas têm gerado a partir da implantação desta tecnologia está impulsionando ainda mais a rápida adoção da computação virtualizada desde o *desktop* até o *data center*.

Para gerenciar as MVs, um ambiente virtualizado faz uso do hipervisor, que segundo Xavier et al. (2013), é uma camada de *software* que posiciona-se entre as MVs e o *hardware*, adicionando uma carga extra de uso dos recursos. Visto que as MVs executam sobre o mesmo,

[1] <https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html>

não é possível remover essa carga extra na MV, uma vez que ela é gerada pelo hipervisor. Com essa problemática de sobrecarga gerada pelos hipervisores, têm-se buscado técnicas de virtualização que possam melhorar o desempenho para ambientes virtualizados.

Existe atualmente uma vasta lista de soluções de virtualização, dentre as principais estão, *VMware vSphere*[2], *Xen*[3], KVM[4], *Hyper-V*[5], *Virtual Box*[6], *Citrix*[7], *Virtuozzo*[8] e *Docker*[9]. Segundo Raho Moritz Spyridakis (2015), os mais importantes hipervisores de código aberto são Xen e KVM.

Está entre as principais alternativas à virtualização baseada em hipervisores, à virtualização baseada em contêineres, que provê uma virtualização leve a nível de sistema operacional. De acordo com Dua et al. (2014), contêineres executam instruções nativas diretamente no *hardware* sem a necessidade de emulação de instruções, melhorando consideravelmente o desempenho. Outra alternativa interessante é o *Kernel Based Virtual Machine* (KVM). O KVM também é um hipervisor, contudo ele utiliza-se da técnica de virtualização assistida por hardware, sendo considerado o hipervisor tipo 1. Que possibilita que uma MV execute diretamente sobre o *hardware*, reduzindo assim a complexidade e a sobrecarga. Diferentemente do hipervisor tipo 2 que utiliza paravirtualização, onde o mesmo adiciona uma camada de *software* entre as MVs e o *hardware*.

No presente trabalho realizamos uma análise de desempenho de disco para pequenos arquivos, entre as ferramentas *Docker* e KVM. Os fatores que influenciaram diretamente para escolha das ferramentas foram, ambas serem de código aberto, amplamente utilizadas nos provedores de nuvem pública, possuírem comunidade ativa e participativa, ter política de lançamento de melhorias constantes, documentação completa e de fácil acesso e facilidade para implantação e configuração de um ambiente com tais ferramentas no *hardware* que tínhamos disponível para realização das análises.

Uma análise de desempenho possui diversos fatores que podem ser utilizados afim de comparar tecnologias diferentes. Nós focamos na questão do desempenho de disco para arquivos pequenos por ser ser um fator determinante para diversos tipos de aplicações e por existirem poucos estudos com foco especifico nesse de quesito.

Para realização dos experimentos foi necessário configurar o servidor onde os mesmos iriam ser executados. A configuração do *Docker* foi simples, bastou instalá-lo através de seu repositório. Como o KVM não possui uma interface de linha de comando tão completa como a do *Docker*, utilizamos o *Vagrant* para gerenciar suas VMs. *Vagrant* é uma ferramenta desenvolvida

[2] <http://www.vmware.com/products/vsphere-hypervisor.html>
[3] <http://www.xenproject.org/>
[4] <https://openvirtualizationalliance.org/what-kvm>
[5] <https://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx>
[6] <https://www.virtualbox.org/>
[7] <https://www.citrix.com/>
[8] <https://virtuozzo.com/>
[9] <https://www.docker.com/>

para criar e gerenciar MVs de maneira facilitada, oferece uma linha de comando simples para o gerenciamento das mesmas, e integra-se com o KVM através de um *plugin* do *libvirt*. Após configurar o ambiente para executar *Docker* e KVM foi escolhida a ferramenta pra realizar os testes de desempenho de disco. A Ferramenta escolhida foi o *iozone*, de acordo com Tarasov et al. (2011), o iozone foi utilizado em grande parte dos trabalhos que realizaram testes de sistemas de arquivos ou de disco até o momento da elaboração do estudo. Em seguida foram elaborados os *scripts* responsáveis por executarem os testes de desempenho.

Diante do ambiente configurado e pronto para realização das análises, foram definidos os cenários. No primeiro cenário foram analisados e comparados *Docker*, KVM e o desempenho nativo. Do segundo ao quinto cenários foram avaliados apenas o desempenho do *Docker* e KVM.

A presente análise pretende definir em que tipo de cenário cada uma das ferramentas avaliadas possui melhor desempenho, auxiliando na decisão de qual ferramenta utilizar quando se estiver implantado, contratando uma infraestrutura em nuvem ou até mesmo auxiliar na escolha da ferramenta para implantar um determinado tipo de aplicação.

O trabalho está organizado da seguinte maneira. A próxima seção 2 contém os objetivos do nosso trabalho. A seção 3 descreve os trabalhos relacionados que serviram de base, nela são demonstrados as semelhanças e diferenças entre eles. Na seção 4, são fundamentados os conceitos necessários para compreensão deste trabalho. Na seção 5 são detalhados todos os experimentos por nós realizados. Na seção 6 são analisados os resultados obtidos nos experimentos. Por fim a seção com as conclusões obtidas 7.

2 OBJETIVOS

2.1 Objetivo Geral

O objetivo do presente trabalho é analisar e comparar o desempenho de operações de disco para pequenos arquivos entre contêineres *Docker* e máquinas virtuais providas pelo KVM.

2.2 Objetivos específicos

a) Configurar um servidor de modo a possibilitar variar o número de MVs de maneira fácil e rápida para que seja possível realizar experimentos com diversos cenários.

b) Avaliar o impacto das técnicas de virtualização no hospedeiro e seus desempenhos.

c) Analisar os dados coletados e avaliar qual das tecnologias possui melhor desempenho de disco.

3 TRABALHOS RELACIONADOS

Existem diversos trabalhos na literatura que realizaram análises de desempenho envolvendo virtualização baseada em contêineres e/ou virtualização baseada em hipervisores. Em Xavier et al. (2013) foram realizadas análises de desempenho em um ambiente de computação de alta performance (HPC), onde foram avaliadas técnicas de virtualização baseada em contêineres, hipervisor e um *hardware* nativo *High Performance Computing* (HPC).

O trabalho de Xavier et al. (2013) analisou e comparou o desempenho das técnicas de virtualização em um ambiente de computação de alta performance, com intuito de comparar a sobrecarga gerada pelas técnicas de virtualização baseada em contêineres e virtualização baseada em hipervisores. Foram avaliados e comparados também o desempenho das técnicas de virtualização em relação ao desempenho do *hardware* nativo.

Em Xavier et al. (2013), avaliou-se também se as ferramentas de virtualização baseadas em contêineres já possuíam maturidade e os requisitos necessários para serem adotadas em ambientes HPC. Uma vez que, devido a alta sobrecarga causada pela virtualização baseada em hipervisores, os mesmos ainda não haviam sido adotados em ambientes de computação de alto desempenho. As análises mensuraram desempenho quanto a processamento, memória, rede, disco e isolamento.

O presente trabalho relaciona-se diretamente com Xavier et al. (2013) pois também realiza uma análise de desempenho entre técnicas de virtualização baseada em hipervisores e contêineres. Contudo são bem distintos, no nosso trabalho são avaliadas soluções de virtualização diferentes das avaliadas em Xavier, M. G. et all (2013) seja no virtualização baseada em contêineres onde avaliamos o *Docker*. Ao invés de virtualização baseada em hipervisores do tipo 2 foi adotada a virtualização assistida por *hardware* com o KVM, hipervisor tipo 1. Nosso trabalho foca especificamente na análise de desempenho de disco para pequenos arquivos. Utilizando-se da variação de alguns fatores para obter uma análise mais específica do elemento avaliado.

A análise de desempenho realizada em Joy (2015) também comparou o desempenho das técnicas de virtualização baseada hipervisor e em contêineres, entretanto com um foco diferente da análise supracitada, as análises objetivaram avaliar e mensurar a escalabilidade horizontal de um ambiente de máquinas virtuais (MV) em comparação a um ambiente com contêineres. Visto que estudos anteriores tinham como propósito avaliar o desempenho de uma determinada aplicação dentro de uma máquina virtual e/ou contêiner e não do ambiente como um todo.

Partindo desse pressuposto avaliou-se a influência que a porcentagem de uso de CPU de uma máquina virtual pode causar em outras, impacto que uma falha em uma determinada MV pode causar em uma aplicação; considerando-se o tempo necessário para que outra MV gasta na inicialização para que esteja completamente funcional para a aplicação, escalonamento baseado em gatilhos como porcentagem de uso de CPU.

A relação entre o presente trabalho e Joy (2015), dar-se-a principalmente através do foco de suas análises, ambas avaliam o desempenho de MVs e contêineres em relação ao consumo de recursos de seu hospedeiro. É avaliada a relação entre o consumo de recursos no hospedeiro com a variação do número de MVs e contêineres que estarão sendo executados em um determinado período de tempo.

A presente análise diferencia-se de Joy (2015) pois avalia além do impacto da porcentagem de uso de recurso do hospedeiro, como das ferramentas de virtualização avaliadas. Além do desempenho de disco entre os dois tipos de virtualização. Outro fator diferenciador é o hipervisor utilizado nas análises. Nós utilizamos o hipervisor KVM do tipo 1. Diferentemente de Joy (2015) que utilizou o hipervisor *Xen* do tipo 2.

Em Felter et al. (2014) é realizada uma avaliação de desempenho entre as técnicas de virtualização baseada em contêineres utilizando o *Docker* versus virtualização assistida por *hardware* com o KVM. O foco da avaliação de Felter et al. (2014), foi na comparação de sobrecarga de um ambiente virtualizado e de um *hardware* nativo, tendo em vista o impacto da sobrecarga sobre os recursos e para o fluxo normal de trabalho de um servidor. Para que fosse possível mensurar o impacto da sobrecarga em um servidor, foram realizados testes de estresse de recursos como disco, CPU, memória e tráfego de rede. Para possibilitar que nos testes de estresse tanto *Docker* quanto KVM podessem consumir todos os recursos do servidor, foram realizadas configurações específicas utilizando o *cpufreq governor*[10] que permite desabilitar o gerenciamento de energia no servidor, permitindo maior taxa utilização de recursos.

Os trabalhos assemelham-se pois ambos realizam uma comparação de desempenho entre *Docker* e KVM, avaliam o desempenho de disco. Contudo este trabalho além de realizar um teste voltado especificamente para desempenho de disco, também analisa o impacto da variação da quantidade de contêineres e VMs no hospedeiro. Além disso o *Docker* possui melhorias significativas de segurança, isolamento e restrição de recursos na versão 1.11, que foi a adotada neste trabalho.

No trabalho Paul e Sastri (2015) é realizada uma comparação de desempenho entre *Docker* e uma máquina física. São avaliados e comparados desempenho de disco, memória, CPU e de rede. Afim de atestar que o desempenho de um contêiner *Docker* sem limitação de recursos possui desempenho muito próximo de um *hardware* nativo, sem virtualização. Com esses testes foi possível mostrar que o *Docker* possui realmente desempenho muito semelhante a um ambiente sem virtualização. Apontando o *Docker* como uma tecnologia promissora.

Em Paul e Sastri (2015) são realizados testes de desempenho de disco, CPU, memória e rede para um contêiner e para hospedeiro, os trabalhos assemelham-se nesse ponto, visto que ambos realizam um avaliação do desempenho de disco. Contudo diferenciamo-nos por realizar uma análise mais completa na parte de desempenho de disco, com diversos tamanho de arquivos,

[10] <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Power_Management_Guide/cpufreq_governors.html>

diversos cenários diferentes e com avaliação do impacto do aumento do número de VMs e/ou contêineres.

4 FUNDAMENTAÇÃO TEÓRICA

Esta seção aborda e explana os conceitos necessários para compreensão da proposta deste trabalho, conceitos esses indispensáveis para a realização do mesmo.

4.1 Virtualização

Existem diversas definições de virtualização aceitas pela academia, de acordo com Xavier et al. (2013), virtualização de recursos consiste na utilização de uma camada de *software* intermediária na parte superior de um sistema de base de modo a proporcionar captações de recursos virtuais. Em geral, os recursos virtuais são chamadas máquinas virtuais (MVs) e pode ser visto como contextos de execução isolados.

Há uma variedade de técnicas de virtualização. Em geral os ambientes virtualizados são compostos por um grande número de máquinas físicas (MFs), que juntas formam um centro de dados *(datacenter)* e interligam-se através de uma ou mais redes. Dentro das máquinas físicas é onde são executadas as máquinas virtuais, a quantidade de MVs contidas em uma MF varia de acordo com a quantidade de recursos que a mesma possui.

De acordo com Intel (2006), a maioria das organizações de TI atualmente reservam uma maioria significativa de seus recursos para manutenção de sistemas e aplicações existentes. Virtualização e consolidação de servidores pode ajudar a aumentar a utilização, simplificar o ambiente e reduzir os custos totais.

A computação em nuvem atualmente está consolidada e é uma realidade no mercado de tecnologia. Segundo a consultoria *gartner*[11], o mercado mundial de serviços de nuvem pública deverá crescer 16,5 por cento em 2016 para um total de $ 204 bilhões de dólares, acima dos US $ 175 bilhões em 2015. Ainda segundo a consultoria o maior crescimento advém de serviços de revenda de infraestrutura como serviço (IaaS), que está projetado para crescer 38,4 % somente no ano de 2016.

A virtualização por sua vez não pode ser dissociada da computação em nuvem, de acordo com Zhang et al. (2010), virtualização é uma tecnologia chave para o sucesso do modelo de computação em nuvem, sendo a principal das tecnologias envolvidas. Provendo sua base, uma vez que fornece a capacidade de reunir recursos de computação de um conjunto de servidores e dinamicamente atribuir ou reatribuir recursos virtuais para aplicações sob demanda.

[11] <http://www.gartner.com/newsroom/id/3188817>

Como citado previamente, o conceito de virtualização não é recente. De acordo com Graziano (2011), acredita-se que a virtualização tem as suas origens no final dos anos 1960 e início dos anos 1970, quando a IBM investiu muito tempo e esforço no desenvolvimento de soluções de compartilhamento de tempo robustas. Compartilhamento de tempo *(time-sharing)* refere-se ao compartilhamento de recursos de computação entre usuários de um mesmo recurso físico, com o objetivo de aumentar a eficiência dos mesmo por parte dos usuários e/ou aplicações.

Quando a *VMware* implementou a primeira solução de virtualização corporativa para a plataforma de arquitetura x86 no ano de 1998, o que até o momento acreditava-se ser impossível, criou-se a partir de então o mercado de virtualização x86. A solução desenvolvida pela *VMware* consistia de uma combinação de tradução binária e execução direta no processador que permitia que vários sistemas operacionais hóspedes fossem executados com isolamento total no mesmo computador. Segundo a (VMWARE, 2007), as economias que dezenas de milhares de empresas têm gerado a partir da implantação desta tecnologia está impulsionando ainda mais a rápida adoção da computação virtualizada desde o *desktop* até o *data center*.

A popularização da virtualização fez com que fornecedores de hardware adotassem rapidamente a virtualização e desenvolvessem novos recursos para simplificar as técnicas de virtualização. A virtualização é a tecnologia que possibilita que os provedores de infraestrutura como serviço (IaaS), plataforma como serviço (PaaS) e *software* como serviço (SaaS) como *Amazon Web Services* (AWS)[12], *Google Cloud Platform*[13] e *Microsoft Azure*[14] consiguam oferecer seus serviços em nuvem.

A virtualização traz uma série de benefícios, segundo (VMWARE, 2007), estão entre os principais:

- Redução dos custos operacionais.
- Redução do tempo gasto em tarefas administrativas de TI.
- Facilidade para fazer backup e proteção de dados.
- Consolidação de servidores.
- Aumento da disponibilidade de aplicativos.
- Facilidade para recuperação de falhas.

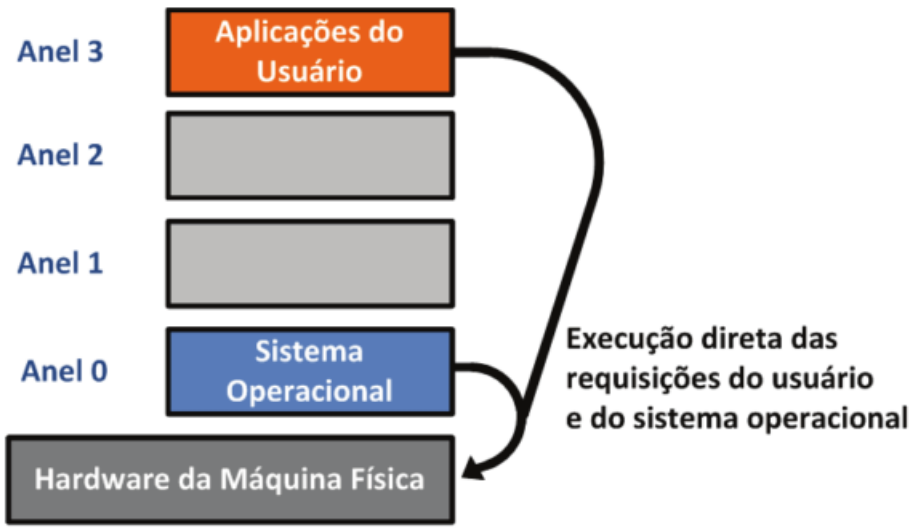
Uma MV é uma abstração virtualizada de uma máquina física, tal abstração é possível devido aos hipervisores ou como também são conhecidos os monitores de máquinas virtuais (VMM). Eles são os responsáveis por interceptar e emular algumas instruções emitidas pelas MFs. Um hipervisor provê uma interface que permite ao usuário inicializar, pausar, serializar e desligar múltiplas MVs (KEAHEY et al., 2007).

[12] <https://aws.amazon.com/>
[13] <https://cloud.google.com/>
[14] <https://azure.microsoft.com/en-us/>

Dentre as diversas opções de hipervisores existentes resolveu-se utilizar o KVM, pois ele está entre os mais populares atualmente, é uma solução de código aberto, possui um projeto colaborativo da *Linux Foundation* o *Open Virtualization Alliance* que é uma organização composta por líderes da indústria de virtualização, *data center* e soluções de nuvem, com foco no aumento da conscientização e adoção de MKVM. Principais participantes são, Intel, IBM e Red Hat.

Existem diversas técnicas de virtualização de recursos, contudo existem diversas diferenças entre elas. Dividimo-nas nos tópicos a seguir. A principal delas é a forma como instruções privilegiadas advindas das MVs alcançam o hardware. Sistemas operacionais de arquitetura x86 foram projetados para acessar os recursos do hardware diretamente. Ou seja funcionam sobre o hardware real de modo a possuir o controle completo dos recursos. Conforme a Figura 1, a arquitetura x86 oferece quatro níveis de privilégio. São eles, Anel 0, 1, 2 e 3, essa divisão foi implementada para que sistemas operacionais e aplicativos pudessem gerenciar o acesso ao hardware do computador.

Figura 1 – Níveis de privilégio da arquitetura x86 nativa.



Fonte: (VMWARE, 2007)

Aplicações de nível de usuário utilizam o Anel 3 geralmente, o sistema operacional precisa ter acesso direto à memória e hardware, e deve executar suas instruções privilegiadas no Anel 0. A virtualização da arquitetura x86 adiciona uma camada de virtualização sob o SO hospedeiro, é essa camada que permite criar e gerenciar as MVs que acessam recursos compartilhados. O principal problema dessa abordagem é que algumas instruções sensíveis não podem efetivamente ser virtualizadas, por possuírem semânticas distintas quando não são executadas no Anel 0. O desafio de capturar e traduzir solicitações de instrução sensíveis e privilegiadas, em

tempo de execução, fez com que se pensasse ser impossível realizar virtualização da arquitetura x86.

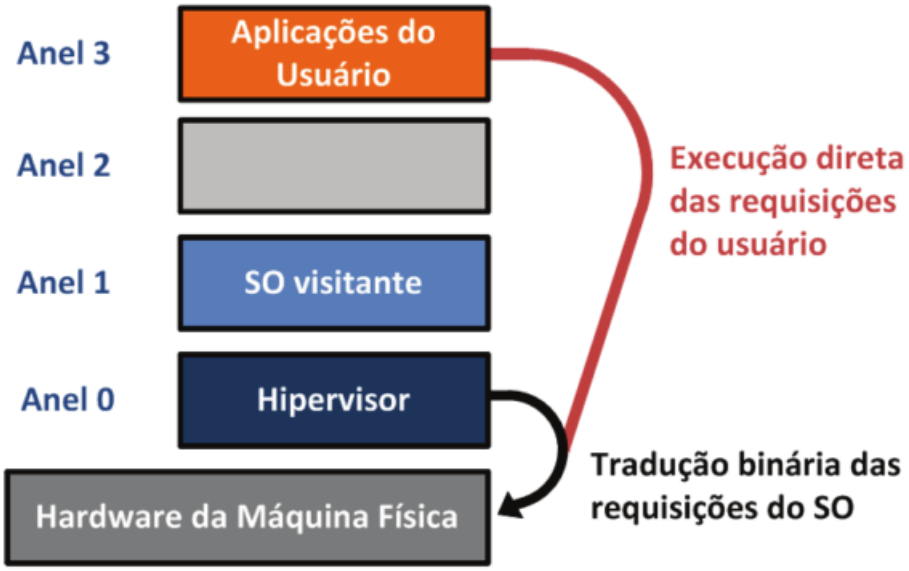
4.1.1 Virtualização total

Este tipo de virtualização provê uma virtualização completa do *hardware* adjacente através de emulação. Com isso é possível criar dispositivos virtuais com tudo o que é necessário para executar um SO sobre o mesmo, por acreditar está executando sobre *hardware* real não é necessário fazer modificações no *Kernel* do SO visitante.

É comum que sejam combinadas técnicas de execução direta juntamente com tradução binária para realizar as chamadas de sistema.Tais chamadas são interceptadas pelo hipervisor, que as mapeia para o hardware real subjacente como é mostrado na Figura 2, enquanto isso afim de obter um melhor desempenho parte do código do nível do usuário pode ser executado diretamente no processador. Quanto ao hipervisor ele é quem garante independência e autonomia entre as MVs que estão executando sobre um mesmo *hardware*.

A virtualização total permite executar MVs com SO *Windows* em máquinas físicas com sistema operacional *Linux*. *VMware* e *Virtual Box* são hipervisores que utilizam esse tipo de virtualização.

Figura 2 – Abordagem de virtualização total com tradução binária em arquitetura x86.



Fonte: (VMWARE, 2007)

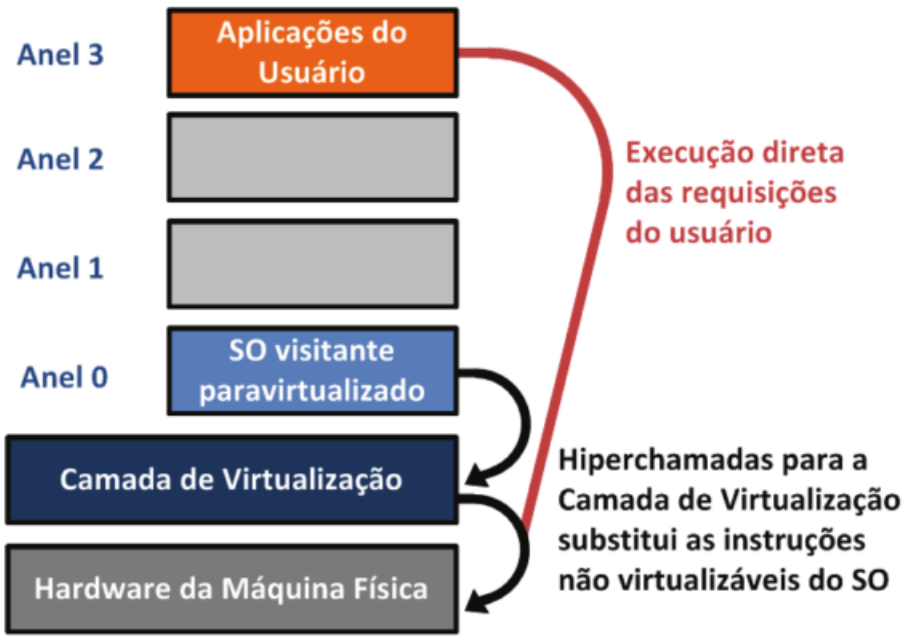
4.1.2 Paravirtualização

Este tipo de virtualização requer que o *kernel* do SO visitante seja modificado especificamente para executar no hipervisor. Na Figura 3, podemos visualizar esse processo, que inclui substituição de quaisquer operações privilegiadas, que só poderiam ser executadas no Anel 0 da CPU, por chamadas ao hipervisor conhecidas como *hypercalls*.

Nessa técnica de virtualização é o hipervisor que é reponsável por executar a tarefa em nome da MV, além disso ele oferece interfaces de hiperchamadas para outras operações críticas do *Kernel*, tais como gerenciamento de recursos de memória, CPU e interrupções.

A paravirtualização busca sanar deficiências da virtualização total, afim de permitir que as MVs tenham acesso ao hardware subjacente. Diferentemente da virtualização total na paravirtualização as MVs sabem que estão executando sobre *hardware* virtualizado.

Figura 3 – Abordagem com paravirtualização em arquitetura x86.



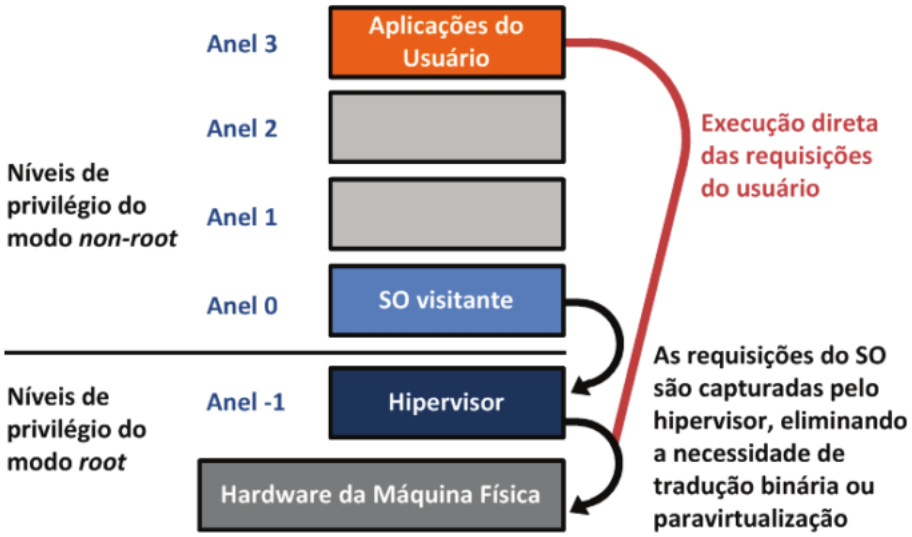
Fonte: (VMWARE, 2007)

4.1.3 Virtualização assistida por hardware

De acordo com a (VMWARE, 2007), este tipo de virtualização utiliza recursos das tecnologias *Intel Virtualization Technology* (VT-x) e AMD-V. Ambos os processadores oferecem extensões necessárias para executar MVs com SOs não modificados, mitigando as desvantagens

inerentes a emulação de *hardware*, como a tradução binária. Na Figura 4, podemos visualizar como esses processadores oferecem um modo de privilégio adicional abaixo do anel 0. Este privilégio adicional fornecido por esses processadores é conhecido como Anel -1 e permite que os hipervisores que suportam esta tecnologia executem no Anel -1 como mostra a Figura 4. Deu-se o nome de virtualização assistida por *hardware* a está tecnologia pois ela permite que o hipervisor virtualize eficientemente todo o conjunto de instruções x86, interagindo com as instruções críticas usando um modelo clássico através do hardware, ao invés de software. Os hipervisores que suportam esta tecnologia podem funcionar no Anel -1 e as MVs podem utilizar a CPU no Anel 0, como é feito o acesso numa MF. Isto permite virtualizar SOs visitantes sem nenhuma modificação.

Figura 4 – Abordagem com virtualização assistida por hardware.



Fonte: (VMWARE, 2007)

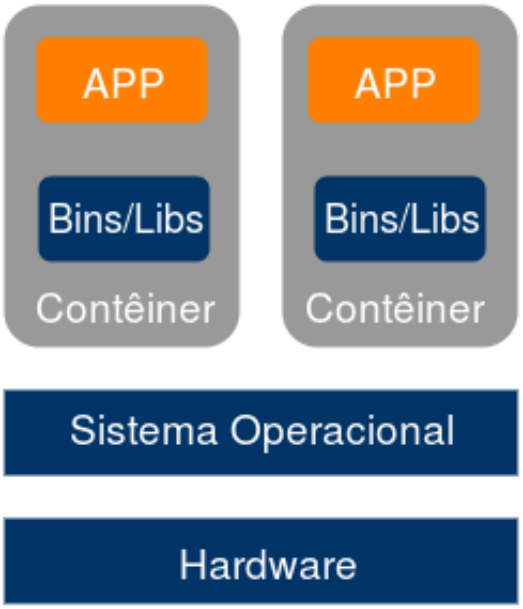
4.1.4 Virtualização ao nível do sistema operacional

Virtualização ao nível do sistema operacional também conhecida como virtualização assistida por contêineres. Técnica de virtualização que permite que o *Kernel* de um sistema operacional crie contêineres ou mesmo de partições isoladas em uma única MF. Na Figura 5, podemos visualizar como os contêineres executam sobre uma camada de software que é adicionada sobre o SO hospedeiro fazendo uso do mesmo, entretanto os contêineres executam de maneira isolada, o *Kernel namespaces* é o recurso que implementa isolamento de recursos como sistema de arquivo, identificador de processos (PID), rede, usuários, comunicação inter processos (IPC) e *hostname*.

Apesar da utilização compartilhada dos recursos do hospedeiro cada contêiner tem seus próprios recursos e funciona isolado dos outros. Nesta virtualização cada contêiner tem seu próprio sistema de arquivos mas partilham o *Kernel* do hospedeiro, ou seja o escalonamento de CPU, gestão de memória, interrupções e gestão de outros recursos é realizada pelo hipervisor utilizando recursos providos pelo SO hospedeiro. Estão entre os principais hipervisores dessa técnica de virtualização OpenVZ, Virtuozzo e o Docker.

A Figura 5 mostra a arquitetura da virtualização baseada em contêineres, onde os contêineres compartilham compartilham o *Kernel* com o sistema operacional hospedeiro.

Figura 5 – Abordagem de virtualização baseada em contêineres.



Fonte: Autor

4.2 Docker

Docker é uma plataforma para desenvolvedores e administradores de sistemas que possibilita desenvolver, empacotar, entregar, implantar e executar aplicações em contêineres. A plataforma *Docker* possibilita separar as aplicações da infraestrutura, possibilitando tratá-la como um aplicativo gerenciável, permitindo montar rapidamente aplicativos a partir de componentes previamente definidos. De acordo com a documentação oficial[15], a plataforma permite obter o código testado e implantado em produção o mais rápido possível, através da combinação de uma plataforma de virtualização de contêineres leve, com fluxos de trabalho e ferramentas que ajudam a gerenciar e implantar suas aplicações. Em resumo, a plataforma *Docker* fornece uma maneira de executar praticamente qualquer aplicação segura isolada em um contêiner.

[15] <https://docs.docker.com/>

Segundo desenvolvedores, os principais usos da plataforma seriam:

- **Entrega rápida de aplicações**: Esse item está relacionado ao ciclo de desenvolvimento de aplicações, utilizando a plataforma *Docker* é possível que os desenvolvedores configurem seus ambientes e aplicações localmente e através do compartilhamento de imagens através do *Docker Hub*[16]. Sua equipe consigue reproduzir o mesmo ambiente em qualquer outro servidor.
- **Implantação e escalabilidade mais fácil**: Devido a natureza leve e simples dos contêineres e sua ampla adoção por diversos provedores de IaaS torna-se simples sua escalabilidade tanto em ambientes em nuvem quanto em ambientes sob demanda.
- **Alcançar maior densidade e funcionar mais cargas de trabalho**: *Docker engine* é rápido e leve. Fornece uma alternativa viável e com menor custo do que máquinas virtuais baseadas em hypervisor.

A plataforma *Docker* contém diversos componentes, que podem atuar juntos ou separadamente. A terminologia dos componentes da plataforma é importante para entender como os mesmos interagem entre si, sua função e seu uso no presente trabalho. É importante frisar que quando refere-se a plataforma *Docker* estamos nos referindo a sua completude, ou seja, todos os recursos por ela oferecidos. Enquanto que *Docker engine* refere-se especificamente ao motor do *Docker* responsável pelo gerenciamento dos contêineres dentre outras funções, que serão explanadas mais a frente. Contudo a partir deste ponto quando utilizarmos o termo *Docker* estaremos nos referindo ao motor.

A plataforma *Docker* possui um desenvolvimento frenético dado o número considerável de contribuidores que atualmente estão engajados no projeto como aponta o repositório da plataforma no *github*[17]. Fazendo com que a mesma tenha constantemente atualizações, adição de novos recursos, integração com ferramentas, correção de falhas e implantação de melhorias. Por essa dinamicidade não descreveremos todos os seus componentes no presente trabalho, mas apenas os componentes por nós utilizados.

Docker: Como já definido previamente é uma plataforma de código aberto de virtualização de contêineres.

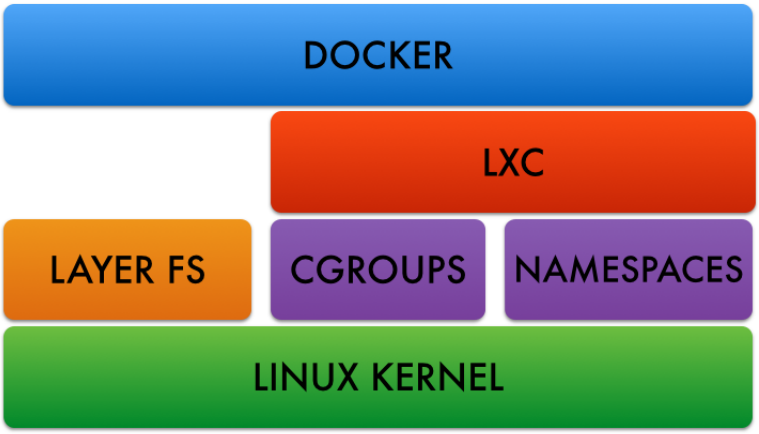
Docker Hub: É um serviço de registro de imagens de contêineres *Docker* baseado em nuvem. Permite conectar-se a repositórios de imagens, onde é possível construir testar e armazenar imagens. Além de permitir lincar imagens diretamente da *Docker Cloud*[18] de modo que é possível implantar imagens em qualquer servidor. Esse serviço fornece um recurso centralizado para a descoberta de imagem de contêineres, distribuição e gerenciamento de mudanças, de usuário, colaboração em equipe e automação de fluxo de trabalho em todo o fluxo de desen-

[16] <https://hub.docker.com/>
[17] <https://github.com/docker/docker>
[18] <https://cloud.docker.com/>

volvimento. No presente trabalho foi utilizado o *Docker Hub* para armazenar a imagem base para o contêiner em que foram realizadas as análises.

O núcleo da plataforma *Docker* é o *Docker Engine*, um motor *(engine)* em tempo de execução leve e robusto que cria e executa os contêineres *Docker*. O *Docker Engine* executa diretamente sobre o *Kernel Linux* para criar o ambiente operacional para as aplicações distribuídas. o *Docker* faz uso de diversos recursos do Kernel Linux para possibilitar que os contêineres possam realizar as ações de gerenciamento de contêineres, imagens, rede e volumes, a Figura 6 ilustra como o *Docker* utiliza recursos do sistema operacional para prover isolamento e gerenciamento de recursos. Na Figura 6, podemos visualizar uma camada que é o *Docker* e sob ela alguns dos recursos do sistema operacional utilizados pelo *Docker* para gerenciar e prover isolamento de recursos, como *cgroups*[19], *namespaces*[20], e o *lxc*[21]. *Docker* é um dos projetos de código aberto com crescimento mais rápido até hoje no *GitHub*. O mesmo está disponível para *download* do código, que é aberto, ou com uma assinatura de suporte comercial.

Figura 6 – Componentes internos do Docker



Fonte: Dedoimedo (2016)

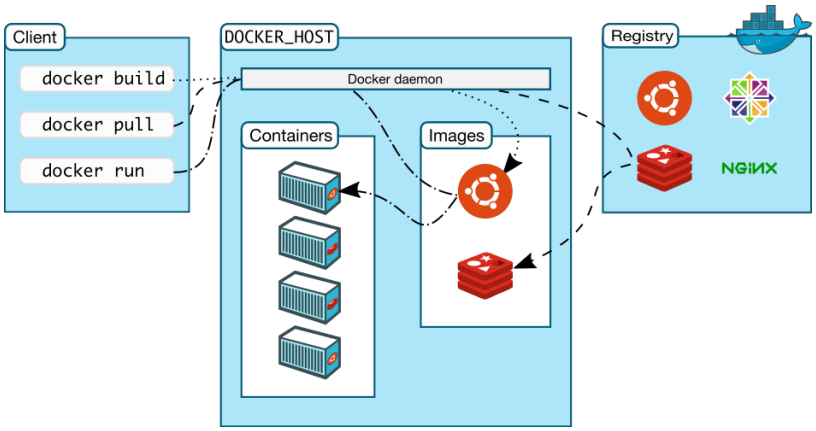
4.2.1 Arquitetura

A arquitetura do *Docker* é cliente servidor. Onde o *Docker client*, provê uma interface facilitada para o usuário interagir com o *Docker Daemon* que é responsável por executar as atividades relacionadas aos recursos do *Docker*, que serão descritos em detalhes ao longo desse trabalho.

[19] <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>
[20] <http://man7.org/linux/man-pages/man7/namespaces.7.html>
[21] <https://linuxcontainers.org/lxc/documentation/>

Docker Client e *Docker Daemon* podem funcionar na mesma máquina ou em máquinas separadas, podendo comunicar-se através de *sockets* ou *RESTful* API. A Figura 7, representa a arquitetura básica da comunicação entre tais componentes, com a adição do *Docker Registry* que é o responsável por prover as imagens dos contêineres que o usuário deseja implantar no ambiente. Nesse caso, poderia ser o *Docker Hub* ou outro serviço de registro de imagens fornecido por uma empresa ou mesmo um serviço de registro local.

Figura 7 – Arquitetura cliente servidor Docker



Fonte: Docker (2016)

O funcionamento interno do *Docker* é fundamentado nos seguintes recursos:

- *Docker images*.
- *Docker registries*.
- *Docker containers*.

Docker images: São *templates* somente leitura para criação de um contêiner. Pode-se, por exemplo criar um contêiner a partir de uma imagem que contém apenas um sistema operacional como o *debian* ou mesmo a partir de uma imagem que contenha um serviço ou aplicação com um *web server*, que pode já está previamente instalado e configurado. As imagens são os componentes de criação de contêineres do *Docker*.

Docker registries: É uma aplicação servidora altamente escalável que é utilizada para manter as imagens do *Docker*. Essas imagens podem ser públicas ou privadas, em caso de serem públicas ficam disponíveis para uso geral. Como já citado, o serviço de registro público do *Docker* é provido pelo *Docker Hub*. Com esse serviço é possível baixar imagens já criadas e/ou configuradas por terceiros, criar e compartilhar suas próprias imagens. Esse serviço é considerado o componente de distribuição do *Docker*.

Docker Containers: Um contêiner assemelha-se a um diretório que contém tudo o que é necessário para que uma aplicação ser executada. Uma vez que cada contêiner e criado a partir

de uma imagem, pode-se considerar um contêiner como uma instância em execução de uma imagem. Cada contêiner executa isoladamente de maneira segura, eles são o componente de execução do *Docker*.

Cada imagem consiste de uma série de camadas, a geração de uma imagem a partir de tais camadas é possível devido ao modo como o *Docker* faz uso do *Union File System* (Unionfs)

> *Unionfs* pode ser útil para manter conjuntos relacionados, mas de diferentes arquivos em locais separados. Os usuários, no entanto, muitas vezes preferem ver estes arquivos relacionados juntos. Nesta situação, o *Unionfs* permite que os administradores mantenham esses arquivos separados fisicamente, mas que seja possível fundi-los logicamente em um único ponto de visão. Uma coleção de diretórios mesclados é chamado de *Union*, e cada diretório físico é chamado de um *branch* Wright e Zadok (2004).

Uma das razões para a leveza do *Docker* é exatamente o uso de camadas. Quando é necessário alterar uma imagem existente, instalar um novo pacote ou fazer implantação de um novo recurso de uma aplicação. Adiciona-se apenas uma nova camada as existentes. Ao invés de construir uma imagem completamente nova como é comum de acontecer quando se trata de MVs. Quando a alteração é feita em uma imagem quem a utiliza só precisa obter a nova camada e não a imagem completa, visto que as outras camadas já existem.

Cada imagem inicia de uma imagem base, que nada mais é do que uma imagem de um sistema operacional qualquer, super enxuta, é possível ter como imagem base um *Ubuntu, CentOS* ou imagens minimalista como *Alpine*[22] que possui apenas 5 MB de tamanho. No presente trabalho foi utilizada a imagem base do *Ubuntu* para a execução das análises.

Após definir a imagem base é que realmente inicia-se o processo de construção de uma imagem personalizada. O processo de geração de uma nova imagem personalizada a partir de uma imagem base dar-se através da execução de um passo a passo, também chamados instruções. As instruções incluem diversas operações, que podem incluir:

- Executar um comando.
- Instalar um ou mais pacotes.
- Adicionar arquivo ou diretório.
- Criar uma variável de ambiente.

Todos essas instruções são armazenadas em um arquivo chamado *Dockerfile*. O *Docker* lê esse arquivo quando é solicitada a criação de uma imagem, ele executa as instruções uma a uma e caso não ocorram erros ele retorna uma imagem final. Cada instrução corresponde a uma camada na imagem final, ou seja, deve-se ter cuidado ao criar imagens com muitas instruções. De acordo com a documentação oficial do Docker (2016) é possível criar mais de 127 camadas em uma imagem, contudo não se tem um número limite especificado.

[22] <http://alpinelinux.org/>

Um contêiner consiste em um sistema operacional, que contém arquivos adicionados pelo usuário e metadados. Como citado anteriormente, o contêiner é construído a partir de uma imagem. Essa imagem fornece ao *Docker* as informações que o contêiner possui, o processo a ser executado quando o recipiente é executado e outras configurações que o mesmo possui. A imagem *Docker* é somente leitura ao se executar um contêiner a partir de uma imagem, ela adiciona uma camada de leitura e escrita na parte superior da imagem. A camada de escrita é quem possibilita que o contêiner poder receber modificações

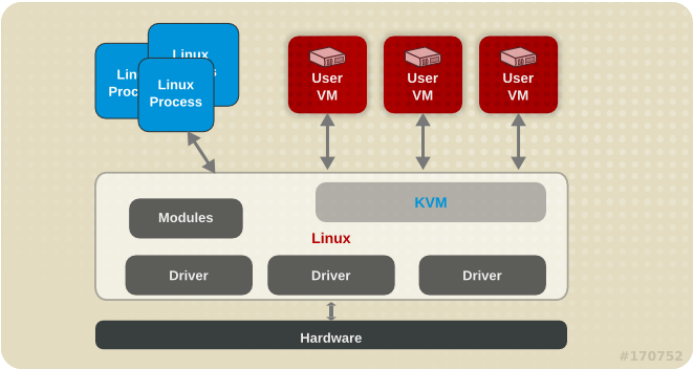
4.3 KVM

Segundo Chen (2011), o KVM é um projeto *open source* que foi desenvolvido inicialmente pela empresa israelense Qumranet, e adquirido posteriormente pela Red Hat em 2008. KVM é implementado como um módulo que é carregado dentro do *kernel Linux*, o módulo foi incorporado ao *Kernel Linux* a partir da versão 2.6, já o componente de espaço de usuário foi adicionado previamente com o *QEMU* na versão 1.3 do *Kernel Linux*. Permitindo que qualquer distribuição *Linux* que utilize um *Kernel* 2.6 ou mais novo possa usufruir do KVM, desde que possua suporte a virtualização assistida por *hardware*.

4.3.1 Arquitetura

De acordo com Kadam (2014), KVM é um módulo carregável no kernel do Linux que permite ao sistema operacional Linux funcionar como um hypervisor tipo 1, como explicado no tópico 4.1.3. Na Figura 8, podemos visualizar o módulo do KVM e como ele interagi com os outros componentes do sistema operacional.

Figura 8 – Arquitetura e componentes do KVM



Fonte: Red Hat (2016)

Um hipervisor pode ser visto como um sistema operacional especializado em executar VMs ao invés de aplicações comuns. Exceto pela necessidade de especialização em virtualiza-

ção, o restante das tarefas de um hipervisor são semelhantes a um sistema operacional como *Linux*, *FreeBSD* ou *Windows*. Tarefas essas que incluem gerenciamento de memória, escalonamento de processos, lidar com *drivers* , gerenciamento de entrada e saída (I/O), etc.

O módulo KVM implementa no *Linux* as capacidades essenciais para que MVs funcionem sobre o mesmo. Segundo Gillen (2011), adota a filosofia de não reinventar a roda e usa as funções do sistema operacional *Linux* estabelecidas e comprovadas para o resto. Reduzindo ou até mesmo eliminando a necessidade de reescrever funções básicas, permitindo que o desenvolvimento volte os seus esforços na otimização do *Linux* para o gerenciamento de processos de VMs. Outro ponto importante, é não replicar essas funções dentro da pilha de código hypervisor, ou seja, todos os avanços no *Linux* como sistema operacional aplicam-se também para a virtualização, visto que a mesma faz uso dessas funções, tais como:

- Escalonamento.
- Controle de recursos.
- Gerenciamento de memória.
- Armazenamento.
- Suporte a *hardware*.
- Segurança.

O KVM está presente nativamente em todos os sistemas operacionais *Red Hat* e *CentOS*. Como já citado previamente a comunidade em torno do KVM é *Open Virtualization Alliance* (OVA) tendo como principais colaboradores *HP, IBM, Intel, Net App* e *Redhat*.

O KVM possui um bom nível de maturidade inclusive sendo adotado por diversos provedores de nuvem, entre os mais populares temos *Digital Ocean*[23] *e Rackspace*[24], mas não se resume apenas a esses dois. É componente do catálogo de serviços da empresa Red Hat (2016) e de acordo com OVA[25], análises mostram que KVM pode ser de 60 a 90% mais barato do que outras soluções, enquanto oferece a mesma funcionalidade. Em relação ao *Xen* Hipervisor, bastante popular em provedores de nuvem, segundo Graziano (2011), o KVM possui desempenho e isolamento de recursos bastante superiores. O KVM é capaz de proporcionar isolamento de desempenho muito superior entre máquinas virtuais e não requer modificações para o sistema operacional convidado.

5 EXPERIMENTOS

Nossa análise teve como objetivo identificar qual das tecnologias avaliadas possui melhor desempenho nas operações de disco. Para que isso fosse possível foi necessário configurar

[23] <https://www.digitalocean.com/help/technical/general/>
[24] <https://wiki.openstack.org/wiki/HypervisorSupportMatrix>
[25] <https://openvirtualizationalliance.org/what-kvm/benefits-kvm>

um ambiente de testes que nos possibilitasse criar MVs e contêineres de maneira simples e rápida. Tendo em vista que foi avaliado também o impacto do aumento do número de MVs e contêineres no hospedeiro durante a execução dos seus respectivos testes, fez-se necessário a utilização de uma ferramenta que agilizasse esse processo de escalabilidade para o ambiente KVM, uma vez que o gerenciamento de contêineres com o *Docker* é bastante simples não foi necessário a utilização de nenhuma ferramenta complementar.

5.1 Ambiente

Executamos todos os testes em um computador *Lenovo ThinkCentre* M91p 4518 com processador Core i5 2400 3.1 GHz com suporte a virtualização assistida por hardware, 8 GB de memória ram e 1 TB de disco com sistema operacional *ubuntu 14.04 LTS*.

Para automatizar a implantação do ambiente utilizamos utilizamos o *Vagrant*[26] ferramenta desenvolvida para criar e gerenciar de maneira facilitada MVs, a mesma oferece uma linha de comando super simples para o gerenciamento de MVs, integração com diversos hipervisores, como VMware, VirtualBox, Hyper-V e KVM através da utilização do *plugin* do *libvirt*.

O *Vagrant* permite fazer a definição da configuração de uma ou mais MVs utilizando-se sua linguagem de domínio específico *domain-specific language* (DSL)[27]. É uma linguagem que permiti definir as configurações de uma MV e, independentemente do hipervisor que o *Vagrant* seja executado, sempre produzirá uma MV com a mesma configuração.

O *Vagrant* integra-se com diversos hipervisores através de *plugins*. No nosso caso o *plugin* do *libvirt* que é *Open source* e está disponível no seu repositório no *GitHub*[28]. A integração de ambos possui um bom nível de maturidade como é possível conferir no repositório supracitado. Com a integração de ambas as ferramentas é possível gerenciar todos os recursos do KVM sem restrições, a não ser alguma que por ventura o *libvirt* possua.

Para criação e gerenciamento dos contêineres foi utilizada somente a linha de comando do *Docker* que por ser bastante simples, robusta e intuitiva dispensando o uso de ferramentas complementares.

O *iozone*[29] foi adotado para realização das análises, iozone é uma ferramenta para análise de desempenho de sistemas de arquivos, possui suporte para a maioria dos sistemas operacionais atuais. Dentre as operações que o iozone permite avaliar estão: escrita, reescrita, leitura, releitura, leitura e escrita randômicas.

Para cada operação avaliada foram executados 100 testes para cada um dos tamanhos

[26] <https://www.vagrantup.com/>
[27] <http://blog.scottlowe.org/2014/09/12/a-quick-introduction-to-vagrant/>
[28] <https://github.com/vagrant-libvirt/vagrant-libvirt>
[29] <http://iozone.org/>

de gravação. o intervalo de confiança definido para os testes foi de 95 %.

Após definição das ferramentas para provisionar o ambiente das análises pode-se então definir os cenários, fatores e níveis como mostrado na seção a seguir.

5.2 Cenários

Para que fosse possível realizar uma avaliação relevante sobre desempenho de disco das tecnologias por nós avaliadas, foram definidos 5 cenários com variação de alguns fatores e níveis objetivando ter uma análise ampla e confiável. A métrica avaliada foi a saída em *bytes/s* para cada uma das operações: escrita, reescrita, leitura, releitura, escrita e leitura randômicas. Os fatores definidos para a análise foram: quantidade de arquivos e a quantidade de contêineres e VMs. No primeiro cenário foram utilizados todos os tamanhos de arquivos que contidos na tabela 2, enquanto que do segundo ao quinto cenário variou-se o número de contêineres e VMs como mostra a tabela 1, utilizando apenas um tamanho de arquivo 4096 *bytes* e seus respectivos tamanho de gravação.

A tabela 1 contém as informações dos cenários por nós definidos. Cada um dos cenários possui uma quantidade de contêineres e VMS e a quantidade de arquivos. O cenário 1 é o único onde o hospedeiro foi avaliado, no restante sua avaliação não se aplica (N/A).

Tabela 1 – Cenários e fatores

Cenários	Docker	KVM	Hospedeiro (nativo)	Qtd. de arquivos
1º cenário	1 contêiner	1 VM	1	6
2º cenário	2 contêineres	2 VMs	N/A	1
3º cenário	4 contêineres	4 VMs	N/A	1
4º cenário	6 contêineres	6 VMs	N/A	1
5º cenário	8 contêineres	8 VMs	N/A	1

Para definição do tamanho dos arquivos a serem utilizados na análise foram utilizados estudos prévios como Tarasov et al. (2011) onde foi proposta uma análise comparativa de sistema de arquivos em várias dimensões, assim com um apanhado da ampla gama de ferramentas e técnicas de uso mais recorrente em análises de desempenho de sistemas de arquivos. Utilizamos também tamanhos baseados nos tamanhos de bloco utilizados por aplicações como banco de dados mysql e seus motores de armazenamento *(storage engine) innodb*[30] e *myisam*[31].

A seguir temos a tabela 2 com os tamanhos dos arquivos e os seus respectivos tamanhos de gravação *(record legth)*, ambos estão representados em kbytes. Tamanho de gravação específica o tamanho do bloco a ser utilizado na operação que está sendo realizada, seja qual for a operação. Por exemplo para um arquivo de tamanho 256 *kbytes* serão escritos blocos de 4 *kbytes*.

[30] <http://dev.mysql.com/doc/refman/5.7/en/key-cache-block-size.html>
[31] <https://dev.mysql.com/doc/refman/5.5/en/optimizing-innodb-diskio.html>

Tabela 2 – Tamanhos dos arquivos da análise

Tamanho do arquivo em *kbytes*	Tamanho da gravação *(record length)*
256	64
	256
512	128
	512
1024	256
	1024
2048	512
	2048
4096	1024
	4096
8192	2048
	8192

6 ANÁLISE DOS RESULTADOS

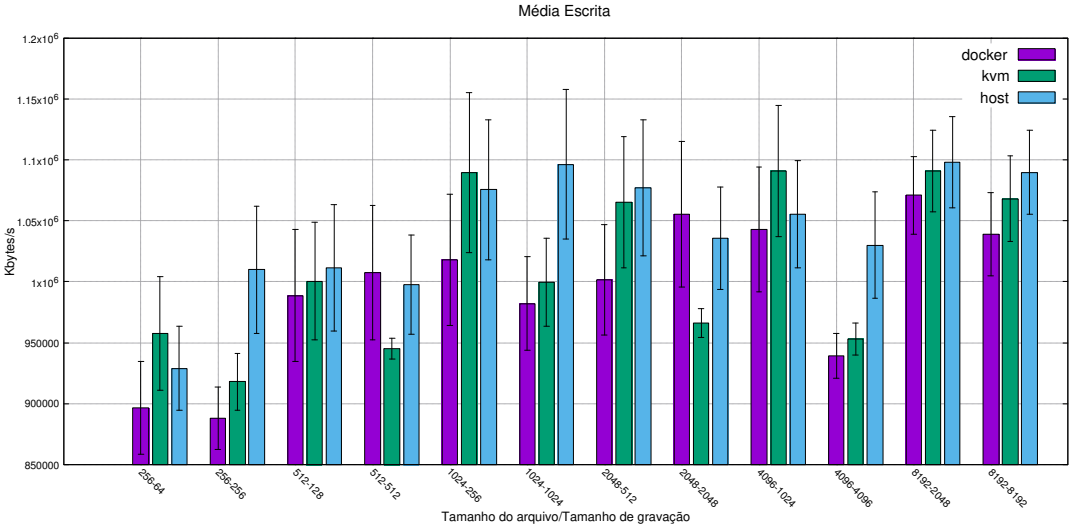
A avaliação está dividida em duas partes principais, primeiro cenário onde foram realizados testes de escrita, reescrita, leitura, releitura, escrita e leitura randômicas para cada um dos tamanhos de arquivos e seus respectivos tamanho de gravação como mostrado na tabela 2. Em seguida foram gerados os gráficos com o comparativo de desempenho entre hospedeiro (nativo), *Docker* e KVM.

Do segundo cenário em diante foram comparados apenas os desempenhos de *Docker* e KVM, utilizando-se apenas o arquivo de tamanho 4096 kbytes e seus respectivos comprimentos de gravação. Como citado previamente esse tamanho de arquivo foi adotado com base nos motores de armazenamento do *mysql*.

6.1 Cenário - 01

Na figura 9, podemos visualizar o gráfico de desempenho de escrita das tecnologias *Docker*, KVM e nativo *(host)*. Em relação ao *Docker* o KVM obteve melhor desempenho de escrita em mais da metade dos casos, o *host* teve desempenho melhor do que *Docker* e KVM em aproximadamente metade dos casos, como podemos notar, por exemplo, no arquivo de tamanho 1024 *kbytes* com tamanho de gravação 1024 *kbytes* onde o hospedeiro possui certa vantagem em relação ao desempenho das ferramentas de virtualização.

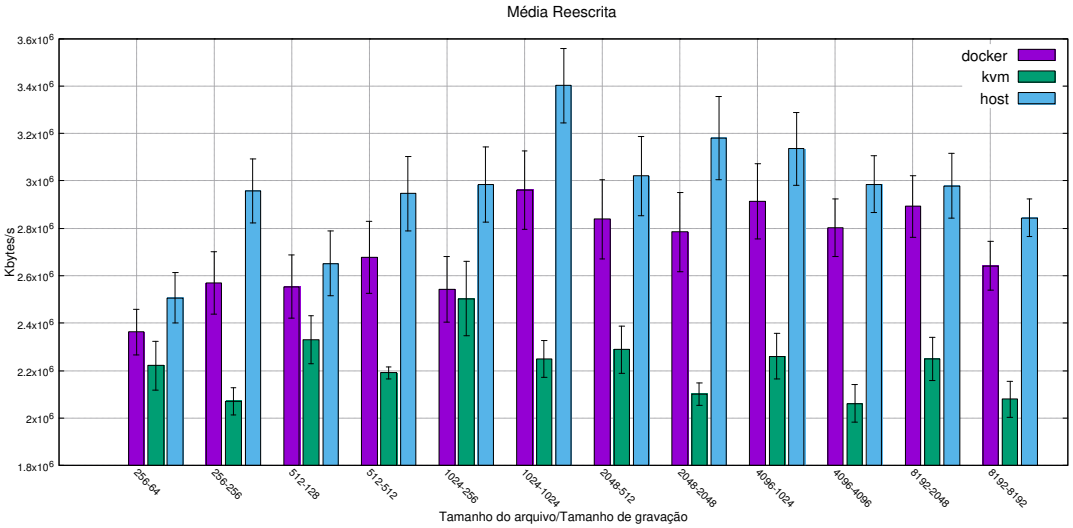
Figura 9 – Teste de escrita com 6 tamanhos de arquivos



Fonte: Autor

A figura 10, contém o gráfico para a operação de reescrita. O desempenho do *Docker* é aproximado do desempenho nativo em alguns casos. O KVM mostrou ter desempenho inferior ao *Docker* e hospedeiro para essa operação em todos os tamanhos de arquivos e gravação testados.

Figura 10 – Teste de reescrita com 6 tamanhos de arquivos

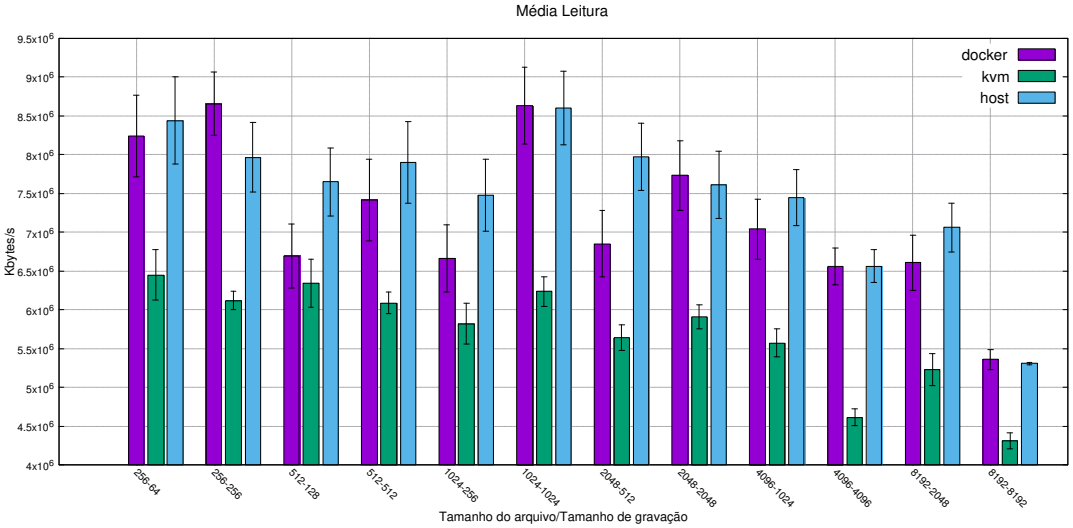


Fonte: Autor

Na figura 11, é possível visualizar o gráfico para o teste de leitura. Pode-se observar que o desempenho do *Docker* é bastante semelhante ao desempenho nativo, para alguns tamanhos

de gravação chega a ser um pouco melhor. Já o KVM tem desempenho bem abaixo do *Docker* e nativo para praticamente todos os tamanhos de arquivos e gravação.

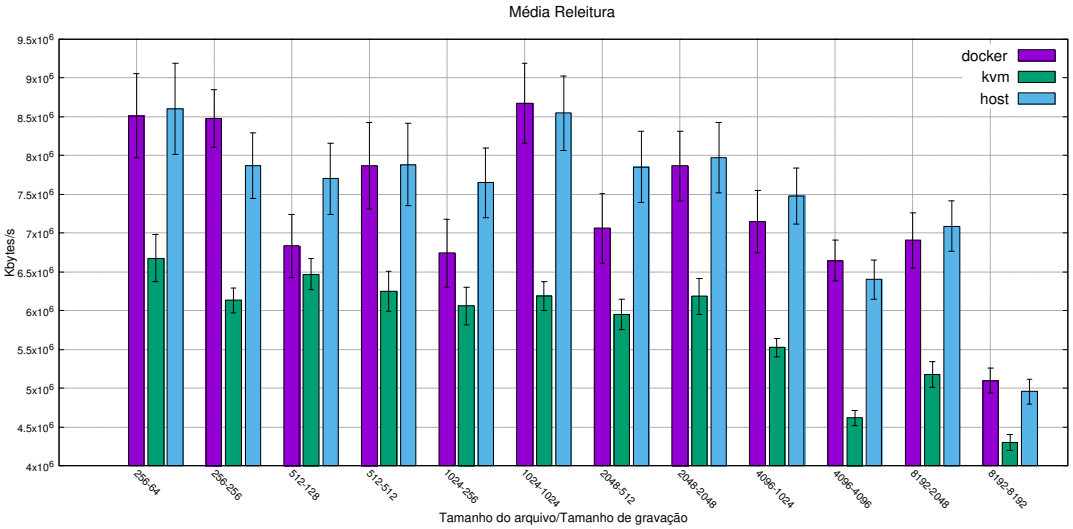
Figura 11 – Teste de leitura com 6 tamanhos de arquivos



Fonte: Autor

Na figura 12, podemos visualizar o gráfico de releitura. Que possui bastante similaridade com o desempenho do teste de leitura onde o *Docker* possui desempenho semelhante ao nativo e o KVM bem abaixo de ambos para esta operação.

Figura 12 – Teste de releitura com 6 tamanhos de arquivos

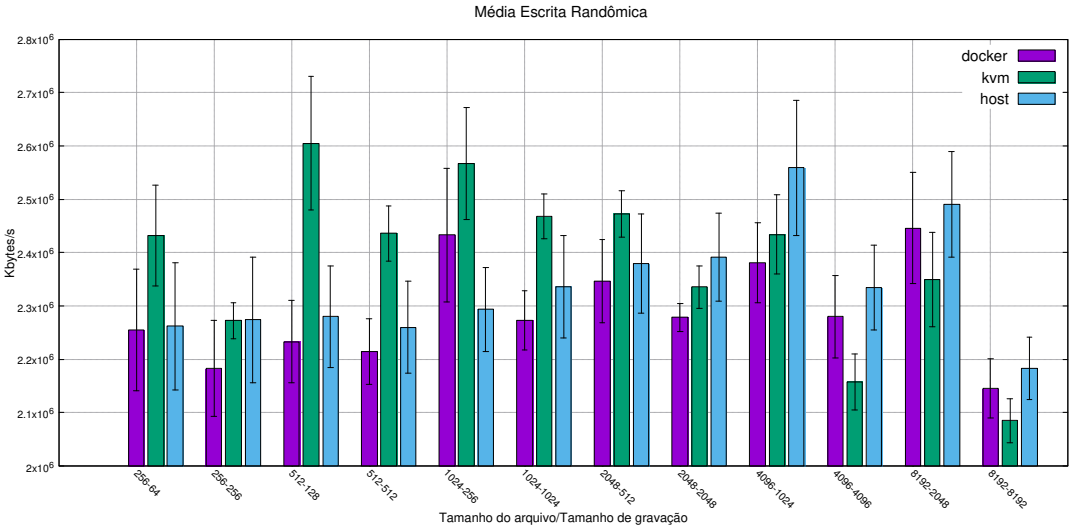


Fonte: Autor

Na figura 13, podemos visualizar o desempenho de escrita randômica, onde o KVM

possui desempenho superior ao *Docker* em sete dos doze casos e seis em relação ao nativo. Contudo a medida em que o tamanho dos arquivos e gravação aumentam o desempenho do mesmo vai se equiparando e até mesmo sendo superado pelo desempenho nativo e do *Docker*.

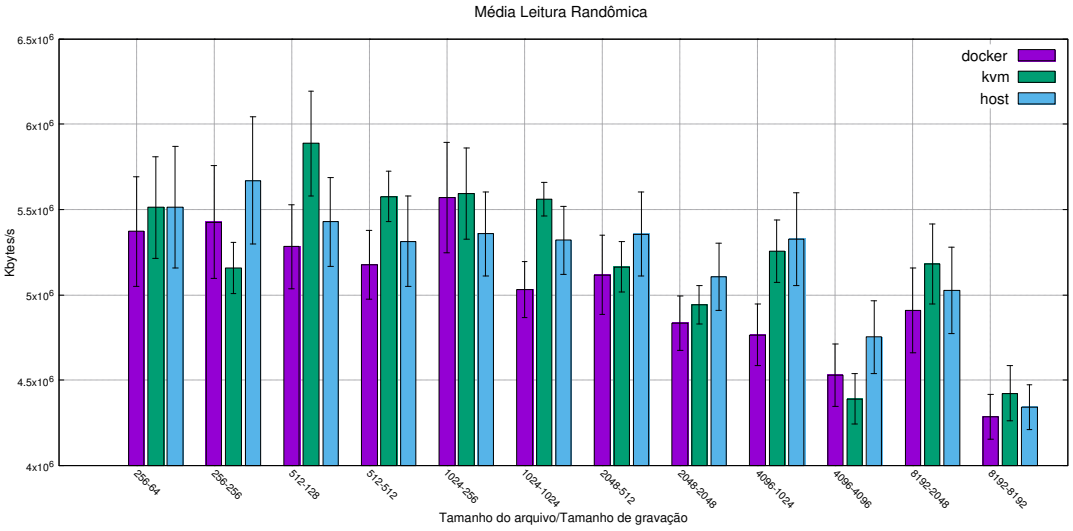
Figura 13 – Teste de escrita randômica com 6 tamanhos de arquivos



Fonte: Autor

Na figura 14, podemos observar o gráfico de leitura randômica. O desempenho do *Docker*, KVM e nativo é bem parecido para a maioria dos tamanhos de arquivo e gravação.

Figura 14 – Teste de leitura randômica com 6 tamanhos de arquivos

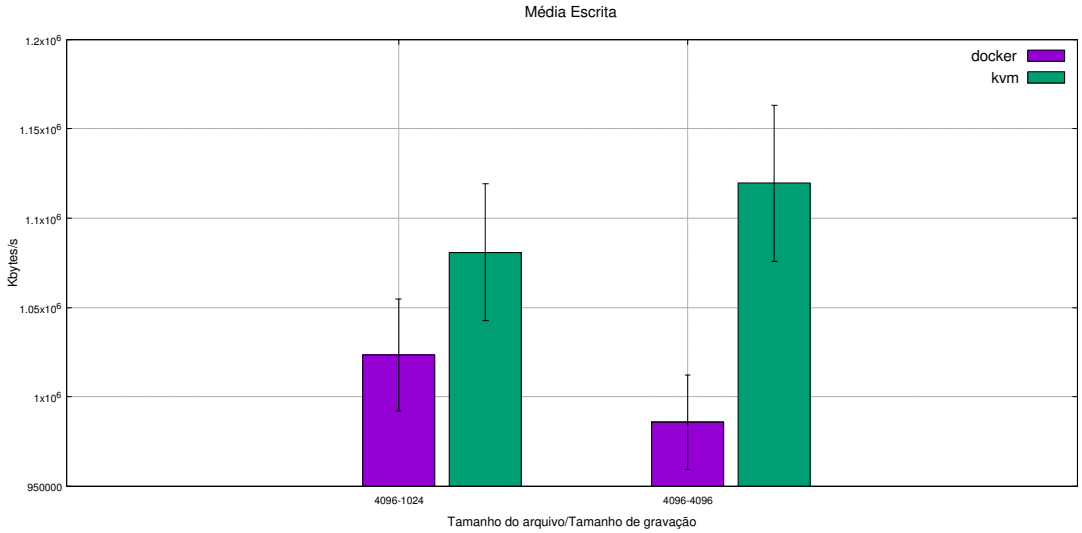


Fonte: Autor

6.2 Cenário 02

Na figura 15, podemos visualizar o gráfico para a operação de escrita de arquivos com tamanho de 4096 *kbytes*, o KVM possui um desempenho consideravelmente melhor do que o *Docker* para ambos os tamanhos de gravação.

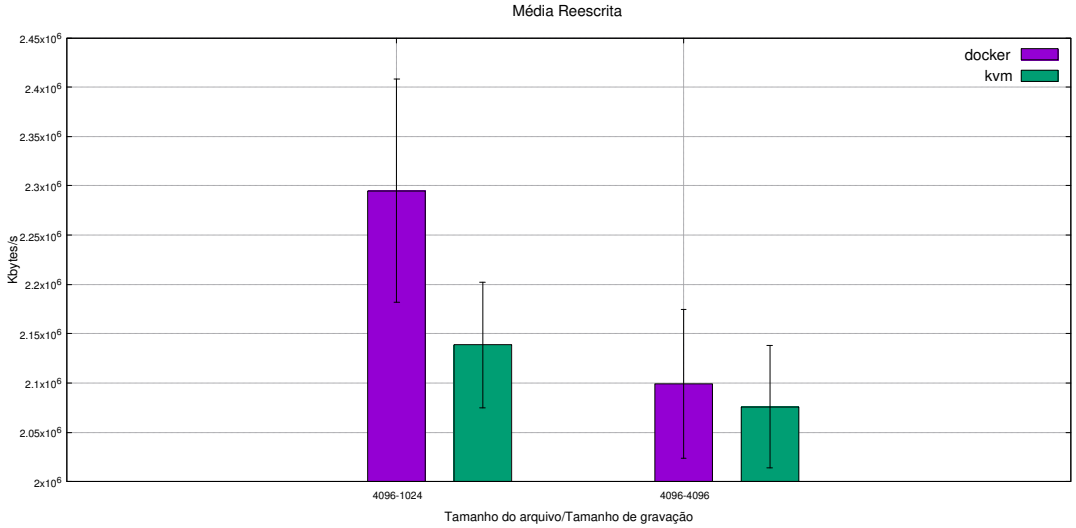
Figura 15 – Teste de escrita com 1 tamanho de arquivo



Fonte: Autor

Na figura 16, podemos visualizar o gráfico para a operação de reescrita, em que arquivos com tamanho de gravação de 1024 *kbytes* o *Docker* possui melhor desempenho do que o KVM, já para o tamanho de gravação 4096 *kbytes* ambos possuem desempenho praticamente equivalente.

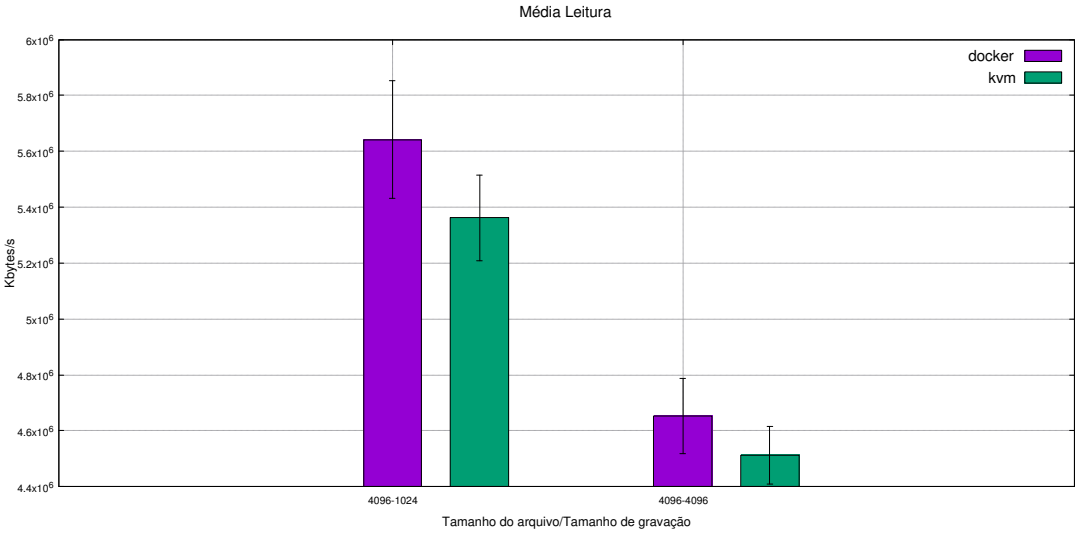
Figura 16 – Teste de reescrita com 1 tamanho de arquivo



Fonte: Autor

Na figura 17, podemos visualizar o gráfico para a operação de leitura. O *Docker* possui desempenho um pouco superior ao KVM em ambos os tamanhos de gravação para este teste.

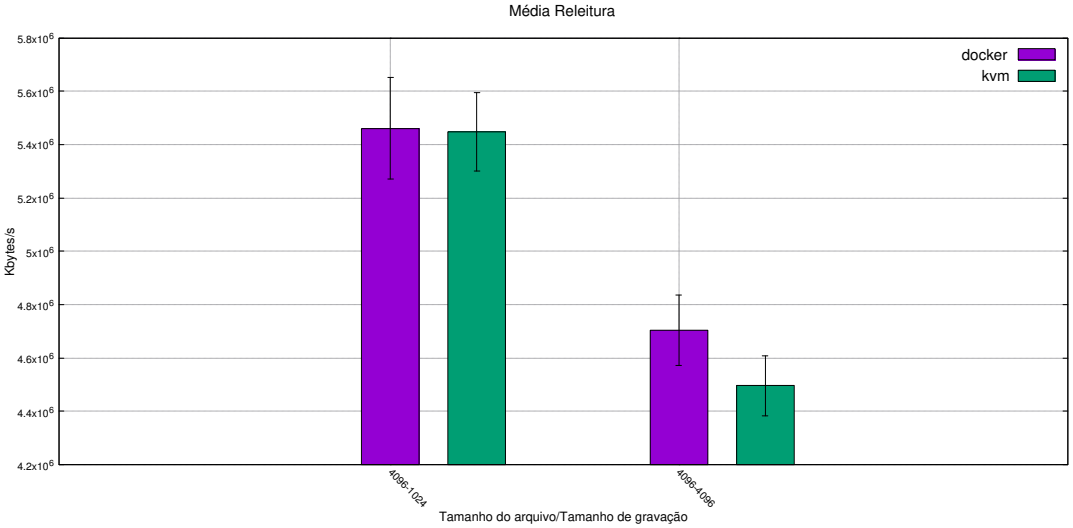
Figura 17 – Teste de leitura com 1 tamanho de arquivo



Fonte: Autor

Na figura 18, podemos visualizar o gráfico da operação de releitura. Com tamanho de gravação de 1024 *kbytes Docker* e KVM possuem desempenhos praticamente idênticos, já para o tamanho de gravação de 4096 *kbytes* o *Docker* tem desempenho levemente superior.

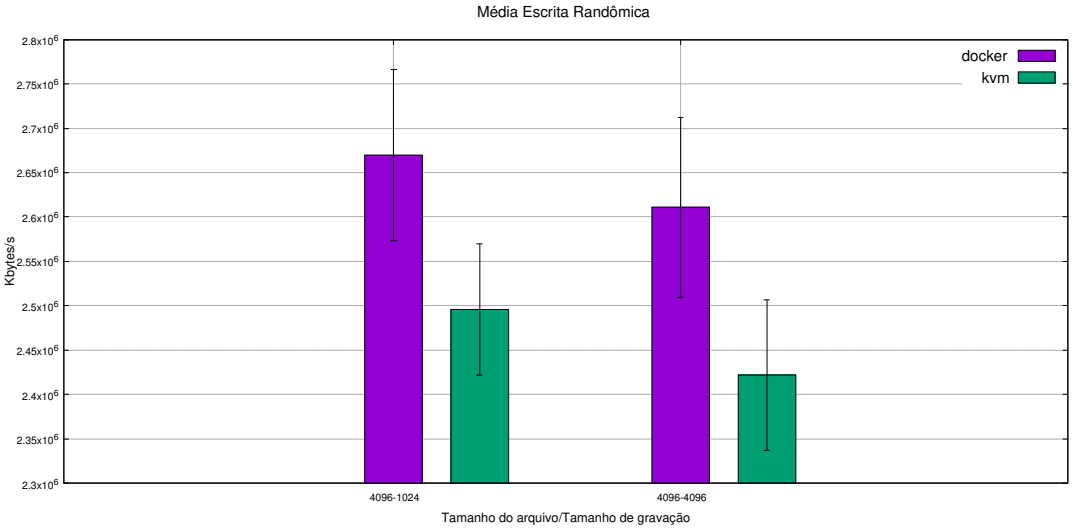
Figura 18 – Teste de releitura com 1 tamanho de arquivo



Fonte: Autor

Na figura 19, podemos visualizar o gráfico da operação escrita randômia. O *Docker* tem desempenho superior ao KVM para ambos os tamanhos de gravação.

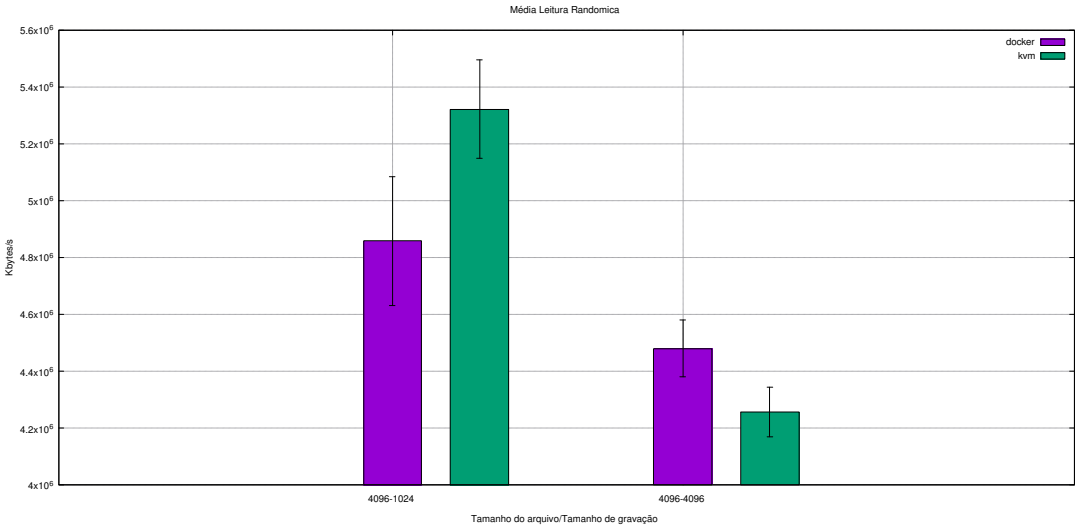
Figura 19 – Teste de escrita randômica com 1 tamanho de arquivo



Fonte: Autor

Na figura 20, podemos visualizar o gráfico da operação de escrita randômica. O desempenho do *Docker* para esta operação utilizando tamanho de gravação de 1024 *kbytes* foi superior ao KVM, que por sua vez foi superior com tamanho de gravação de 4096 *kbytes*.

Figura 20 – Teste de leitura randômica com 1 tamanho de arquivo

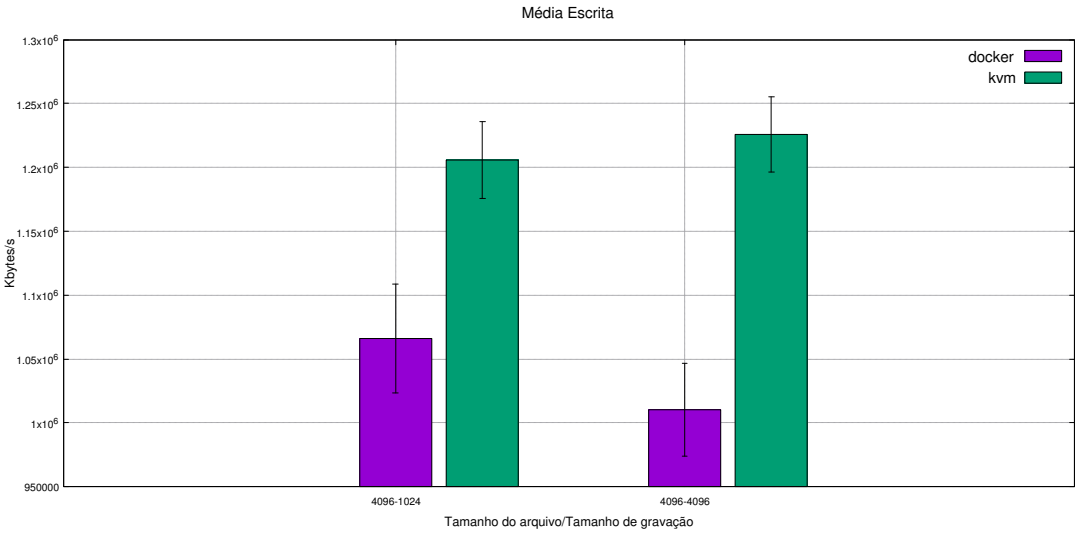


Fonte: Autor

6.3 Cenário 03

Na figura 21, podemos visualizar o gráfico para a operação de escrita, onde o KVM obteve melhor desempenho do que o *Docker* para ambos os tamanhos de gravação.

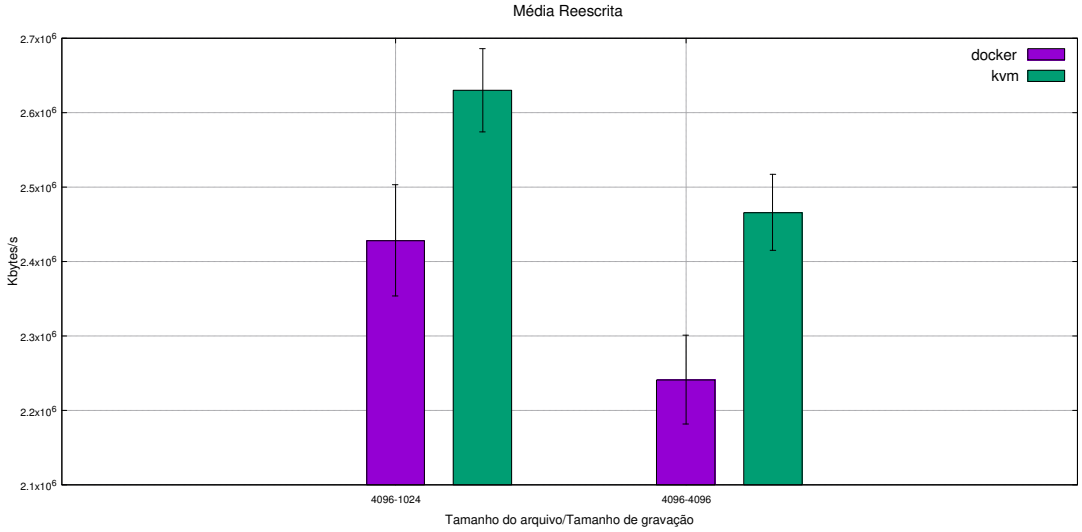
Figura 21 – Teste de escrita com 1 tamanho de arquivo



Fonte: Autor

Na figura 22, podemos visualizar o gráfico de reescrita. O KVM obteve melhor desempenho do que o *Docker* para ambos os casos.

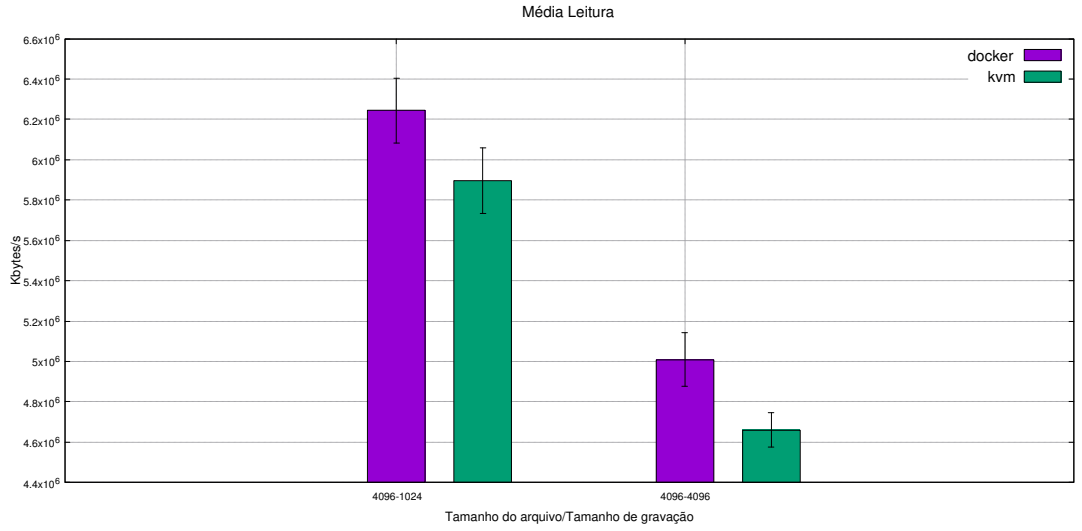
Figura 22 – Teste de reescrita com 1 tamanho de arquivo



Fonte: Autor

Na figura 23, podemos visualizar o gráfico para a operação de leitura. Onde o *Docker* conseguiu melhor desempenho do que o KVM.

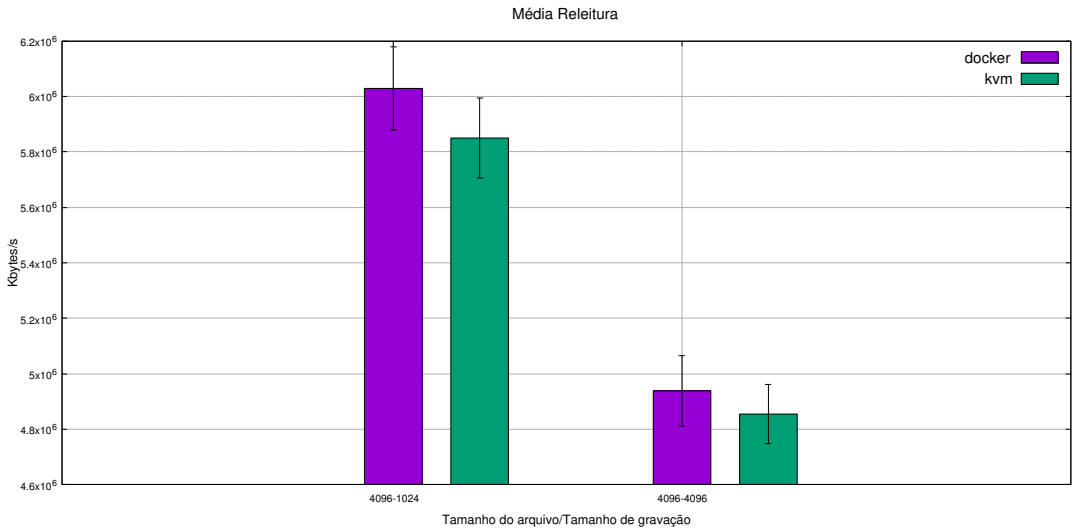
Figura 23 – Teste de leitura com 1 tamanho de arquivo



Fonte: Autor

Na figura 24, podemos visualizar o gráfico da operação de releitura. O *Docker* teve melhor desempenho do que o KVM nesse teste.

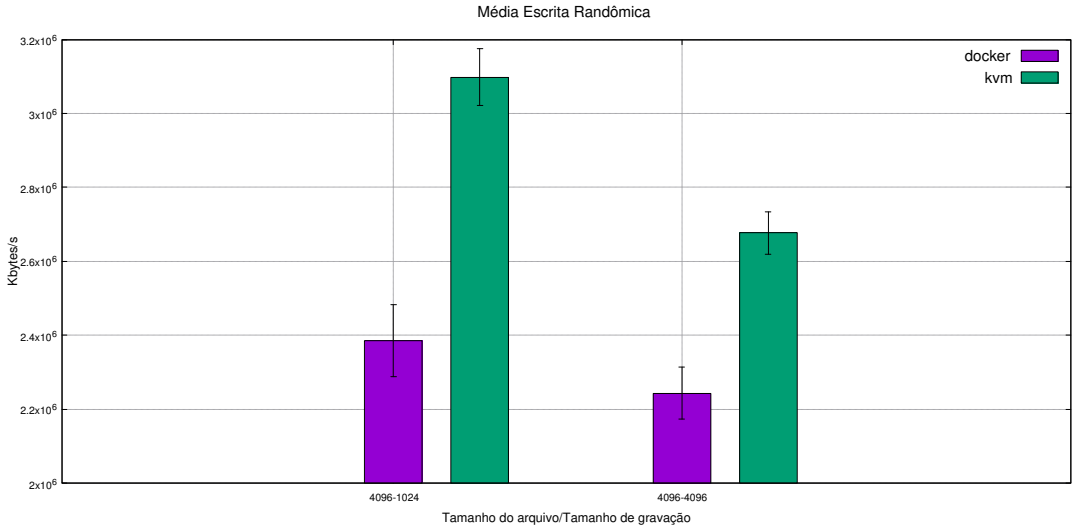
Figura 24 – Teste de releitura com 1 tamanho de arquivo



Fonte: Autor

Na figura 25, podemos visualizar o gráfico da operação de escrita randômica. O KVM teve desempenho superior ao *Docker* para ambos os tamanhos de gravação.

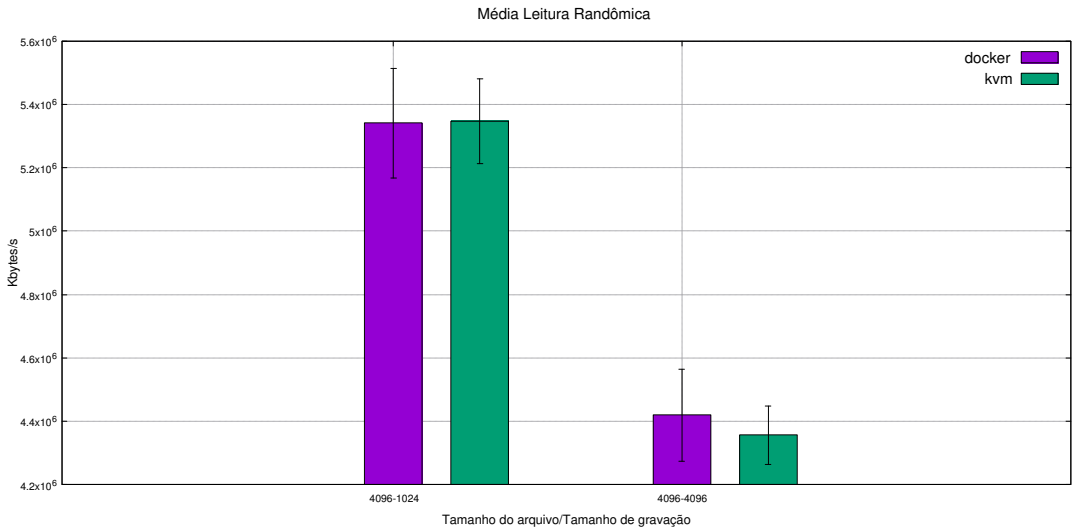
Figura 25 – Teste de escrita randômica com 1 tamanho de arquivo



Fonte: Autor

Na figura 26, podemos visualizar o gráfico da operação de leitura randômica. O *Docker* obteve desempenho praticamente equivalente ao KVM para o tamanho de gravação de 1024 *kbytes*, já para o tamanho de gravação de 4096 *kbytes* o *Docker* tem pequena vantagem.

Figura 26 – Teste de leitura randômica com 1 tamanho de arquivo

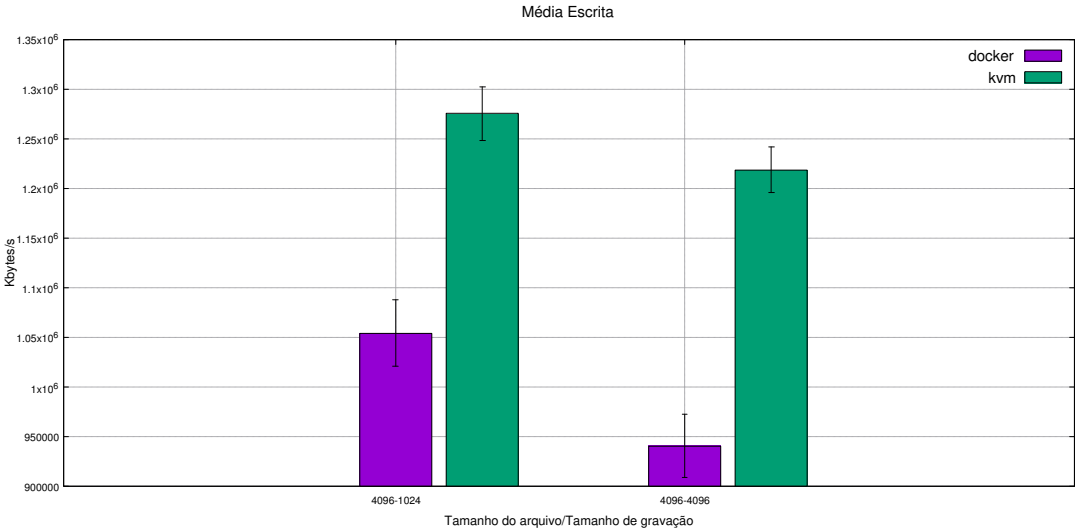


Fonte: Autor

6.4 Cenário 04

Na figura 27, podemos visualizar o gráfico de desempenho da operação de escrita. O KVM teve desempenho superior ao *Docker*, principalmente para o tamanho de gravação de 4096 *kbytes*, onde a diferença de desempenho foi maior.

Figura 27 – Teste de escrita com 1 tamanho de arquivo

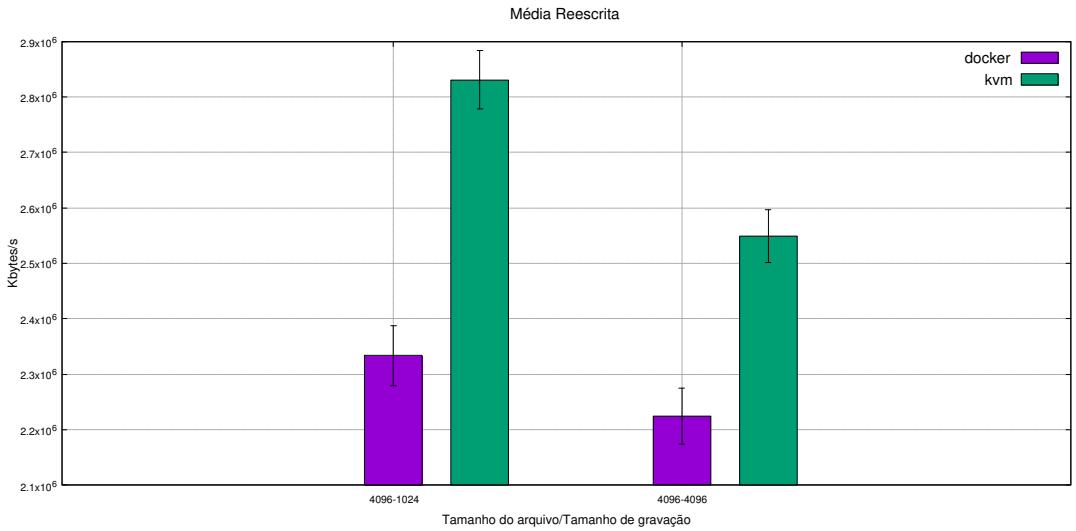


Fonte: Autor

Na figura 28, podemos visualizar o gráfico da operação de reescrita, onde o desempenho

do KVM é superior ao desempenho do *Docker* de maneira acentuada, para ambos os tamanhos de gravação.

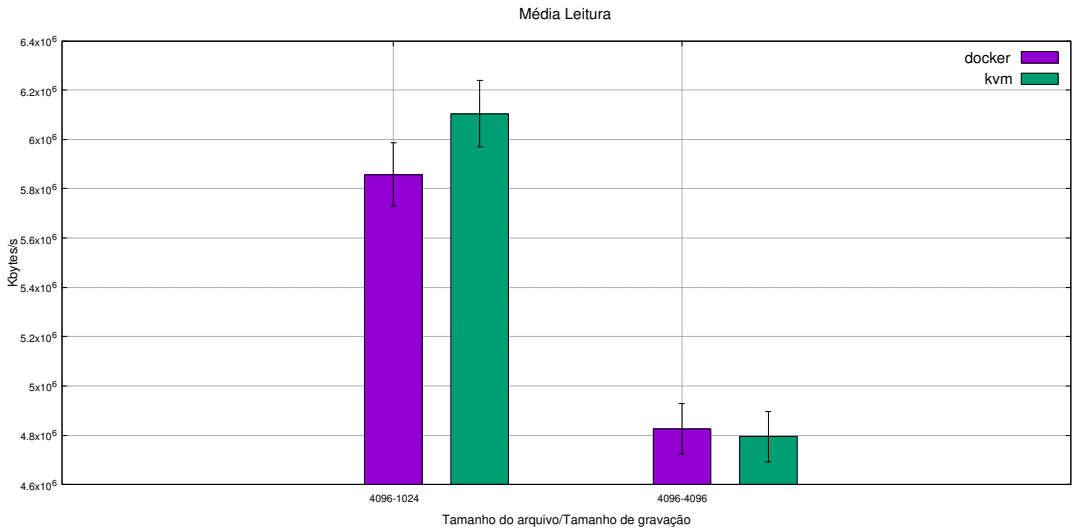
Figura 28 – Teste de reescrita com 1 tamanho de arquivo



Fonte: Autor

Na figura 29, podemos visualizar o gráfico para a operação de leitura. Com tamanho de gravação de 1024 *kbytes* o *Docker* obteve desempenho pouco inferior ao do KVM, assim como o KVM teve desempenho inferior ao *Docker* com tamanho de gravação de 4096 *kbytes*.

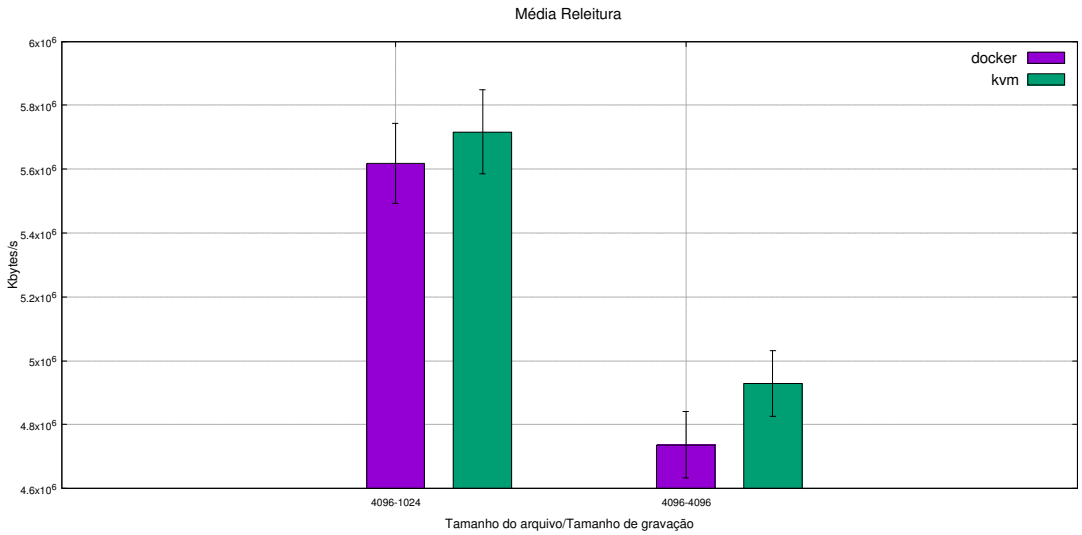
Figura 29 – Teste de leitura com 1 tamanho de arquivo



Fonte: Autor

Na figura 30, podemos visualizar gráfico da operação de releitura. O KVM conseguiu um desempenho pouco melhor que o *Docker* para ambos os tamanhos de gravação.

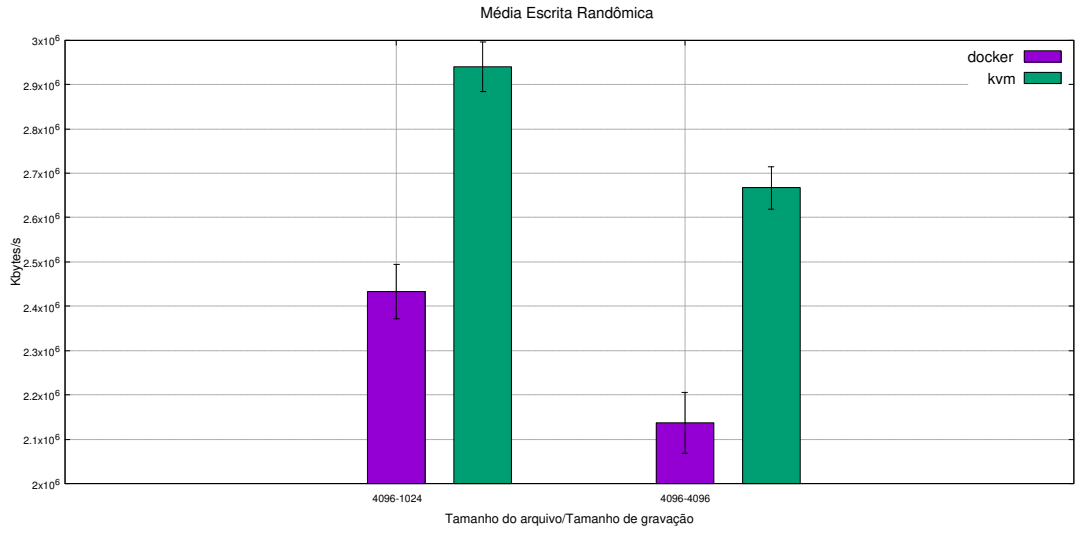
Figura 30 – Teste de releitura com 1 tamanho de arquivo



Fonte: Autor

Na figura 31, podemos visualizar o gráfico da operação de escrita randômica. Onde o desempenho do KVM é consideravelmente melhor em relação ao *Docker* para ambos tamanhos de gravação.

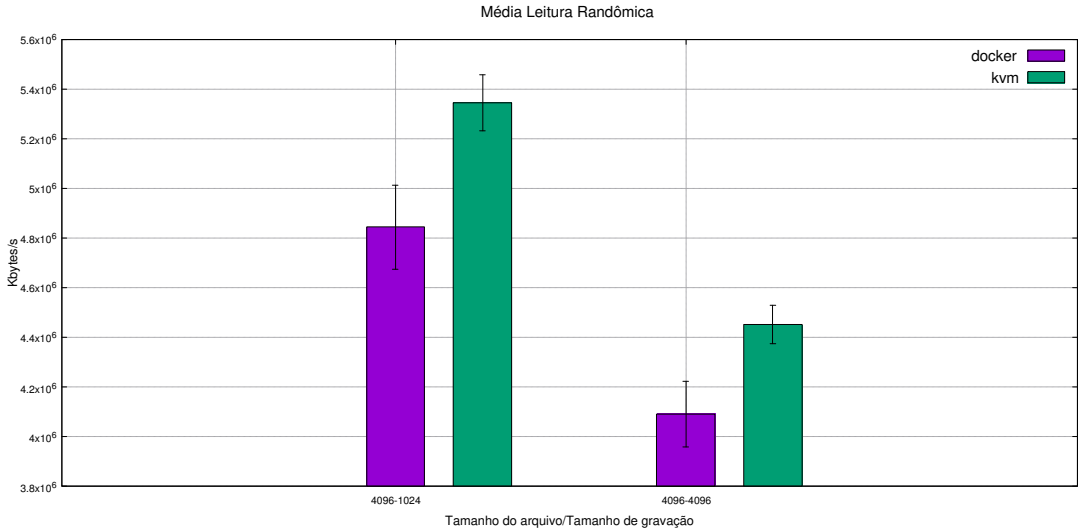
Figura 31 – Teste escrita randômica com 1 tamanho de arquivo



Fonte: Autor

Na figura 32, podemos visualizar o gráfico para a operação de leitura randômica. O desempenho do KVM foi superior ao *Docker* em ambos tamanhos de gravação a proporção de vantagem do KVM se manteve.

Figura 32 – Teste de leitura randômica com 1 tamanho de arquivo

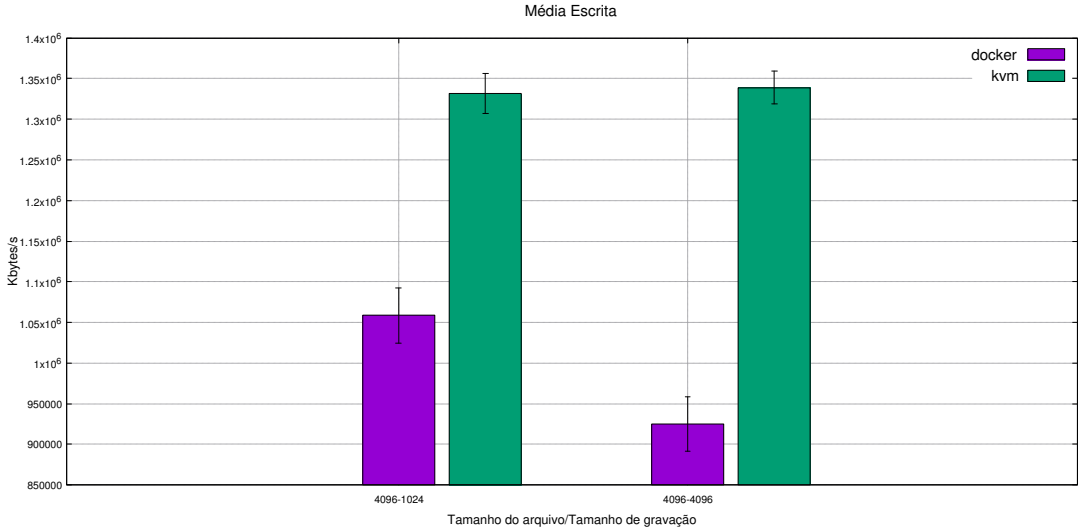


Fonte: Autor

6.5 Cenário 05

Na figura 33, podemos visualizar o gráfico da operação de escrita. O KVM foi superior ao *Docker* para ambos tamanhos de gravação, contudo para o tamanho de gravação de 4096 *kbytes*, obteve um desempenho proporcionalmente maior do que com 1024 *kbytes* de tamanho de gravação.

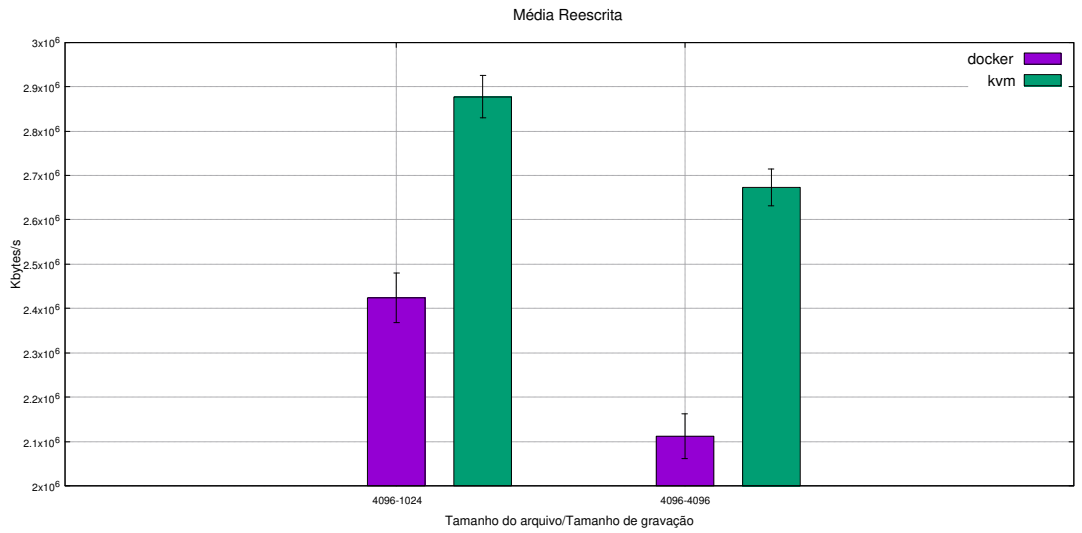
Figura 33 – Teste de escrita com 1 tamanho de arquivo



Fonte: Autor

Na figura 34, podemos visualizar o gráfico para a operação de reescrita. O desempenho do KVM foi superior ao *Docker* para ambos tamanhos de gravação, entretanto para o tamanho de gravação de 4096 *kbytes*, o *Docker* com tamanho de gravação de 4096 *kbytes* mostrou-se bastante inferior ao desempenho do KVM. A diferença de desempenho KVM e *Docker* é proporcionalmente maior com 4096 do que com 1024 *kbytes* de tamanho de gravação. Assim como na maioria dos testes do cenário cenário - 05, para operações de escrita reescrita e escrita randômica.

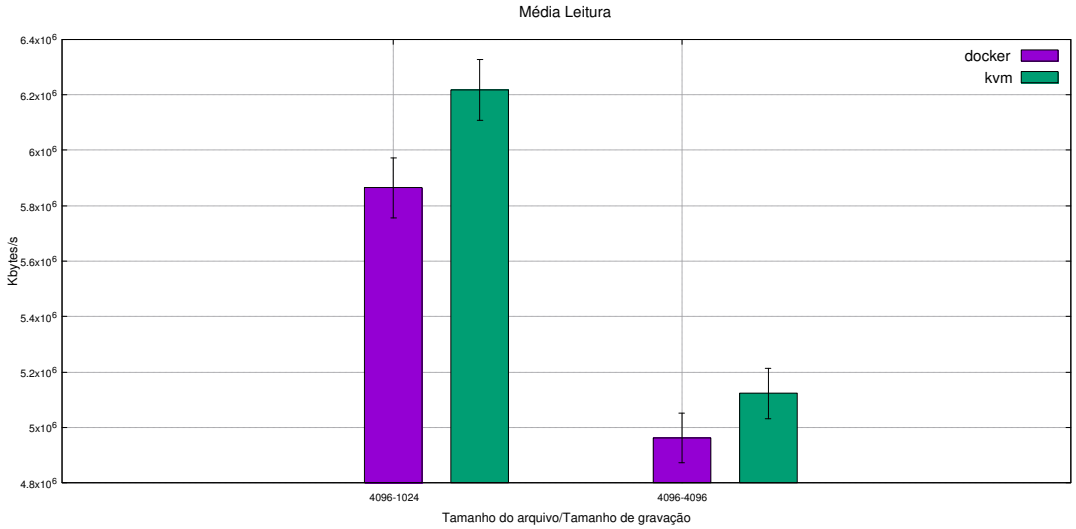
Figura 34 – Teste de reescrita com 1 tamanho de arquivo



Fonte: Autor

Na figura 35, podemos visualizar o gráfico para a operação de leitura. O desempenho de leitura do KVM foi superior ao *Docker*, todavia não possui mesma vantagem das operações de escrita, reescrita e escrita randômica.

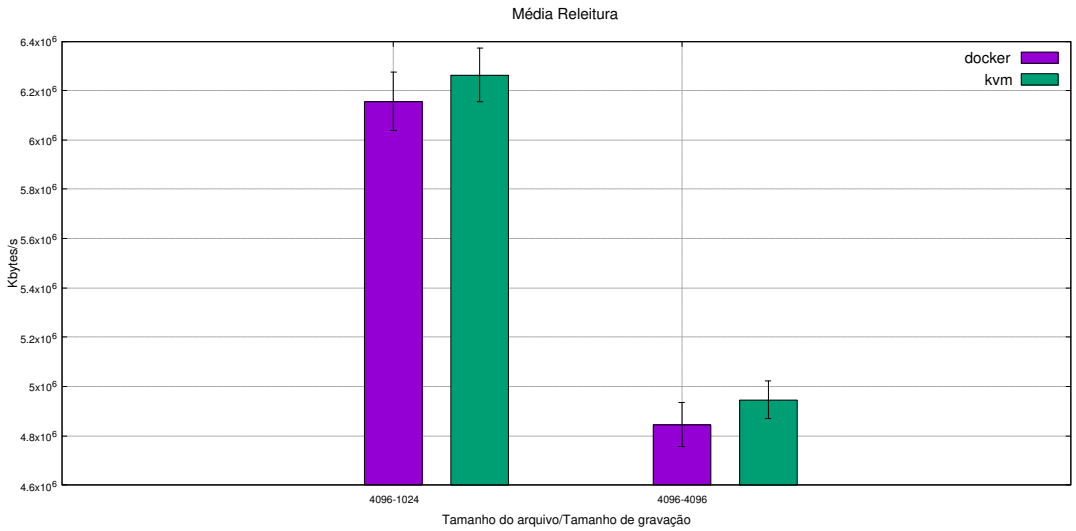
Figura 35 – Teste de leitura com 1 tamanho de arquivo



Fonte: Autor

Na figura 36, podemos visualizar o gráfico para a operação de releitura. O *Docker* conseguiu desempenho muito aproximado do KVM.

Figura 36 – Teste de releitura com 1 tamanho de arquivo

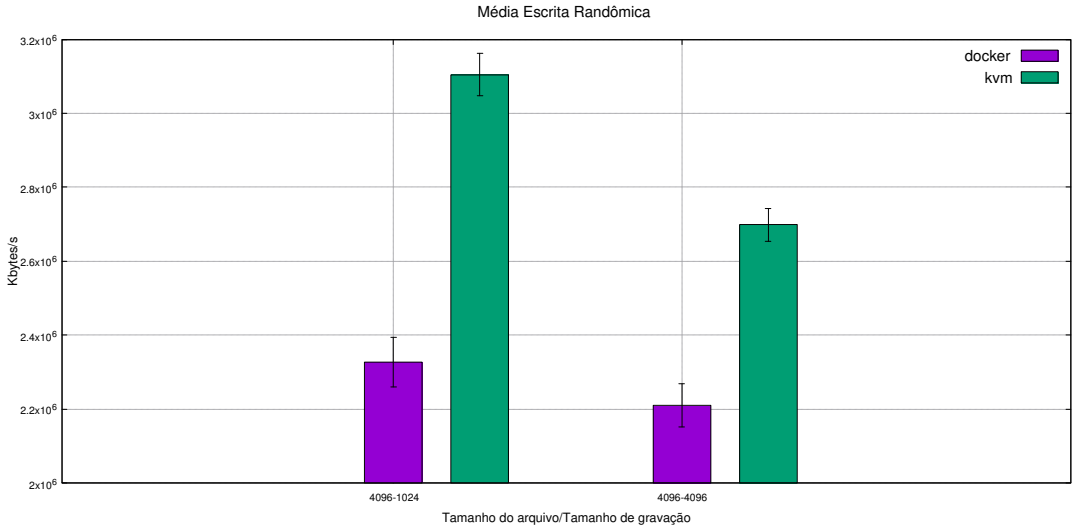


Fonte: Autor

Na figura 37, podemos visualizar o gráfico da operação de escrita randômica, como foi

comum na maioria dos testes do cenário 6.5, o desempenho de escrita do KVM mostrou-se superior ao *Docker* para ambos os tamanhos de gravação.

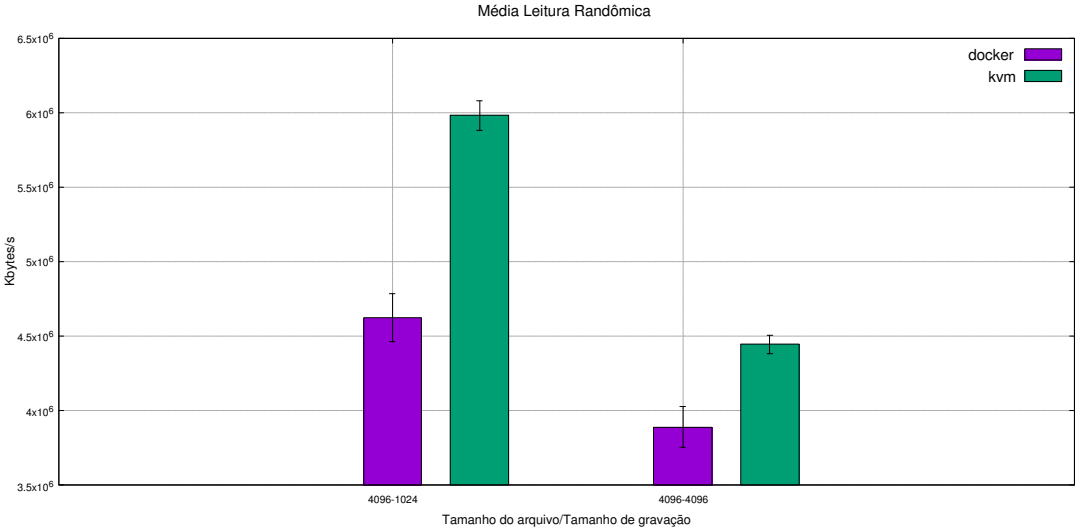
Figura 37 – Teste de escrita randômica com 1 tamanho de arquivo



Fonte: Autor

Na figura 38, podemos visualizar o gráfico para a operação de leitura randômica. O KVM foi superior ao *Docker* em ambos os tamanhos de gravação.

Figura 38 – Teste de leitura randômica com 1 tamanho de arquivo



Fonte: Autor

6.6 Consumo de CPU

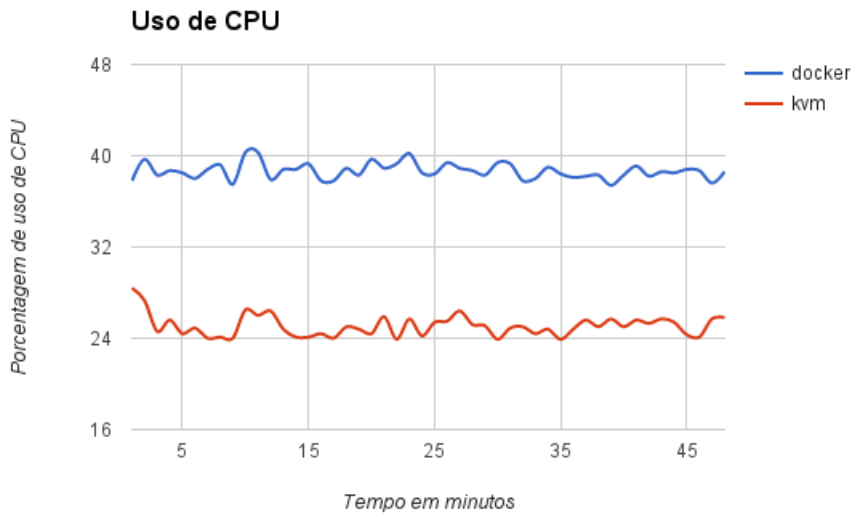
Afim de identificar quais das duas ferramentas avaliadas no presente trabalho faz mais uso do recurso de CPU do hospedeiro, utilizando o *Python psutil* monitoramos o consumo de CPU. O *Python psutil* retorna o uso da CPU em porcentagem, que salvamos em um arquivo .csv para gerar os gráficos.

Foram gerados gráficos para cada uma das ferramentas nos cenários cenario-03, cenario-04 e cenario-05. Foram removidos os valores referentes aos 15 minutos iniciais e finais para cada um dos gráficos de uso de CPU, essa remoção se fez necessário, pois o KVM tem um pico de uso de CPU considerável ao iniciar as MVs, contudo depois estabiliza, visto que o *Docker* não possui esse pico inicial removemos esses valores do gráfico, afim de equiparar as avaliações.

As Figuras 39, 40 e 41 representam o uso de CPU do hospedeiro no momento da execução das análises de desempenho dentro dos contêineres e MV respectivamente. Com seus respectivos números de contêineres e MVs, como definido na Tabela 1 de cenários e fatores.

Na figura 39, podemos visualizar o gráfico contendo a variação do uso de CPU em percentual para o cenário 03, com 4 contêineres e 4 MVs. É possível perceber que para o cenário 03 o *Docker* utilizou mais recursos de CPU do que o KVM.

Figura 39 – Uso de CPU do cenário 03

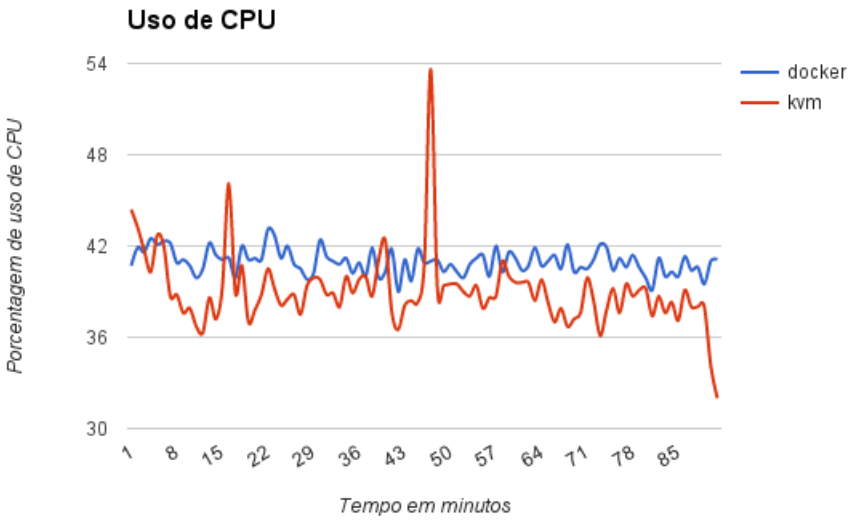


Fonte: Autor

Na figura 40, podemos visualizar o gráfico com a variação de uso de CPU no cenário 04 para as duas ferramentas avaliadas, é possível notar que a diferença de uso de recursos caiu consideravelmente, estando agora *Docker* apenas um pouco mais de recursos do que o

KVM. É possível notar que existe um pico de uso por parte do KVM, visto que os valores são coletados pelo *psutil* e capturam todo o uso de CPU durante o intervalo de um minuto e salva no .csv, esse pico não possui representatividade, considerando que na maior parte do tempo o KVM se manteve instável e esse pico pode ter sido causado pelo próprio hospedeiro durante o gerenciamento de recursos.

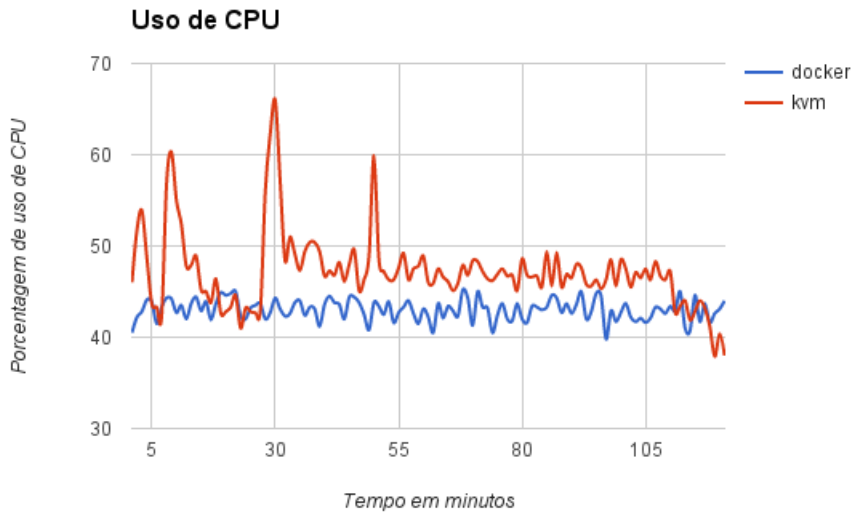
Figura 40 – Uso de CPU do cenário 04



Fonte: Autor

Na figura 41, podemos visualizar o gráfico contendo a variação do uso de CPU em porcentagem para o cenário 05. Nesse cenário o *Docker* está consumindo menos recursos do que o KVM, exceto por pequenos intervalos de tempo. É possível notar também que o *Docker* possui uso mais estável dos recursos, diferentemente do KVM que oscilou relativamente mais do que nos outros cenários.

Figura 41 – Uso de CPU do cenário 05



Fonte: Autor

6.7 Discussão

Após comparar os resultados das análises podemos perceber que o desempenho de disco para pequenos arquivos do *Docker* e KVM em linhas gerais é semelhante. Contudo, como podemos observar no cenário cenario - 01, há operações específicas em que o *Docker* é melhor, enquanto em outras o KVM possui desempenho melhor. Para as operações de reescrita, leitura e releitura o *Docker* foi melhor que o KVM e teve desempenho aproximado do nativo. Por sua vez, o KVM teve desempenho melhor que o *Docker* nas operações de escrita, escrita randômica e leitura randômica. Em relação ao desempenho nativo o KVM só conseguiu ser um pouco melhor em leitura randômica, nos outros cenários teve desempenho pior que o nativo.

A partir do cenário cenario - 02 onde avaliamos o tamanho de arquivo de 4096 *kbytes* com tamanhos de gravação de 1024 e 4096 *kbytes*. O KVM mostrou-se com melhor desempenho para operação de escrita. Enquanto o *Docker* obteve melhor desempenho em operações de leitura e releitura. Avaliamos individualmente cada figura do cenário 02 até o cenário 05 e elaboramos uma tabela com o resumo de qual das ferramenta obteve melhor desempenho para cada um das operações. A primeira coluna da tabela contém o cenário avaliado, a segunda contém a operação e a terceira contém a ferramenta que obteve melhor desempenho. Quando ambas as ferramentas estão em uma célula da terceira coluna é por que cada uma teve desempenho melhor em um tamanho de gravação diferente.

Tabela 3 – Resultados por gráfico de operação

Cenários	Gráfico da operação	Melhor desempenho
cenário 02	escrita	KVM
	reescrita	*Docker*
	leitura	*Docker*
	releitura	*Docker*
	escrita randômica	*Docker*
	leitura randômica	*Docker*/KVM
cenário 03	escrita	KVM
	reescrita	KVM
	leitura	*Docker*
	releitura	*Docker*
	escrita randômica	KVM
	leitura randômica	*Docker*
cenário 04	escrita	KVM
	reescrita	KVM
	leitura	*Docker*/KVM
	releitura	KVM
	escrita randômica	KVM
	leitura randômica	KVM
cenário 05	escrita	KVM
	reescrita	KVM
	leitura	KVM
	releitura	KVM
	escrita randômica	KVM
	leitura randômica	KVM

7 CONCLUSÃO

Este trabalhou realizou uma análise de desempenho de disco para pequenos arquivos entre as ferramentas *Docker* e KVM. Essa avaliação se fez necessária perante a importância da virtualização para a computação em nuvem e devido o pequeno número de análises prévias com foco tão específico.

Para atingir nosso objetivo, foram executados alguns passos, primeiramente buscamos dentre as ferramentas de virtualização de código aberto as mais promissoras e que fossem largamente utilizadas atualmente. Em seguida, definimos com base em pesquisa uma análise de desempenho que fosse relevante. Após escolher as ferramentas de virtualização e o tipo de análise a se realizar, definimos os cenários, fatores e níveis para as análises.

Houveram dificuldades para conseguir fazer com que contêineres e VMs utilizassem a mesma quantidade de recursos do hospedeiro, e para gerenciar de maneira facilitada a MVs do KVM, todavia esses problemas foram solucionados com estudo mais aprofundado das documentações de *Docker* e KVM, assim como o *Vagrant* resolveu o gerenciamento das MVs no

KVM.

Os resultados dos experimentos mostraram que ambas as ferramentas possuem desempenho próximo ao desempenho do *hardware* nativo quando executados sem limitação de recursos. Quando comparados os desempenhos entre *Docker* e KVM entre sí, ambos demonstraram estarem próximos na maioria dos casos, como relatamos na subceção Discussão. Contudo, constatamos que a partir do cenário Cenário 03 até o cenário Cenário 05, o KVM obteve melhor desempenho como mostrado na tabela Tabela 3. Outro fator a ser considerado é que, de acordo com que o número de contêineres e MV aumentava em seus respectivos cenários o *Docker* foi acentuando o consumo de CPU enquanto o KVM aumentava o consumo em relação ao *Docker*.

Esperava-se que o *Docker* obtivesse melhor desempenho na maioria dos casos, entretanto não foi o que se constatou. Para operações de escrita o *Docker* teve desempenho inferior ao KVM em praticamente todos os testes. Essa desvantagem deu-se por que o *Docker* utiliza por padrão o *Advanced multi-layered Unification FileSystem* (AUFS)[32] , que utiliza um sistema de camadas para criação de imagens. Uma imagem são varias camadas sobrepostas, onde somente na camada do topo é permitido escrita e leitura, todas as outras camadas adjacentes são somente leitura. Quando a operação de escrita de um arquivo é iniciada o mesmo tem que ser copiado para a camada superior integralmente e seguida poderá ser alterado.

Por fim, concluímos que ambas as tecnologias possuem desempenho semelhante, em geral, o *Docker* com melhor desempenho em operações de leitura e releitura e o KVM obtendo melhor desempenho para as operações escrita e reescrita. Como mostra a Tabela 3.

Como trabalhos futuros podemos citar uma análise de segurança e do isolamento de recursos providos pelo *Docker*. Pois em atualização recente (DOCKER, 2015), o *Docker* implementou novos mecanismos de segurança e isolamento. Com os novos recursos de segurança e isolamento é possível mitigar o ataque *Fork Bomb* que de acordo com Eric S. Raymond's (2016), é um ataque de negação de serviço onde um processo cria copias de sí mesmo indefinidamente causando lentidão e até travamento de servidores.

[32] <https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/>

REFERÊNCIAS

CHEN, A. G. G. Kvm for server virtualization: An open source solution comes of age. 2011.

DEDOIMEDO. **Dedoimedo**. [S.l.], 2016. A place to learn lot about a lot! Disponível em: <http://www.dedoimedo.com/computers/docker-guide.html>. Acesso em: 15 jun. 2016.

DOCKER. Introduction to Container Security. p. 8, 2015.

DOCKER. **Docker Documentation**. [S.l.], 2016. Disponível em: <https://docs.docker.com>. Acesso em: 18 jun. 2016.

DUA, R.; RAJA, A. R.; KAKADIA, D. Virtualization vs containerization to support paas. In: **Cloud Engineering (IC2E), 2014 IEEE International Conference on**. [S.l.: s.n.], 2014. p. 610–614.

ERIC S. RAYMOND'S. **fork bomb**. [S.l.], 2016. Disponível em: <http://catb.org/~esr/jargon/html/F/fork-bomb.html>. Acesso em: 21 jul. 2016.

FELTER, W.; FERREIRA, A.; RAJAMONY, R.; RUBIO, J. An Updated Performance Comparison of Virtual Machines and Linux Containers. **Technology**, v. 25482, p. 171–172, 2014.

GILLEN, A. WHITE PAPER KVM for Server Virtualization : An Open Source Solution Comes of Age. 2011.

GRAZIANO, C. A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project. **Master's Thesis**, 2011.

INTEL. Enhanced Virtualization on Intel® Architecture- based Servers. **Management**, 2006.

JOY, A. M. Performance comparison between Linux containers and virtual machines. **Conference Proceeding - 2015 International Conference on Advances in Computer Engineering and Applications, ICACEA 2015**, p. 342–346, 2015.

KADAM, N. V. A. P. R. Review on kvm hypervisor. **International Journal of Recent Technology and Engi neering (IJRTE)**, v. 3, 2014.

PAUL, B.; SASTRI, Y. Evaluation of Docker Containers Based on Hardware Utilization. p. 697–700, 2015.

RAHO MORITZ SPYRIDAKIS, A. P. M. R. D. Kvm, xen and docker: a performance analysis for arm based nfv and cloud computing. 2015.

RED HAT. **KVM Overview**. [S.l.], 2016. Disponível em: <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Tuning_and_Optimization_Guide/sect-Virtualization_Tuning_Optimization_Guide-Introduction-KVM_Overview.html>. Acesso em: 30 jun. 2016.

TARASOV, V.; BHANAGE, S.; ZADOK, E.; SELTZER, M. Benchmarking file system benchmarking: It* is* rocket science. **HotOS XIII**, p. 1–5, 2011.

VMWARE. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. **Memory**, p. 17, 2007. Disponível em: <www.vmware.com>.

WRIGHT, C. P.; ZADOK, E. Kernel korner: Unionfs: Bringing filesystems together. **Linux J.**, Belltown Media, Houston, TX, v. 2004, n. 128, p. 8–, dez. 2004. ISSN 1075-3583.

XAVIER, M. G.; NEVES, M. V.; ROSSI, F. D.; FERRETO, T. C.; LANGE, T.; De Rose, C. A. F. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In: **2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing ({PDP)}**. [S.l.: s.n.], 2013. p. 233–240.

XAVIER, M. G.; OLIVEIRA, I. C. D.; ROSSI, F. D.; PASSOS, R. D. D.; MATTEUSSI, K. J.; ROSE, C. A. D. A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds. **2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**, p. 253–260, 2015.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: State-of-the-art and research challenges. **Journal of Internet Services and Applications**, v. 1, n. 1, p. 7–18, 2010.