UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Nicolas Vanz

Virtualização e Migração de Processos em um Sistema Operacional Distribuído para Lightweight Manycores

Florianópolis
22 de maio de 2023

RESUMO

A classe de processadores *lightweight manycore* surgiu para prover um alto grau de paralelismo e eficiência energética. Contudo, o desenvolvimento de aplicações para esses processadores enfrenta diversos problemas de programabilidade provenientes de suas peculiaridades arquitetônicas. Especialmente, o gerenciamento de processos precisa mitigar problemas provenientes das pequenas memórias locais e da falta de um suporte robusto para virtualização. Nesse contexto, este trabalho visa desenvolver o suporte da migração de processos em um Sistema Operacional (SO) distribuído para *lightweight manycores* através de uma abordagem de virtualização leve baseada em contêineres. Particularmente, este trabalho está incluído no projeto Nanvix, um SO distribuído de código aberto projetado para *lightweight manycores*. Ao final deste trabalho espera-se melhorar o gerenciamento de processos no Nanvix, bem como abstrair e auxiliar o gerenciamento dos recursos do processador.

Palavras-chave: lightweight manycores. sistemas operacionais. migração de processos. virtualização. conteinerização

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

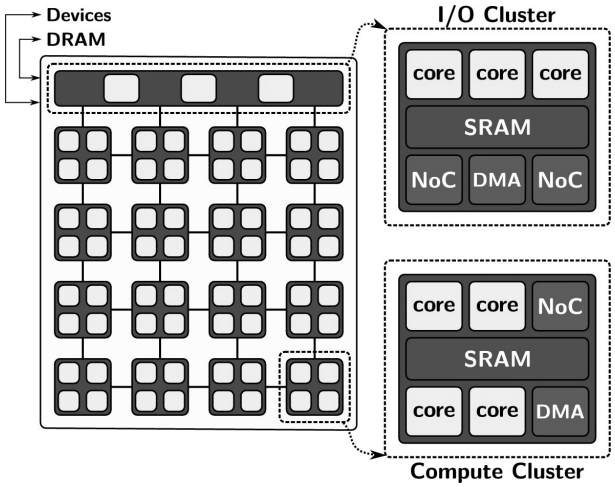
Durante muitos anos, o aumento do desempenho de sistemas computacionais esteve intrinsecamente associado ao aumento da frequência de relógio dos processadores e avanços na tecnologia dos semicondutores. Essas técnicas se mantiveram eficientes até o momento em que a dissipação de calor interna dos *chips* necessária para viabilizar o aumento da frequência atingiu um limite físico. Este fato associado com o fim iminente da lei de Moore (MOORE, 1965), fez com que a exploração de novas maneiras de aumentar o poder computacional dos sistemas se tornasse uma prioridade.

Como alternativa à limitação do aumento da frequência de relógio, processadores com múltiplos núcleos de processamento foram desenvolvidos, *aka* multicores. O desempenho dos processadores *multicore* não dependem mais diretamente das altas frequências de relógio, recorrendo ao paralelismo como principal vantagem aos processadores com um único núcleo, *aka single-cores*. Deste modo, mesmo com a estagnação da frequência de relógio nos processadores (AMROUCH et al., 2018), esse aumento na quantidade de *cores* (GEPNER; KOWALIK, 2006) em conjunto com outras melhorias no *hardware* (FULLER; MILLETT, 2011), como o aumento no número de transistores nos *chips*, aperfeiçoamento dos preditores de desvio e adaptações na hierarquia de memória, o desempenho dos sistemas computacionais continuaram a aumentar.

Atualmente, a eficiência energética dos sistemas computacionais revela-se tão importante quanto seu desempenho. Segundo o Departamento de Defesa do Governo dos Estados Unidos (DARPA/IPTO) (KOGGE et al., 2008), a potência recomendada para um supercomputador atingir o *exascale* (10^{18} *Floating-point Operations per Second* (FLOPS)), é de 20 MW, o que é inviável para a realidade dos sistemas computacionais modernos. Nesse cenário, observou-se o surgimento de uma nova classe de processadores chamada *lightweight manycore*. Esses processadores são classificados como *Multiprocessor System-on-Chip*s (MPSoCs) e têm como objetivo atrelar alto desempenho à eficiência energética (FRANCESQUINI et al., 2015). Para atingir esse objetivo, a arquitetura dos *lightweight manycores* é caracterizada por:

(i) Integrar de centenas à milhares de núcleos de processamento operando a baixas frequências em um único *chip*;
(ii) Processar cargas de trabalho *Multiple Instruction Multiple Data* (MIMD);
(iii) Organizar os núcleos em conjuntos, denominados *clusters*, para compartilhamento de recursos locais;
(iv) Utilizar *Networks-on-Chip* (NoCs) para transferência de dados entre núcleos ou *clusters*;
(v) Possuir sistemas memória distribuída restritivos, compostos por pequenas memórias locais; e
(vi) Apresentar componentes heterogêneos.

Figura 1 – Visão conceitual de um processador *lightweight manycore*



Fonte: Penna et al. (2021)

A Figura 1 ilustra uma visão conceitual da arquitetura de um *lightweight manycore*. Neste exemplo, o processador contém 1 *I/O Cluster* e 16 *Compute Clusters*. O *I/O Cluster* contém 3 núcleos, enquanto os *Compute Clusters* contêm 4 núcleos cada. Os núcleos compartilham uma mesma memória local do *cluster* e o *clusters* são interconectados por uma NoC em malha. Os processadores Kalray MPPA-256 (DINECHIN et al., 2013), PULP (ROSSI et al., 2017) e Sunway SW26010 (FU et al., 2016) são exemplos comerciais dessa classe de processadores.

Apesar dos processadores *lightweight manycores* serem uma alternativa às abordagens tradicionais no que se refere ao aumento de desempenho, as características arquiteturais introduzem severos problemas de programabilidade ao desenvolvimento de *software* de aplicações paralelas (CASTRO et al., 2016). Entre eles, podemos citar:

(i) Necessidade do uso de um modelo de programação híbrida que força troca de informação entre os *clusters* exclusivamente por troca de mensagens via NoC enquanto a comunicação interna em um *cluster* ocorre sobre memória compartilhada (KELLY; GARDNER; KYO, 2013);
(ii) Presença de um sistema de memória distribuída restritivo, formado por múltiplos espaços de endereçamento, o que exige o particionamento do conjunto de dados em blocos pequenos para manipulação nas pequenas memórias locais. A manipulação deve ocorrer um bloco de cada vez, necessitando a troca explicita de blocos com uma memória remota (CASTRO et al., 2016);
(iii) Falta de suporte de coerência de *cache* em *hardware* visando a economia de energia. Exigindo do programador a gerência da *cache* via *software* (FRANCESQUINI et al., 2015); e
(iv) Configuração heterogênea do *hardware*, como *clusters* destinados a funcionalidades específicas (computação útil e I/O), o que dificulta o desenvolvimento de aplica-

ções (BARBALACE et al., 2015).

Atualmente, estudos exploram soluções para amenizar o impacto arquiteturais sobre o desenvolvimento de *software*. Sistemas Operacionais (SOs) distribuídos destacam-se por proverem um ambiente de programação mais robusto e rico (ASMUSSEN et al., ; KLUGE; GERDES; UNGERER, ; PENNA et al., 2019). Dentre essas soluções, o modelo de um SO distribuído baseado em uma abordagem *multikernel*, o Nanvix, destaca-se por aderir a natureza distribuída e restritiva dos *lightweight manycores* (PENNA et al., 2017; PENNA et al., 2019).

Apesar do Nanvix ser uma solução promissora para o desenvolvimento de *software* em *lightweight manycores*, a maneira como o SO é estruturado atualmente ainda pode ser melhorada. O mecanismo de gerenciamento de processos dificulta a mobilidade dos processos dentro do processador, já que o processo passa todo seu ciclo de vida em um mesmo *cluster*. Isso significa que a partir do momento em que o processo inicia a execução em um *cluster*, ele fará todo seu trabalho computacional e finalizará a execução no mesmo *cluster*. Isso afeta negativamente o desempenho do sistema pois impede o remanejamento dos processos sob demanda para melhor aproveitamento dos recursos disponíveis. Por exemplo, alocar processos que se comunicam intensamente em *clusters* próximos reduz a latência de comunicação e aumenta o desempenho da aplicação (VANZ; SOUTO; CASTRO, 2022).

Neste cenário, a virtualização dos recursos do processador é importante para o suporte a multi-aplicação, melhor uso do *hardware* disponível e aumento da mobilidade dos processos (VANZ; SOUTO; CASTRO, 2022). Contudo, as características arquiteturais dos *lightweight manycores*, especialmente relacionadas à memória, inviabilizam um suporte complexo para virtualização. Por exemplo, máquinas virtuais utilizadas em ambientes *cloud* possuem à disposição centenas de GBs de memória para isolar duplicatas inteiras do SO com a ajuda de virtualização a nível de instrução (SHARMA et al., 2016). Nos *lightweight manycores*, as pequenas memórias locais e a simplificação do *hardware* para reduzir o consumo energético restringem os tipos de virtualização suportados.

Neste contexto, este trabalho explora um modelo mais leve de virtualização para *lightweight manycores* baseado no conceito de contêineres. Contêineres são executados pelo SO como aplicações virtuais e não incluem um SO convidado, resultando em um menor impacto no sistema de memória e requisitando menor complexidade do *hardware* (THALHEIM et al., 2018; SHARMA et al., 2016).

1.1 OBJETIVOS

Com base nas motivações citadas previamente, os objetivos deste trabalho serão especificados nas próximas seções.

1.1.1 Objetivo Principal

O objetivo principal deste trabalho é virtualizar os recursos internos de um *cluster* de um *lightweight manycore*. Ao desvincular os recursos locais utilizados por um processo dentro do Nanvix, um SO distribuído para *lightweight manycores*, conseguimos prover maior controle e mobilidade de processos no processador.

1.1.2 Objetivos Específicos

(i) Explorar e implementar um modelo de virtualização baseado em contêineres no Nanvix;
(ii) Explorar e implementar um modelo de migração de processos no Nanvix utilizando a abordagem de virtualização com contêineres;
(iii) Analisar o impacto do modelo de virtualização no Nanvix;
(iv) Analisar a corretude, eficiência e impacto da migração de processos através do desenvolvimento de testes e experimentos;

1.2 CONTRIBUIÇÕES

Esse trabalho de conclusão propõe o suporte à virtualização e migração de processos no Nanvix. A parte inicial deste trabalho foi publicado na Escola Regional de Alto Desempenho da Região Sul (ERAD/RS) e recebeu o prêmio Aurora Cera de melhor artigo do Fórum de Iniciação Científica (VANZ; SOUTO; CASTRO, 2022).

1.3 ORGANIZAÇÃO DO TRABALHO

Os próximos capítulos deste trabalho estão organizadas da seguinte maneira. O Capítulo 2 apresenta conceitos fundamentais para o entendimento do trabalho, tais como um detalhamento sobre virtualização, migração de processos e um aprofundamento sobre a arquitetura dos *lightweight manycores* e do Nanvix. O Capítulo 3 discute os trabalhos relacionados. O Capítulo 4 expõe a proposta deste trabalho de conclusão de curso e os detalhes de desenvolvimento da solução. O Capítulo 5 exibe como ocorreu a tomada de decisão sobre o que avaliar e como os testes foram feitos. Os resultados experimentais são exibidos e discutidos no Capítulo 6. Por fim, o Capítulo 7 apresenta as conclusões deste trabalho e pontua os próximos passos da pesquisa.

2 REFERENCIAL TEÓRICO

Neste capítulo serão apresentados conceitos fundamentais para o entendimento do trabalho. A Seção 2.1 apresenta uma visão geral da evolução dos processadores, partindo dos *single-cores* até os *lightweight manycores*. A Seção 2.2 apresenta o Nanvix, Sistema Operacional (SO) distribuído que será utilizado no desenvolvimento deste trabalho. A Seção 2.3 descreve detalhes importantes sobre a virtualização e migração de processos.

2.1 DOS *SINGLE-CORES* AOS *LIGHTWEIGHT MANYCORES*

O aumento de desempenho dos sistemas computacionais manteve-se como uma necessidade constante para o avanço da ciência em vários setores: astrologia, biologia, engenharia, etc. Até tempos atrás, esse objetivo era alcançado através do aumento da frequência de relógios do núcleo de processamento, do avanço na tecnologia dos semicondutores e do acréscimo do número de transistores em um *chip* (AMROUCH et al., 2018). Atualmente, nós já chegamos ao limite físico que impede a aplicação de parte dessas técnicas. Além da dificuldade de garantir a dissipação de calor à medida que a frequência aumenta, o número de transistores que conseguimos colocar em uma mesma área de um *chip* chegou a um limite físico, i.e., o tamanho dos transitores alcançou a escala atômica.

Como alternativa para a continuidade nos avanços de poder computacional, foram exploradas novas técnicas (FULLER; MILLETT, 2011; GEPNER; KOWALIK, 2006). Em especial, foram desenvolvidas arquiteturas paralelas, que exploram o poder de processamento paralelo, o qual é atingido pela execução de múltiplos *cores* simultaneamente (GEPNER; KOWALIK, 2006). Essas novas arquiteturas são classificadas de acordo com a maneira com que conseguem manipular os dados. São elas: (i) *Single Instruction Single Data* (SISD); (ii) *Single Instruction Multiple Data* (SIMD); (iii) *Multiple Instruction Single Data* (MISD); (iv) *Multiple Instruction Multiple Data* (MIMD). Neste trabalho, estamos interessados nas arquiteturas que suportam cargas de trabalho MIMD.

Algumas dessas arquiteturas são ilustradas pela Figura 2. A Figura 2(a) apresenta uma visão conceitual de um multiprocessador de memória compartilhada, em que os múltiplos *cores* têm acesso a toda a memória, a qual é compartilhada por todos os núcleos. A Figura 2(b) exemplifica um multicomputador com troca de mensagens, em que conjuntos compostos por *cores* e memória são interconectados por uma rede em *chip*. As memórias são acessíveis somente pelos *cores* pertencentes ao seu conjunto e os *cores* se comunicam entre si via troca de mensagens através da rede que interconecta todos os conjuntos. Já a Figura 2(c) evidencia um sistema distribuído de grande escala, em que computadores são conectados através de uma *wide-area network* (WAN) com o intuito de formar um sistema distribuído.

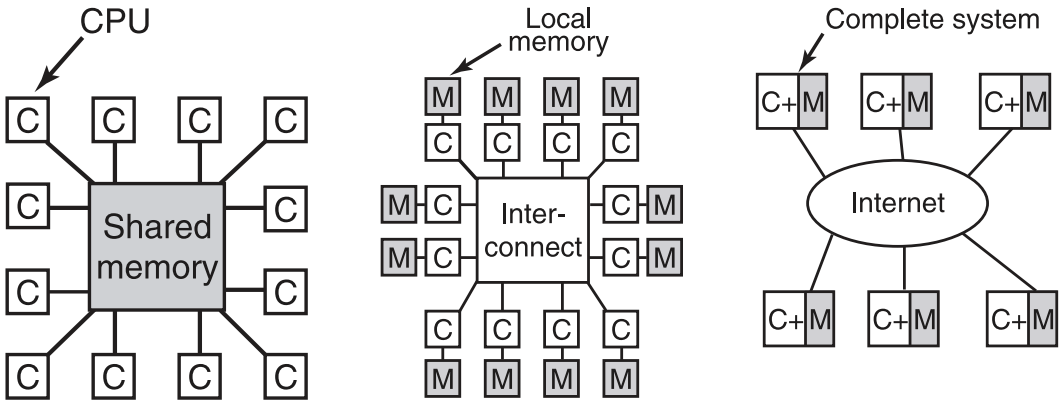
Neste contexto, a classe de processadores *lightweight manycores* destacam-se por atrelar alto poder de processamento com eficiência energética (FRANCESQUINI et al.,

Figura 2 – Exemplos de arquiteturas MIMD.

(a) multiprocessador de memória compartilhada.

(b) multicompudator com troca de mensagens.

(c) sistema distribuído de grande escala.



Fonte: Adaptado de Tanenbaum e Bos (2014)

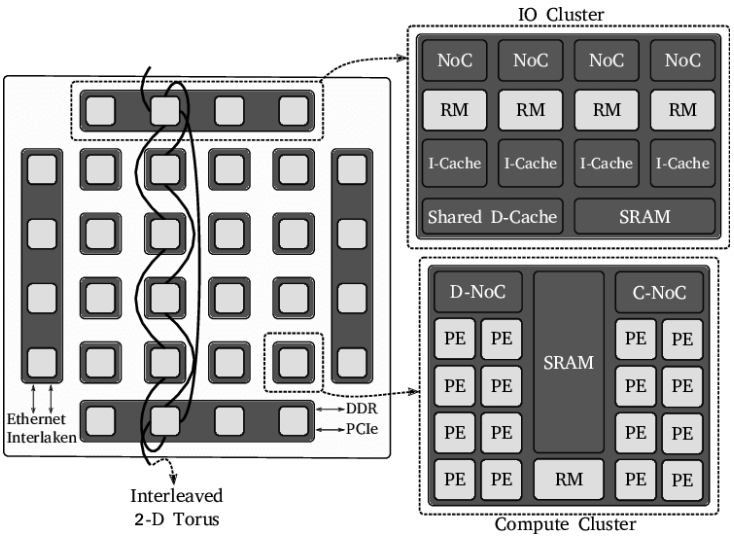
2015). Os *lightweight manycores* são classificados como *Multiprocessor System-on-Chip* (MPSoC) e suas arquiteturas apresentam as seguintes características:

(i) Integrar de centenas à milhares de núcleos de processamento operando a baixas frequências em um único *chip*;
(ii) Processar cargas de trabalho MIMD;
(iii) Organizar os núcleos em conjuntos, denominados *clusters*, para compartilhamento de recursos locais;
(iv) Utilizar *Networks-on-Chip* (NoCs) para transferência de dados entre núcleos ou *clusters*;
(v) Possuir sistemas memória distribuída restritivos, compostos por pequenas memórias locais; e
(vi) Apresentar componentes heterogêneos (*Compute Clusters* e *I/O Clusters*).

Alguns exemplos comerciais bem sucedidos de *lightweight manycores* são o Kalray MPPA-256 (DINECHIN et al., 2013), PULP (ROSSI et al., 2017) e Sunway SW26010 (FU et al., 2016). Especificamente, nós utilizamos o processador Kalray MPPA-256 para o desenvolvimento deste trabalho. A Figura 3 apresenta uma visão geral do processador Kalray MPPA-256 e suas peculiaridades, tais como:

(i) Integrar 288 núcleos de baixa frequência em um único *chip*;
(ii) Possuir núcleos organizados em 20 *clusters*;
(iii) Dispor de 2 NoCs para transferência de dados entre *clusters*, uma para controle e outra para dados;
(iv) Possuir um sistema de memória distribuída composto por pequenas memórias locais, e.g., *Static Random Access Memory* (SRAM) de 2 MB;

Figura 3 – Visão arquitetural do processador Kalray MPPA-256.



Fonte: Penna et al. (2019)

(v) Não dispor de coerência de *cache* em *hardware*;

(vi) Apresentar heterogeneidade, como *clusters* destinados à computação (*Compute Clusters*) e *clusters* destinados à comunicação com periféricos (*I/O Clusters*).

2.2 NANVIX OS

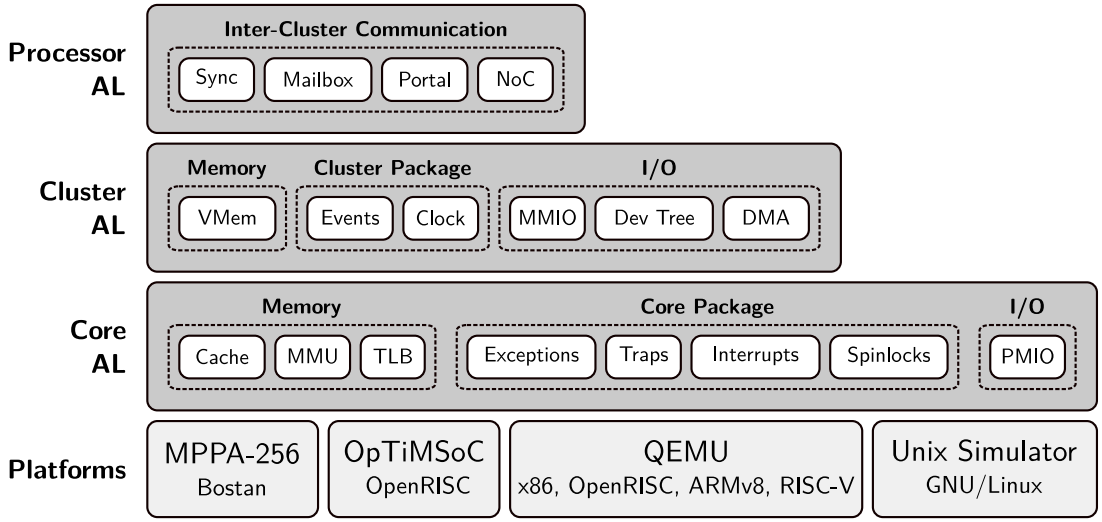
O Nanvix[1] é um SO distribuído e de propósito geral que busca equilibrar desempenho, portabilidade e programabilidade para *lightweight manycores* (PENNA et al., 2019). O *kernel* do Nanvix é estruturado em três camadas de abstração. São elas:

Nanvix *Hardware Abstraction Layer* (HAL) é a camada mais baixa que abstrai e provê o gerenciamento dos recursos de *hardware* sobre uma visão comum (PENNA; FRANCIS; SOUTO, 2019). Entre esses recursos estão: *cores*, *Translation Lookaside Buffer*s (TLBs), *cache*, *Memory Management Unit* (MMU), NoC, interrupções, memória virtual e recursos de *I/O*. De maneira geral, esta camada provê abstrações ao nivel do *core*, *cluster* e comunicação/sincronização entre *clusters* (PENNA, 2021). A Figura 4 ilustra a estrutura interna da HAL do Nanvix.

Nanvix *Microkernel* é a camada intermediária que provê gerenciamento de recursos e os serviços mínimos de um SO em um *cluster*. Entre esses serviços se encontram o gerenciamento de *threads* e memória, controle de acesso à memória, interface para chamadas de sistema e a comunição entre processos. As chamadas de sistema podem ser executadas localmente, caso acessem dados *read-only* ou alterem estruturas internas do *core*, ou remotamente pelo *master core*, que atende à requisição e libera

[1] Disponível em https://github.com/nanvix

Figura 4 – Estrutura interna da HAL do Nanvix.



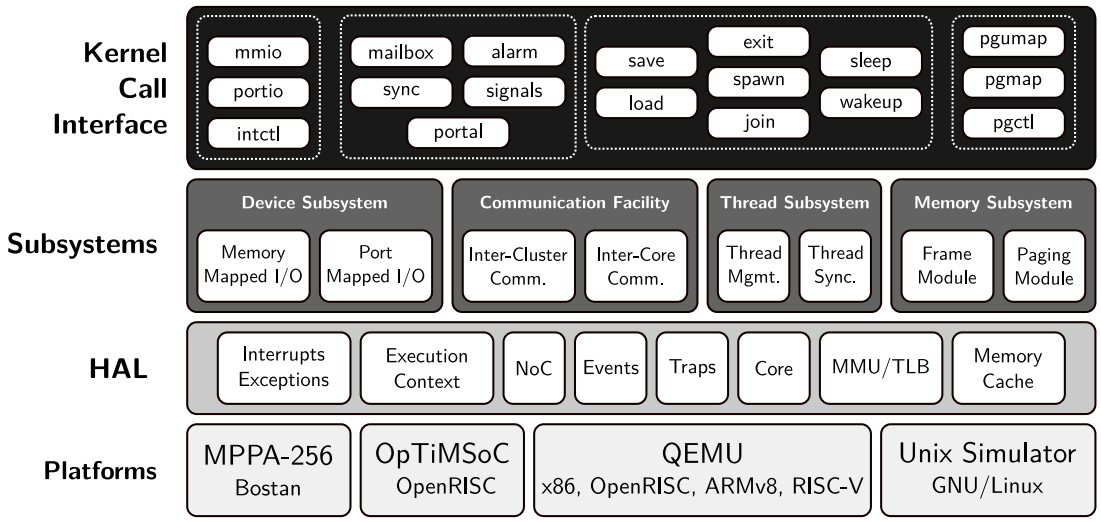
Fonte: Penna (2021)

o *slave core* requisitante ao término da chamada (PENNA, 2021). Essa característica adjetiva o *microkernel* como assimétrico. A Figura 5 ilustra a estrutura interna do *microkernel* do Nanvix.

Nanvix *Multikernel* é a camada superior que provê os serviços mais complexos de um SO e dispõe uma visão a nível do processador em si. Os serviços são hospedados em *I/O Clusters*, i.e., isolados das aplicações de usuário. Os serviços atendem as requisições vindas dos processos de usuário através de um modelo cliente-servidor. As requisições e respostas são enviadas/recebidas através de passagem de mensagem via NoC. Os serviços dessa camada podem ser entendidos como fontes de informação que mantém a execução dos processos consistentes no processador, tendo em vista a natureza distribuída da memória nessas arquiteturas. Alguns serviços incluídos no Nanvix são mecanismos de *spawn* de processos e gerenciamento de nomes lógicos dos processos à fim de abstrair a localização dos processos no processador. A Figura 6 ilustra o *multikernel* do Nanvix.

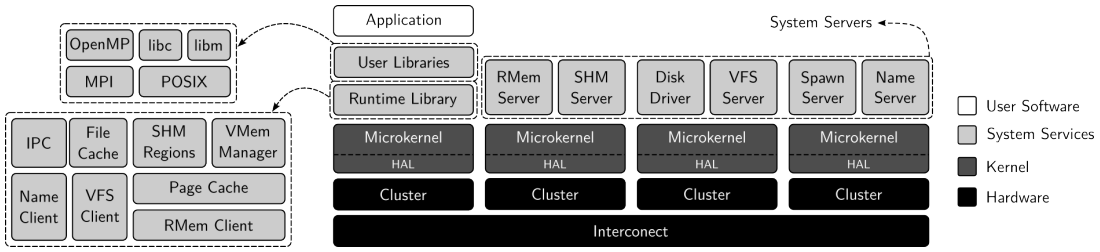
Em sua abordagem original, os processos no Nanvix são estáticos, i.e., cada *cluster* possui apenas um processo. Desse modo, uma vez que o processo inicia sua execução em um *cluster*, este finalizará a execução no mesmo *cluster*. Isso torna o processo dependente do *cluster* que o executa, fazendo com que a comunicação entre processos esteja atrelada aos *clusters* nos quais os processos são executados (e não aos processos em si). A falta de mobilidade dos processos nesse modelo pode trazer sobrecargas ao processador, afetando diretamente o desempenho do sistema quando múltiplas aplicações estão em execução simultânea no processador. No caso de aplicações paralelas, compostas por múltiplos processos (ou *threads*) que se comunicam, a disposição dos processos (ou *threads*) nos *clusters* se torna importante, pois a comunicação entre *clusters* próximos é mais rápida e

Figura 5 – Estrutura interna do *microkernel* do Nanvix.



Fonte: Penna (2021)

Figura 6 – Estrutura interna do *multikernel* do Nanvix.



Fonte: Penna (2021)

resulta em menor consumo energético do processador. Sendo assim, melhorar a mobilidade e a disposição dos processos no processador possibilitaria melhorar o gerenciamento dos recursos do mesmo.

Um exemplo de mobilidade é viabilizar a migração de processos entre *clusters*. Neste contexto, este trabalho explora essa desassociação entre o processo e o *cluster* que o executa. Deste modo, nós aumentamos a mobilidade dos processos, permitindo a migração de processos entre os *clusters* do processador.

2.2.1 Abstrações de Comunicação do Nanvix

O Nanvix dispõe de três abstrações de comunicações para transferência de dados e sincronização entre *clusters* (PENNA, 2021). Nas próximas seções serão detalhadas as três abstrações principais do Nanvix.

Sync. A abstração *Sync* suporta a sincronização entre *kernels*. Através dela um processo pode esperar um sinal, que pode ser disparado por outro processo remotamente através das interfaces NoC. Essa abstração é muito utilizada na inicialização do

sistema para garantir um estado inicial consistente dos subsistemas do SO (PENNA, 2021).

Mailbox. A abstração *Mailbox* é responsável pelo suporte ao envio de mensagens de controle através da troca de pequenas mensagens de tamanho fixo. A abstração segue a semântica $N : 1$ e funciona da seguinte forma: um nó (destinatário da mensagem) possuí uma *Mailbox*, da qual lê mensagens, e múltiplos nós (remetentes da mensagem) podem escrever nessa *Mailbox* (PENNA, 2021).

Portal. A abstração portal suporta a troca de mensagens grandes e segue a semântica 1 : 1. A abstração pode ter uso em diversos cenários que exigem grandes transferências de dados entre *clusters* (PENNA, 2021).

2.3 VIRTUALIZAÇÃO

A virtualização pode ser entendida como uma técnica de abstração de *hardware* que permite a criação de uma versão virtual de um ambiente, como computadores, SOs, sistemas de armazenamento, redes, aplicações, etc. Nesse cenário, muitas vezes é possível a criação de múltiplas instâncias dessa versão virtual, as quais competem pelos recursos físicos/reais. A virtual ização pode ser classificada em três grupos principais: a virtualização total; a para-virtualização e a virtualização a nível de processo. Além disso, o isolamento e independência das instâncias virtuais garantem à virtualização algumas vantagens muito exploradas atualmente, especialmente em ambientes *cloud* (MANOHAR, 2013). Dentre elas: flexibilidade, portabilidade, escalabilidade e segurança.

O conceito de virtualização pode ser traçado desde a década de 50, durante a época dos *mainframes* e do emergente conceito de memória virtual (CAMPBELL; JERONIMO, 2006). Nesse período, a preocupação era tornar um recurso físico acessível a múltiplos usuários simultaneamente. Essa motivação sustentou a evolução da virtualização, levando ao surgimento das Máquinas Virtuais (VMs) e dos *hypervisors*. Com o tempo, viu-se o surgimento de novos projetos, como o M44/44x da *International Business Machines Corporation* (IBM), responsável pelo nascimento de um novo *design* para os sistemas de tempo compartilhado. Nessa nova estrutura, a máquina central repartia seus recursos em diversas instâncias de VMs, que eram utilizadas por múltiplos usuários simultaneamente.

O retorno das pesquisas sobre virtualização ocorreu mais recentemente, na década de 90 (CAMPBELL; JERONIMO, 2006). Essa foi a época em que o número de serviços e servidores cresceu bruscamente. Naturalmente, com o aumento do número de servidores e aplicações hospedadas nesses servidores, a necessidade de gerenciamento desses recursos também aumentou. Nesse cenário, a virtualização mostrou-se uma solução viável por permitir que diversas VMs compartilhassem um único servidor mantendo, ainda assim, a independência dos serviços providos por essas VMs encapsulados. Isso significa que a interrupção ou quebra de um serviço não afeta os demais graças à virtualização,

que garante uma maior flexibilidade e escalabilidade. Por consequência, a virtualização reduziu os custos de manutenção e operação, já que os recursos de *hardware* foram utilizados de maneira mais eficiente e houve a redução na quantidade de máquinas físicas para gerenciar.

Atualmente, a tendência de uso de VMs em servidores continua crescente. Hoje, a virtualização de servidor é uma das formas mais comuns de virtualização, sendo utilizada em ambientes *cloud* para garantir o suporte à execução de múltiplas aplicações, possivelmente em SOs distintos sobre o mesmo *hardware* (MANOHAR, 2013).

2.3.1 Virtualização total

A virtualização total tem como objetivo abstrair o *hardware* de um computador como um todo. Cada instância executa isoladamente e independentemente uma das outras. Neste tipo de virtualização, é utilizado um *Virtual Machine Monitor* (VMM), também conhecido como *hypervisor*, o qual, na virtualização total, é classificado como tipo 1 (CAMPBELL; JERONIMO, 2006). O *hypervisor* tipo 1 é um *software* que roda no nível mais privilegiado e atua como um intermediário entre o *hardware* e os múltiplos SOs. O *hypervisor* tipo 1 é o único programa do sistema que possui o acesso ao *hardware* físico, e.g., *Central Processing Unit* (CPU), memória e armazenamento, sendo responsável por gerenciar esses recursos de *hardware* para cada instância virtual da máquina virtualizada (SWEENEY, 2016).

2.3.2 Para-virtualização

De forma similar à virtualização total, o objetivo da para-virtualização também é a abstração da máquina em sua totalidade. Contudo, em contraste com a virtualização total, na para-virtualização uma única instância da máquina executa um SO, chamado de SO hospedeiro, que detém o acesso ao *hardware*. Enquanto as demais instâncias executam seus respectivos SOs, chamados de SOs convidados, sob o intermédio de um *hypervisor* (VMM) tipo 2, que pode ser entendido como um processo regular do SO hospedeiro (CAMPBELL; JERONIMO, 2006). Sendo assim, o *hypervisor* tipo 2 atua como um intermediário entre o SO convidado e o SO hospedeiro. O SO hospedeiro reconhece as requisições do SO convidado e gerencia os recursos de *hardware* deste (SWEENEY, 2016).

2.3.3 Virtualização a Nível de Processo e Conteinerização

Virtualizar um processo ou aplicação é o processo de desacoplar a execução de um processo do sistema que o executa. Nesse contexto, o processo tem uma visão virtual

única do sistema, de modo que a execução de cada aplicação ocorre independentemente uma da outra.

A Figura 7(c) ilustra o modelo de execução de uma aplicação conteinerizada. Diferentemente dos modelos que utilizam *hypervisors* tipo 1 ou 2, respectivamente ilustrados nas Figuras 7(a) e 7(b), neste tipo de virtualização, cada aplicação é isolada em um ambiente virtual, também conhecido como contêiner. No contêiner estão contidas todas as bibliotecas, arquivos e configurações necessárias para a execução da aplicação. Como não é necessária a criação de um SO para cada aplicação, a virtualização se torna muito mais leve, i.e., tem um impacto menor na memória.

O fato do modelo de conteinerização exigir muito menos espaço e complexidade arquitetônica para existir torna este modelo muito atrativo para sistemas com restrições de memória, tal qual os *lightweight manycores*.

Alguns conceitos muito presentes na conteinerização são:

namespace. *Namespaces* foram introduzidos no *kernel* do Linux com o objetivo de isolar recursos virtuais de um grupo de processos (GAO et al., 2017). Os processos podem ser associados a vários *namespaces* e, por isso, podem ter visões diferentes e customizadas dos recursos do sistema (WATADA et al., 2019). Atualmente, existem seis tipos diferentes de *namespaces* (WATADA et al., 2019):

(i) *mount*: O *mount namespace* isola o sistema de arquivos e permite que um processo veja um sistema de arquivos diferente do sistema de arquivos real do hospedeiro (WATADA et al., 2019). Através dele é possível que cada contêiner tenha seu diretório raiz específico (DUA; RAJA; KAKADIA, 2014);

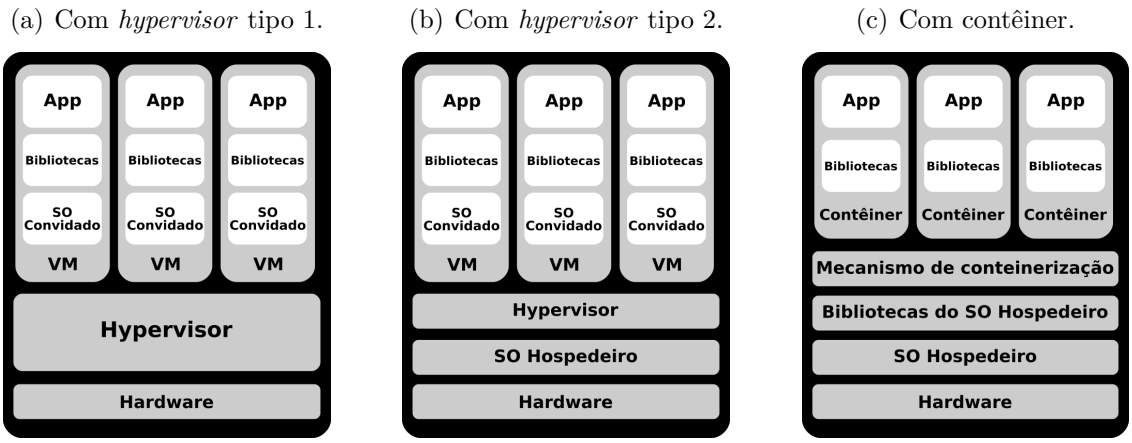
(ii) *Unix Timesharing System* (UTS): garante que cada contêiner possa ver e alterar seu próprio *hostname*, tornando-o um nó independentemente nomeado na rede (WATADA et al., 2019; DUA; RAJA; KAKADIA, 2014);

(iii) *Process ID* (PID): virtualiza os identificadores dos processos de forma que cada contêiner tenha sua própria árvore de processos i.e., os processos pertencentes a um *namespace* são visíveis àqueles pertencentes ao mesmo *namespace*. Dessa forma, cada processo em um contêiner é associado a dois identificadores: um global e único associado ao hospedeiro; e um local associado ao contêiner (WATADA et al., 2019);

(iv) NET: virtualiza os artefatos de rede, como dispositivos de rede, endereços IPs, portas e tabelas de roteamento, permitindo que cada contêiner tenha seus próprios componentes de rede virtuais (WATADA et al., 2019; DUA; RAJA; KAKADIA, 2014);

(v) *Inter-Process Communication* (IPC): provê isolamento de mecanismos de comunicação entre processos, tais como: filas de mensagens, semáforos e segmentos de memória compartilhados (WATADA et al., 2019; DUA; RAJA; KAKADIA, 2014);

Figura 7 – Comparação de modelos de ambientes de execução de aplicação.



Fonte: Adaptado de Combe, Martin e Pietro (2016)

(vi) *user*: isola o usuário, permitindo que um usuário comum (sem privilégios) no hospedeiro seja considerado um superusuário por um processo dentro do *namespace* (WATADA et al., 2019).

***control groups* (cgroups).** É um mecanismo que permite a limitação da quantidade de recursos que um contêiner pode utilizar. Os recursos que podem ser limitados incluem: memória, CPU, *I/O* e rede (WATADA et al., 2019). Esta é uma ferramenta utilizada por muitas plataformas de conteinerização, como o *Docker* e *Linux Containers* (LXC), para garantir que um contêiner não utilize mais recursos do que o necessário.

Várias ferramentas de conteinerização utilizam *namespaces* e cgroups, ou conceitos similares, para a criação de um ambiente virtual para execução de um contêiner de aplicação ou contêiner de máquina.

LXC é uma dessas ferramentas de conteinerização. Ele é uma implementação que usa as funcionalidades de *namespaces* e cgroups do Linux para isolar os recursos do sistema hospedeiro (WATADA et al., 2019). O *Linux Containers Daemon* (LXD) é a interface com que o usuário cria e interage com os contêineres. A principal vantagem do LXC surge por ser baseado em mecanismos do próprio *kernel*. Essa característica o torna mais eficiente e provê um melhor isolamento de recursos, em especial de rede e sistema de arquivos (DUA; RAJA; KAKADIA, 2014). Em contrapartida, dentre algumas limitações do LXC estão: por ser uma implementação baseada no *kernel* do Linux, o LXC não é portável para outros sistemas operacionais; há ainda problemas de segurança em aberto; e os contêineres compartilham o mesmo *kernel* (DUA; RAJA; KAKADIA, 2014).

Outra ferramenta muito conhecida é o *Docker*. O *Docker* traz uma abordagem um pouco diferente do LXC. Enquanto no LXC é posssível criar VMs, com diversos processos executando isoladamente do hospedeiro em um mesmo contêiner, a proposta do

Docker é a execução de processos únicos i.e., o foco é a portabilidade e ser um suporte a execução de microsserviços (KAHUHA, 2023). Em sua abordagem inicial, o *Docker* utilizava o LXC internamente (DUA; RAJA; KAKADIA, 2014), porém nas versões mais modernas o LXC foi substituído pelo *Conteinerd*, que é outra biblioteca de criação e gerenciamento de contêineres (CROSBY, 2017). Mesmo não sendo mais uma extensão do LXC, o *Docker* ainda utiliza funcionalidades do *kernel* do Linux e.g., *namespaces* e cgroups, em seu funcionamento interno. Isso faz com que sua execução em outros SOs e.g., *macOS* e *Windows*, exija a utilização de uma VM (mais leve que as convencionais) para sua execução.

Essas e outras ferramantas de conteinerização trouxeram novas possibilidades ao mundo da computação. A surgimento dessas tecnologias aumentaram a flexibilidade dos servidores, já que agora é possível criar ambientes isolados para as aplicações. Além disso, esse isolamento de execução permite a utilização de algumas técnicas, como a migração de contêineres, a interrupção de uma aplicação isoladamente, o *checkpointing*, etc.

2.3.4 Outros Tipos de Virtualização

Virtualização de *desktop*: usuários acessam o ambiente computacional, ou *desktop*, remotamente. O poder e recursos computacionais estão centralizados, mas os pontos de acesso podem ser diversos. Também é conhecido como *Virtual Desktop Infrastructure* (VDI).

Virtualização de armazenamento: múltiplos dispositivos de armazenamento são unificados sob uma mesma visão de dados i.e., os dados estão dispersos, mas aparecem como um único conjunto de dados compartilhados.

Virtualização de rede: múltiplas redes virtuais, cada uma com seu conjunto de recursos (como roteadores, *switches* e *firewalls*) operam sobre uma mesma rede física. Isso permite melhor gerenciamento de tráfego e otimização.

2.4 MIGRAÇÃO

A migração é o processo de transferência de uma aplicação ou VM de um ambiente a outro. A migração é muito usada atualmente principalmente em ambientes *cloud* (IMRAN et al., 2022). A migração traz diversos benefícios e vantagens aos serviços em um ambiente computacional, tais como:

(i) Balanceamento de carga: é uma técnica que permite que a carga de trabalho de servidores sobrecarregasdos seja distribuída entre outras máquinas a fim de evitar falhas de sistema ou aumento de latência na resposta dos serviços. Através dessa técnica, VMs alocadas em servidores sobrecarregados são migrados para servidores

menos utilizados, melhorando a utilização dos recursos computacionais (WOOD et al., 2007).

(ii) Tolerância a falhas: é uma técnica que permite a migração de uma VM para um servidor em melhor condição de funcionamento quando o servidor em que está alocada apresenta algum tipo de falha (NAGARAJAN et al., 2007).

(iii) Manutenção de sistema: os servidores requerem que periodicamente sejam feitas revisões/manutenções em seus sistemas. Durante esses períodos, as aplicações alocadas nestas máquinas não conseguiriam executar. Graças à migração, estas aplicações/VMs são tranferidas a outro servidor durante esses intervalos de manutenção sem que os serviços sejam afetados (DEVI et al., 2011).

(iv) Gerenciamento de energia: através da migração, é possível gerenciar a alocação de VMs nos servidores de modo que alguns não tenham carga de trabalho a executar e possam ser desligados, economizando energia. Isso é feito, claro, de uma maneira que não sobrecarregue os servidores que estão em funcionamento (HU et al., 2008).

2.4.1 Tipos de Migração

A migração pode ser classificada em dois tipos distintos: *cold migration* (*non-live migration*) e *hot migration* (*live migration*) (IMRAN et al., 2022).

2.4.1.1 Cold migration

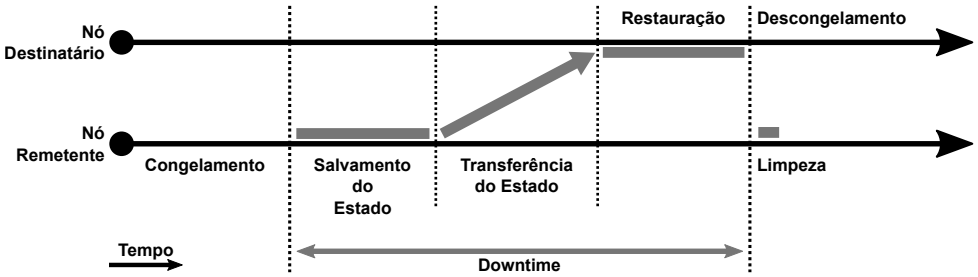
A Figura 8 ilustra o fluxo de execução da *cold migration*,também é conhecida como *non-live migration*. Neste tipo de migração, o sistema a ser migrado (contêiner ou VM) precisa ser desligado antes do processo de migração começar.Detalhadamente, o sistema é desligado, o estado (*checkpoint*) do sistema é salvo e enviado ao SO ou VM destinatário. O sistema é restaurado no destino e o estado do sistema apagado no remetente.

Este geralmente não é considerado um método eficiente e não é muito utilizado na indústria e mercado. Isso porque o período de tempo em que a aplicação fica inativa, *aka down time*, é alto. O *down time* é elevado devido à alta quantidade de dados a serem transferidos e pelo tempo extra para desligar e ligar o sistema novamente. Isso não é considerado aveitável atualmente, haja vista a grande quantidade de serviços que não podem parar sua execução (SINGH et al., 2022; IMRAN et al., 2022).

2.4.1.2 Hot Migration

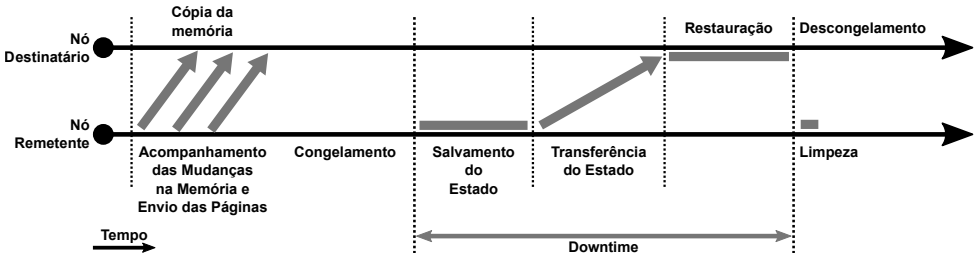
Em contraste com a *cold migration*, na *hot migration*, também conhecida como *live migration*, o sistema a ser migrado não precisa ser desligado antes do procedimento começar. O principal objetivo desse processo é maximizar a performance do sistema

Figura 8 – Fluxo de execução da *cold migration*.



Fonte: Adaptado de Synytsky (2016)

Figura 9 – Fluxo de execução da *pre-copy migration*.



Fonte: Adaptado de Synytsky (2016)

durante a migração, melhorar o uso da rede e reduzir o *down time* do sistema (IMRAN et al., 2022).

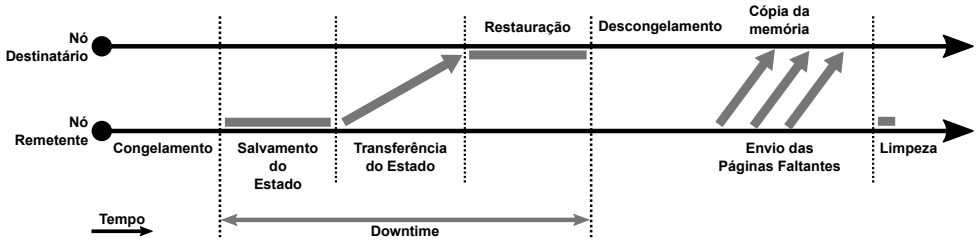
Neste modelo, a migração poder ser classificada de acordo com a técnica utilizada para a transferência dos dados i.e., *pre-copy migration*, *post-copy migration* e migração híbrida.

2.4.1.2.1 *Pre-Copy migration*

A Figura 9 ilustra o fluxo de execução da *pre-copy migration*. Neste modelo, ao receber a requisição de migração, o sistema cria uma pré-imagem do seu estado atual e a envia ao destinatário enquanto continua executando normalmente. Nesta estrutura está contida o esquema de paginação atual do sistema e, por vezes, alguns dados de execução. Conforme o esquema de paginação é modificado no remetente, essa estrutura é atualizada e enviada ao destinatário. Depois disso, é salvo o estado atual completo do sistema, que, então, é enviado ao destino. O sistema é restaurado no destino e o estado do sistema apagado no remetente.

Esse fluxo de migração faz com que o tempo total de migração seja maior que o do *cold migration*, já que há a retransmissão de dados, em especial páginas de memória. Em contrapartida, o *down time* é reduzido, pois o sistema executa normalmente na parte inicial da migração, quando é feita e enviada a pré-imagem ao destinatário (SINGH et al., 2022; IMRAN et al., 2022).

Figura 10 – Fluxo de execução da *post-copy migration*.



Fonte: Adaptado de Synytsky (2016)

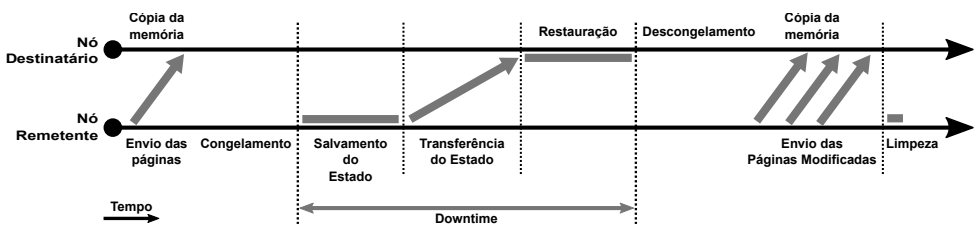
2.4.1.2.2 *Post-Copy migration*

Tal como ilustrado pela Figura 10, no modelo *post-copy migration* o remetente cria um estado parcial do sistema em execução e o envia ao destinatário. Nesse estado, está incluso apenas o essencial para o início do procedimento de descongelamento do sistema, não abrangendo o esquema de paginação. Quando o sistema é restaurado no destinatário, ocorrerão várias faltas de páginas, as quais resultam em requisições de páginas ao remetente, o qual as envia ao destinatário. Quando todas as páginas são transferidas ao destinatário, o sistema é apagado do remetente (SINGH et al., 2022; IMRAN et al., 2022).

2.4.1.2.3 Migração Híbrida

Como podemos observar na Figura 11, no modelo de migração híbrida são utilizadas as técnicas de *pre-copy* e *post-copy* em conjunto. Inicialmente, uma pré-imagem do sistema é feita de forma similar ao modelo *pre-copy migration*. Esta pré-imagem é enviada ao destinatário, enquanto o sistema executa normalmente. Em contraste com o método *pre-copy* nesta etapa não são reenviadas as páginas modificadas/atualizadas. Depois, o estado completo do sistema é enviado e as páginas modificadas/atualizadas são requisitadas ao remetente tal como na *post-copy migration*. Quando todas as páginas são transferidas ao destinatário, o sistema é apagado do remetente (SINGH et al., 2022; IMRAN et al., 2022).

Figura 11 – Fluxo de execução da migração híbrida.



Fonte: Adaptado de Synytsky (2016)

3 TRABALHOS RELACIONADOS

Neste capítulo, serão mostradas técnicas e pesquisas que estão sendo desenvolvidas no que diz respeito à virtualização e migração. Serão apresentados trabalhos relacionados, bem como serão evidenciadas as semelhanças e diferenças com o presente trabalho.

Grande parte das pesquisas relacionadas à migração estão inseridas em ambientes *cloud*. Nesses casos, os esforços estão voltados para redução do tempo total de migração, diminuição do *down time* (STOYANOV; KOLLINGBAUM, 2018; CLARK et al., 2005) e exploração/otimização das vantagens que a migração de processos oferece nesses ambientes computacionais. Entre essas vantagens podem-se citar:

(i) Balanceamento de carga (CHOUDHARY et al., 2017; WANG et al., 2019);
(ii) Tolerância a falhas (FERNANDO et al., 2019);
(iii) Gerenciamento do consumo de energia (ALDOSSARY; DJEMAME, 2018);
(iv) Compartilhamento de recursos; e
(v) Manutenção de sistemas sem interrupções (CHOUDHARY et al., 2017; WANG et al., 2019).

Apesar da maioria das pesquisas estarem voltadas à exploração desses benefícios e diminuição do tempo de migração e *down time* em ambientes *cloud*, há alguns autores preocupados com o desenvolvimento de soluções envolvendo virtualização e migração em ambientes de recursos restritos. Dessa forma, como a temática de limitação de recursos, especialmente de memória, é muito presente neste trabalho, serão abordados nas próximas seções algumas pesquisas desses autores i.e., pesquisas voltadas à busca pelo uso da virtualização/migração de forma mais leve e cujo impacto no *hardware* seja reduzido, adaptando-se a esses sistemas de recursos limitados.

3.1 *"VIRTUALIZATION ON TRUSTZONE-ENABLED MICROCONTROLLERS? VOILÀ!"*

O artigo *"Virtualization on TrustZone-enabled Microcontrollers? Voilà!"* (PINTO et al., 2019) aborda a possiblidade de implementação da virtualização em microcontroladores que utilizam *TrustZone*. *TrustZone* é uma tecnologia de *hardware* voltada à segurança, em que a execução de um sistema pode ser dividida entre normal e segura. Os autores afirmam que essa tecnologia pode ser explorada além das suas propriedades de segurança. Isso porque o *TrustZone* também provê certo nível de isolamento dos recursos, o que o torna viável de ser usado para virtualização, afinal o isolamento cria um ambiente seguro e propício para a execução simultânea e isolada de múltiplas Máquinas Virtuais (VMs).

Pinto et al. (2019) expõe a dificuldade de se implementar a virtualização em *Microcontroller Units* (MCUs) devido aos seus recursos limitados. Nesses ambientes, não

é possível a utilização de *hypervisors* tradicionais, haja vista a baixa complexidade de *hardware* das MCUs. Sendo assim, para atender a necessidade de baixo impacto nos recursos dos MCUs, os autores propõem uma solução que usa um *hypervisor* mais leve para gerenciar as VMs nesses ambientes utilizando a tecnologia *TrustZone* para garantir o isolamento das VMs.

Os testes foram feitos num microcontrolador *Cortex-M4* e a solução proposta garante o suporte à execução múltipla de VMs em microcontroladores.

3.2 *"CHECKPOINTING AND MIGRATION OF IOT EDGE FUNCTIONS"*

O artigo *"Checkpointing and migration of IoT edge functions"* (KARHULA; JANAK; SCHULZRINNE, 2019) propõe um artifício envolvendo migração de contêiners entre dispositivos *Internet of Things* (IoT) de borda como solução para a diminuição do uso de recursos em dispositivos IoT.

Os autores evidenciam que os aparelhos IoT são usados na computação de borda para provomover o que chamamos de *Functions as a Service* (FaaS), que é um tipo de serviço oferecido por diversas plataformas, como a *Amazon AWS Lambda* e *Google Cloud Functions*. Contudo, dispositivos IoT possuem recursos limitados, restringindo-se à execução de poucos contêineres simultaneamente. Além disso, as abordagens tradicionais de FaaS sugerem a execução ininterrupta dos contêineres que são iniciados. Isso torna a computação de borda ineficiente, pois esse esquema pode sobrecarregar rapidamente os dispositivos IoT, haja vista a memória limitada desses. A situação se agrava ainda mais quando consideramos funções de longa duração bloqueantes (muito comuns em sistemas de autenticação) e.g., funções que esperam alguma requisição, resposta ou qualquer tipo de sinal de outro sistema, seja ele um outro dispositivo IoT ou uma ação humana.

Dessa forma, Karhula, Janak e Schulzrinne (2019) propõem um esquema de *checkpointing* utilizando *Docker* e *Checkpoint/Restore In Userspace* (CRIU). Através dessas tecnologias, os contêineres que não estão executando computação útil são interrompidos e salvos em disco, liberando espaço da memória para a execução de outro contêiner. Isso se torna extremamente útil quando consideramos funções de longa duração bloqueantes, já que durante o tempo de espera pelo sinal, a aplicação pode ser interrompida. Além disso, com o estado salvo em disco, a migração de contêineres entre dispositivos IoT de borda se torna possível. Dessa forma, além de reduzir o uso de recursos nos dispositivos de computação em borda, através da migração dos contêineres, outros benefícios surgem, como o balanceamento de carga e tolerância a falhas entre aparelhos IoT de borda.

Os testes foram feitos em uma *Raspberry Pi 2 Model B*, a qual rodava diversos contêineres com aplicações em *Node JS* de longa duração e que simulavam o comportamento bloqueante. Os resultados apontam que houve economia no uso de recursos, em especial da memória, e que a migração de contêineres entre dispositivos IoT de borda é possível.

3.3 *"LIGHTWEIGHT VIRTUALIZATION AS ENABLING TECHNOLOGY FOR FUTURE SMART CARS"*

O artigo *"Lightweight virtualization as enabling technology for future smart cars"* (MORABITO et al., 2017) discorre sobre a possibilidade de usar a virtualização no desenvolvimento de aplicações para carros inteligentes. Os sistemas presentes nos carros inteligentes também tem certa limitação de recursos que dificultam a aplicação direta de *hypervisors* tradicionais, muito comuns em ambientes *cloud*.

Sendo assim, os autores propõem um sistema que utiliza contêineres *Docker* para criar uma camada de abstração a nível de processo. Dessa forma, cada aplicação é executada em um contêiner distinto. Esse sistema tem impacto menor nos recursos de *hardware* e é suficiente para garantir a execução isolada das aplicações virtuais (contêineres).

Além disso, o sistema engloba um escalonador de contêineres, que é responsável por gerenciar os contêineres e o *hardware* alocado para cada um. Ademais, tem finalidade de sinalizar a instanciação e destuição dos contêineres conforme a necessidade. Esse escalonador é capaz de gerenciar os recursos de *hardware* de forma a garantir que os contêineres sejam executados de maneira eficiente, sem que haja desperdício de recursos. No modelo proposto pelos autores, há 4 tipos de tarefas: *critical*, *high*, *moderate* e *low*. Cada um desses tipos possui um nível de prioridade, sendo que o *critical* é o mais prioritário e o *low* é o menos prioritário. O escalonador é responsável por garantir que as tarefas de maior prioridade sejam executadas primeiro. Tarefas relacionadas à segurança dos passageiros e.g., sistemas de alerta ou câmera são consideradas mais prioritárias que tarefas relacionadas à sistemas de entretenimento e.g., sistemas de áudio ou vídeo.

A proposta foi testada em uma *Raspberry Pi 3* e os resultados foram considerados positivos. Os contêineres garantiram a execução do sistema de maneira a considerar a limitação de *hardware* e suportaram a execução paralela de múltiplas aplicações. O escalonador de contêineres foi capaz de gerenciar os recursos de maneira eficiente, priorizando as tarefas de maior prioridade.

3.4 COMPARAÇÃO DO PRESENTE TRABALHO COM OS TRABALHOS RELACIONADOS

O presente trabalho se asemelha com os trabalhos relacionados apresentados no sentido de aplicar a virtualização e migração em um sistema com recursos restritos. Contudo, em contraste com os trabalhos apresentados, a principal vantagem explorada com a virtualização é o aumento da mobilidade dos processos, possibilitando a migração de processos. A utilização eficiente dos recursos promovida pela virtualização, mesmo que necessária nos *lightweight manycores* pela limitação de recursos computacionais (em especial a memória) se torna uma vantagem indireta da virtualização. Isso porque o uso eficiente de *hardware* é provido mais pela migração (através da melhor disposição dos

processos entre os *clusters*) do que pela virtualização em si.

Além disso, o presente trabalho explora a virtualização e migração usando contêineres, tal qual o segundo e terceiro trabalho, porém em um ambiente diferente e com outra abordagem. Neste trabalho, o foco é o desenvolvimento de um sistema de virtualização baseado em contêineres adaptado ao Sistema Operacional (SO) e sem o uso de ferramentas externas, como o *Docker*. Além disso, a migração das aplicações são entre *clusters* de um mesmo processador, e não entre nós de computação de borda ou servidores *cloud*.

4 PROPOSTA DE VIRTUALIZAÇÃO E MIGRAÇÃO DE PROCESSOS PARA *LIGHTWEIGHT MANYCORES*

O Nanvix surgiu com a proposta de resolver os problemas de programabilidade e portabilidade em *lightweight manycores*. Apesar de ser uma abordagem promissora, o Sistema Operacional (SO) ainda possui características para se trabalhar. Em especial, destaca-se a falta de mobilidade dos processos no processador. Essa questão afeta o desempenho do sistema porque impede a redistribuição dos processos para organizações mais eficientes e.g., remanejando processos com comunicação intensa entre si para *clusters* fisicamente mais próximos. Neste contexto, este trabalho de conclusão ataca essa problemática. Ou seja, este trabalho propõe-se a aumentar a independência dos processos no processador através do projeto e desenvolvimento do suporte à virtualização e migração de processos em *lightweight manycores*.

Ambientes *cloud*, nos quais o sistema de memória é de alta capacidade, usufruem da utilização de Máquinas Virtuais (VMs) para isolar duplicatas inteiras de SOs com o auxílio da virtualização a nível de instrução (SHARMA et al., 2016). Em oposição, *lightweight manycores* não dispõem de centenas de GBs de memória, mas sim pequenas memórias locais. Isso associado a outras simplificações de *hardware* faz com que algumas técnicas de virtualização sejam impraticáveis nesses ambientes computacionais.

Nesse contexto, visando atenuar o impacto da virtualização no sistema de memória, o presente trabalho explora um modelo de virtualização mais leve, baseado em contêineres adaptado para *lightweight manycores*. O SO executa os contêineres como aplicações virtuais. Sendo assim, não há a necessidade de um SO convidado, resultando em um menor impacto no sistema de memória e requisitando menor complexidade do *hardware* (THALHEIM et al., 2018; SHARMA et al., 2016).

4.1 VIRTUALIZAÇÃO NO NANVIX

O foco deste trabalho é desacoplar a execução de um processo do *cluster* em que ele é alocado, tornando possível a execução do processo em qualquer *cluster*. Para isso, é necessário que seja introduzido o conceito de virtualização no Nanvix.

É importante destacar que o Nanvix é um SO para *lightweight manycores*, os quais apresentam um sistema de memória restritivo, com memória local pequena. Isso torna difícil a virtualização, pois a criação de uma duplicata inteira do SO para cada processo é inviável. Sendo assim, a utilização de contêineres se torna atrativa.

Na abordagem original do Nanvix, o processo é dependente do *cluster* em que é alocado, o que afeta o suporte a migração e diminui a eficiência computacional, como detalhado na Seção 2.2. Nesse contexto, a virtualização torna-se útil por aumentar a mobilidade dos processos, o que possibilitaria o gerenciamento da distribuição dos processos no processador. Especificamente, este trabalho explora um modelo mais leve de virtua-

lização para *lightweight manycores* baseada em contêineres. Contêineres são executados pelo SO como aplicações virtuais e não incluem um SO convidado, não sendo necessária a criação de duplicatas de SOs e resultando em um menor impacto no sistema de memória e requisitando menor complexidade do *hardware* (THALHEIM et al., 2018; SHARMA et al., 2016; ZHANG et al., 2018).

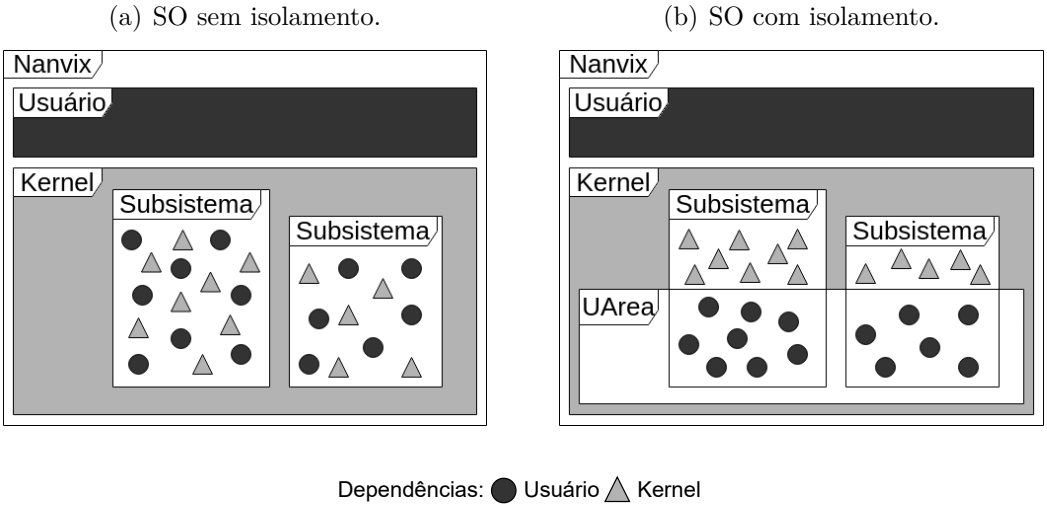
4.2 CONTEXTO DE UM PROCESSO NO NANVIX

Para o desenvolvimento da virtualização, é necessário entender o contexto de um processo no Nanvix e as relações que atrelam o processo ao *cluster* e ao SO i.e., as dependências que o processo tem com os recursos reais de um *cluster* e com a estrutura interna do SO. De maneira geral, os módulos que compõem um processo são: *threads*, *syscalls*, sistema de memória e comunicação. Todos esses módulos de alguma forma têm dependências no *kernel* ou *cluster*. Essas dependências são ilustradas genericamente na Figura 12(a) e algumas delas estão listadas abaixo:

(i) As estruturas das *threads* de usuário estão armazenadas em listas internas de *kernel*, assim como variáveis de sincronização (para junção de *threads*, por exemplo), estruturas de escalonamento, referências às pilhas de execução e outras variáveis/estrutura de controle. É importante destacar que na abordagem inicial do Nanvix não havia separação explicita nessas estruturas para identificar quais variáveis são relacinadas a *kernel* ou usuário.
(ii) As estruturas responsáveis por armazenar as *syscalls*, seus parâmetros e retornos requisitadas pelos *slave cores* ao *master core* estão em espaço de *kernel*.
(iii) Todo o sistema de memória está armazenado no *kernel*. Tabelas de diretórios, tabelas de página, *Translation Lookaside Buffer*s (TLBs), etc.
(iv) O sistema de comunicação tem dependências tanto no *kernel* quanto a recursos físicos do *cluster*. Os identificadores das interfaces *Network-on-Chip* (NoC) i.e., de comunicação entre *clusters* estão armazenadas em espaço de *kernel* e referenciam uma interface física dos *clusters* envolvidos na comunicação (emissor e receptor) e não aos processos envolvidos diretamente.

Sendo assim, o desenvolvimento da virtualização nesse sistema implica no isolamento dessas dependências em um arranjo que guarde todas as informações necessárias para a execução de um processo, sejam elas manipuladas pelo usuário ou pelo *kernel*. Chamamos esse arranjo de contêiner. A ideia principal é garantir que os dados incluidos no contêiner sejam suficientes para o processo executar. Isso inclui todos os dados e códigos de usuário e todas as dependências do processo com o *kernel* e *cluster*. Isso deve ser feito de uma maneira que permita com que o *kernel* execute qualquer contêiner como uma aplicação virtual i.e., deve ser possível que o contêiner se conecte ao *kernel* de forma

Figura 12 – Diferença da estrutura do Nanvix com e sem a *User Area*.



Fonte: Desenvolvido pelo autor.

que consiga utilizar os recursos e serviços de *kernel* sem que interfira em sua estrutura interna.

4.3 ISOLAMENTO DO CONTEXTO DE UM PROCESSO DE USUÁRIO

Para a virtualização de processos através da conteinerização, é recomendável que as informações relevantes para a manipulação dos processos em execução estejam isoladas das informações internas do próprio SO para que os recursos de *hardware* sejam utilizados de maneira eficiente (CHOUDHARY et al., 2017). A Figura 12(a) ilustra como os subsistemas do Nanvix são originalmente estruturados. Não há uma divisão explícita do que são dados para funcionamento interno do SO ou dependências locais do processo. Esta abordagem torna algumas das funcionalidades do SO onerosas porque ela dificulta o acesso às informações do processo e impacta partes independentes do sistema, e.g., migração e segurança dos processos.

4.3.1 Divisão de Dados e Instruções

A geração original de um executável do Nanvix compila todos os níveis em bibliotecas estáticas (*Hardware Abstraction Layer* (HAL), *microkernel*, *libnanvix*, *ulibc* e *multikernel*) e as junta com a aplicação do usuário de forma a misturar o que é *kernel* do que é usuário. Visando a separação das informações entre usuário e *kernel*, nós adaptamos o *script* de ligação original do Nanvix. Na nova versão, as seções .text, .data, .bss e .rodata dos arquivos binários compilados são renomeados, especificando a qual camada de abstração tal arquivo pertence. Desta forma, é possível identificar dados e instruções de cada camada do Nanvix, assim como as informações do usuário.

Sendo assim, são geradas seções .text, .data, .rodata e .bss específicas para o *kernel* e usuário. Portanto, todas as informações de *kernel*, alocadas nos endereços mais baixos da memória, são isoladas das informações de aplicação, alocadas nos endereços mais altos da memória. Neste processo, são exportadas algumas constantes que apontam onde começam e terminam as partes do binário que são relacionadas ao *kernel* e à aplicação e.g., *___KERNEL_TEXT_START*, *___KERNEL_TEXT_END*, *___USER_DATA_START*, *___USER_START*, etc. Essas constantes permitem a manipulação e gerenciamento mais precisos dos segmentos de memória do *kernel* e da aplicação.

Essa estratégia, além de garantir o isolamento do binário de *kernel* e usuário, faz com que todos os *clusters* passam a ter a mesma organização interna de *kernel*, haja vista que a compilação é estática. Nesse cenário, a migração pode ser feita parcialmente através do salvamento dos dados e instruções da aplicação de um *cluster*, os quais estão contidos no intervalo identificado pelas constantes, e restauração destes nas respectivas posições i.e., no mesmo intervalo em outro *cluster*. Com isso, evita-se manipulações mais complexas do processo como a busca em várias regiões de memória para montar o estado interno do processo.

4.3.2 *User Area*

Além da separação de dados e instruções entre *kernel* e aplicação, é necessário a identificação e separação das estruturas internas do SO que são manipuladas pelo usuário e constituem o estado interno do processo. Nesse contexto, é introduzido o conceito de conteinerização para isolar as dependências que o usuário possui dentro do *cluster*. Ou seja, nós isolamos os dados que são gerenciados pelo *kernel* mas pertencem ao contexto do processo de usuário. Neste contexto, nós isolamos tais dados em uma região de memória bem definida, denominada de *User Area* (UArea).

Detalhadamente, a UArea mantém informações sobre:

(i) *Threads* ativas, incluindo identificadores, pilhas de execução e contextos;
(ii) Filas de escalonamento de *threads*;
(iii) Variáveis de controle interno do sistema de *threads*, como quantidade de *threads* ativas;
(iv) Tabela de gerenciamento de chamadas de sistema; e
(v) Estruturas de gerenciamento de memória e.g., sistema de paginação

Essa estrutura genérica foi projetada para englobar as várias arquiteturas suportadas pelo Nanvix. Além disso, a estrutura permite a modificação e expansão, não se limitando ao estado atual do desenvolvimento do Nanvix, para atender os objetivos de outros projetos que usufruam do Nanvix.

4.4 CONTEINERIZAÇÃO

O uso de contêineres faz-se presente neste trabalho, porém seu funcionamento difere das abordagens convencionais. De maneira geral, nas perspectivas tradicionais, um contêiner é uma estrutura responsável por isolar do SO a execução de uma aplicação que roda sob seus limites. Esta estrutura geralmente é construída a partir de uma imagem base de um SO, na qual podem ser instaladas novas bibliotecas ou ferramentas, de modo a permitir a customização do ambiente de execução da aplicação de acordo com as necessidades do usuário. Sendo assim, o contêiner pode executar um SO diferente de onde está sendo executado e também pode conter ferramentas/bibliotecas diferentes ou em uma versão diferente das do SO que executa o contêiner.

Em contraste, a proposta neste trabalho utiliza o conceito de contêiner de uma maneira diferente. A proposta neste trabalho é apenas isolar o processo do *kernel* e do *cluster* que o executa a fim de permitir maior mobilidade do processo no processador. Sendo assim, os contêineres não contêm imagens de SO distintas ou funcionalidades diferentes uns dos outros. Todos os contêineres executam sobre a mesma estrutura de *kernel* (que é idêntica em todos os *clusters* do *lightweight manycore*), apenas se diferenciando pelo estado interno do processo que está sendo executado e.g., quantidade de *threads* criadas, dados manipulados pelo usuário, espaços de memória alocados dinamicamente, etc.

Nesse cenário, um contêiner neste trabalho pode ser definido como a junção do código de usuário, dados de usuário e UArea. Essa união abrange todo o essencial para a execução do processo: código e dados de usuário; e as dependências internas do processo no *cluster* que o executa.

4.5 MIGRAÇÃO DE PROCESSOS

Como aplicação direta do isolamento do processo, conseguida através da virtualização com os contêineres, a migração de processos torna-se mais palpável. Especificamente, nós eliminamos a necessidade de descobrir quais são e onde estão as informações que compõem o estado de um processo dentro do Nanvix porque tudo está isolado via conteinerização, facilitando a transferência de seu contexto. Isso só é possível porque os *clusters* possuem uma estrutura de *kernel* idêntica (devido às mudanças desenvolvidas no processo de compilação detalhados na Subseção 4.3.1). Por este motivo, eliminamos o envio de dados redundantes entre *clusters* referentes à instância local do SO, atenuando o impacto da migração sobre a NoC.

4.5.1 Rotina de migração

Para a migração de um processo entre *clusters* foi desenvolvida uma rotina de migração. A funcionalidade é similar ao *Checkpoint/Restore In Userspace* (CRIU), ferramenta utilizada por *softwares* de gerenciamento de contêineres como o Docker. Porém, a migração é executada por intermédio de *daemons* do SO. Neste projeto, foi implementado o algoritmo *hot migration*, em que a aplicação é migrada enquanto é executada, como detalhado na Subsubseção 2.4.1.2 utilizando a técnica *pre-copy* visto na subseção 2.4.1.2.1. De forma resumida, a aplicação é migrada durante sua execução, sendo restaurada no *cluster* destinatário após a transferência completa dos dados do contêiner. A seguir é detalhado como funciona o *daemon* e o fluxo de migração.

4.5.1.1 Daemon de Migração

O *daemon* de migração é inicializado durante o *boot* do sistema. Resumidamente, o *daemon* é composto por um fluxo de *tasks*, as quais são inicializadas e conectadas durante a inicialização do módulo de migração, que ocorre durante o *boot*. Além disso, nesse período ainda é criada a porta de *Mailbox* por onde o *daemon* recebe as requisições de migração.

O fluxo de migração de inicia com o recebimento de uma requisição de migração. Quando uma requisição é detectada i.e., quando é lida uma mensagem de migração da porta específica do *daemon*, a *task* principal do *daemon* (o *handler* do *daemon* de migração) é ativa. Esta função é responsável por interpretar a mensagem recebida. É neste momento em que os *clusters* envolvidos são identificados i.e., é reconhecido qual é o *cluster* remetente e qual o *cluster* destinatário. É importante destacar que tanto o *cluster* remetente quanto o destinatário recebem a mesma mensagem, porém com códigos de operação diferentes. Enquanto um recebe uma mensagem com o código de envio dos dados (identificando o remetente), o outro recebe uma mensagem com o código de recebimento de dados (identificando o destinatário).

Depois da identificação dos *clusters* envolvidos e seus papéis durante o procedimento, são executadas as *tasks* responsáveis pela migração de fato i.e., pelo envio e recebimento de dados.

4.5.1.2 Fluxo de Migração

Genericamente, podemos dizer que o fluxo de migração é composto por três passos principais, os quais estão descritos abaixo:

1. Congelamento da execução do processo em um estado consistente.

Antes do envio da aplicação a outro *cluster*, é necessário que o processo esteja em um estado consistente e estático. Isso significa que durante o processo de migração é preciso que todas as operações dele sejam pausadas. Isso é feito objetivando evitar inconsistências que podem ser causadas por condições de corrida e.g., impedir perda de instruções, impedir perda de retornos de chamadas de sistemas, impedir perda de sinais de sincronização, impedir inconsistência de valores de variáveis de usuário, etc. Para atingir esse estado consistente, foi criada a chamada de sistema *freeze*, a qual é invocada no início do processo de migração. Esta é uma chamada de sistema que é tratada apenas pelo *master core*. Especificamente, esta chamada ativa uma variável interna do SO que impede o escalonamento de *threads* de aplicação (*threads* que não executam no *master core*) e envia um sinal de reescalonamento para todos os *slave cores*, para que as *threads* de usuário saiam de execução o mais rápido possível. Isso garante uma pausa na aplicação sem que o SO seja impedido de executar, o que é imprescindível para a migração, já que as informações do processo precisam ser enviadas pelas interfaces NoC do *cluster* remetente, o que exige que o SO atenda às requisições de envio de dados. Isso só é possível se as *threads* de sistema continuarem a ser executadas apesar do congelamento das de usuário. Após o travamento no escalonamento de *threads* de usuário, novas chamadas de sistema requisitadas pela aplicação não podem ocorrer. Sendo assim, após a migração, o *cluster* destinatário atenderá às chamadas não atendidas e reconhecerá as atendidas, pois as estruturas de sincronização e variáveis de retorno são migradas também durante o processo. Após o congelamento do escalonamento e a retirada das *threads* de usuário dos *slave cores*, o processo é considerado consistente e seu contexto está apto para ser migrado.

2. **Transferência do contexto do processo entre *clusters*.**
   Com o processo em um estado consistente, uma série de *tasks* de sistema, que são escalonadas no *master core*, são executadas para o envio dos dados ao *cluster* destinatário. O envio é feito através da abstração de comunicação *Portal*, que permite transferência de grandes quantidades de dados. O envio de dados, instruções e UArea garantem que o contexto inteiro do processo seja enviado, possibilitando a retomada da execução no *cluster* destinatário.
3. **Restauração da execução do processo no *cluster* destino.**
   Com o contexto do processo já no *cluster* destinatário, a execução é restaurada. Isso é feito pela chamada de sistema *unfreeze*, que descongela o escalonamento de *threads* de usuário. Assim, a execução do processo continua normalmente, agora em outro *cluster*.

Mais detalhadamente, neste trabalho foi utilizado um sistema de *tasks* que em conjunto constroem essas três etapas apresentadas. A Figura 13 ilustra o fluxo de execução da migração. Nela, cada quadro corresponde a uma *task* e a escrita que cada quadro

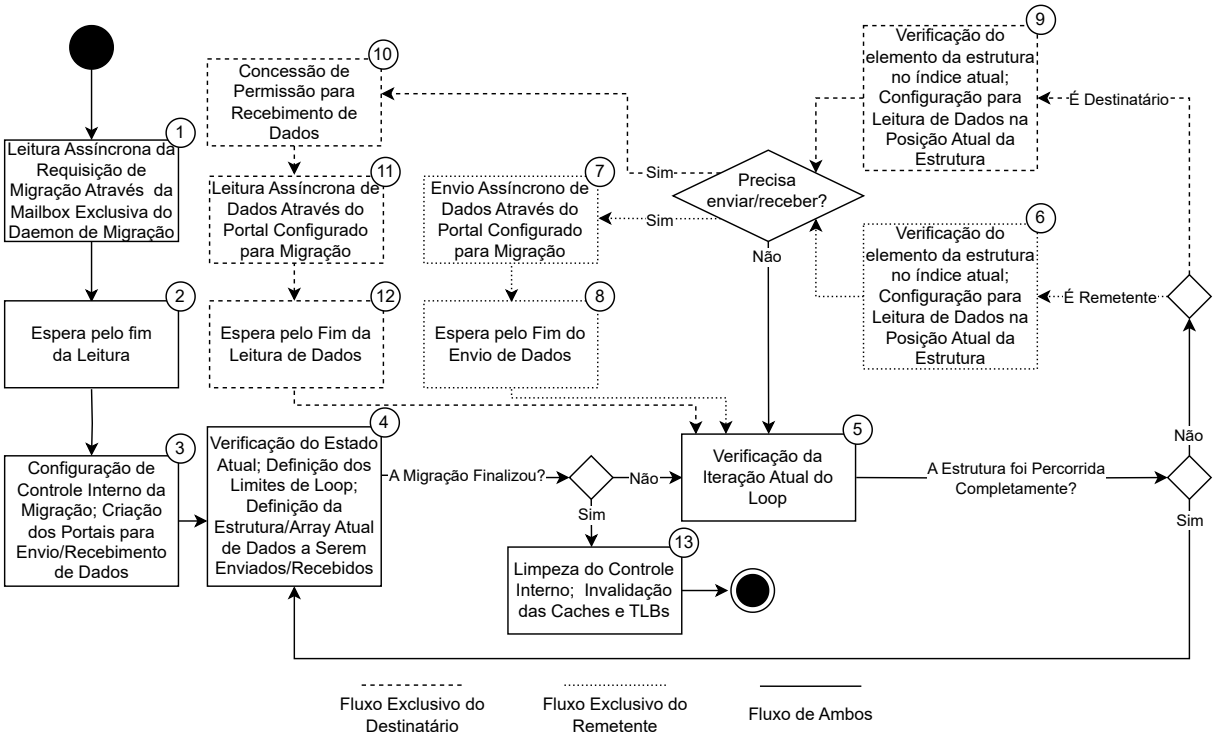
indica o que a *task* faz. Destaca-se que neste fluxo todas as comunicações são assíncronas. O que significa que em nenhum momento uma *task* espera ativamente pelo término de uma comunicação. Isso é feito com o intuito de evitar que o sistema seja bloqueado por alguma *task* que esteja esperando por uma comunicação. Dito isso, esse *design* de migração baseado em *tasks* foi adotado por garantir o isolamento das etapas de migração em passos bem definidos e porque o funcionamento da comunicação assíncrona está intrinsecamente ligado ao sistema de *tasks* do Nanvix. Com mais profundidade, uma comunicação assíncrona funciona da seguinte maneira: uma *task* envia mensagens de tamanho fixo repetidas vezes até que se atinja a quantidade de *bytes* passada como parâmetro. Quando a mensagem é enviada completamente, é enviado um sinal a outra *task*, cuja função é sinalizar o final de uma comunicação. As estruturas de ambas as *tasks* são criadas e configuradas de acordo com a funcionalidade desejada.

Portanto, tendo em vista que a comunicação assíncrona fundamenta-se na utilização de *tasks*, é natural e mais simples que o *daemon* de migração seja composto por *tasks*. Desse modo as *tasks* de início e fim das comunicações são conectadas às *tasks* de controle da migração.

Abaixo estão descritos o funcionamento mais detalhado de cada *task* de migração de acordo com o número identificador ilustrado na Figura 13.

1. Esta é a *task* responsável pela leitura assíncrona de uma mensagem enviada ao

Figura 13 – Fluxo de tasks de migração



Fonte: Desenvolvido pelo autor.

daemon de migração. Esta *task* é ativa no momento de *boot* durante a etapa de configuração do sistema de migração e é reativada no momento em que uma migração finaliza com ou sem êxito i.e., assim que uma migração finaliza seja por ter completado todo seu processo ou por ocorrência de algum erro. Isso permite que o fluxo seja reusado, o que significa que o *cluster* pode atender múltiplas requisições de migração sequêncialmente independentemente do papel do *cluster* (remetente ou destinatário).

2. Esta *task* é ativa no momento em que uma mensagem é lida pela *task* 1. Portanto, esta é a *task* responsável por sinalizar que uma mensagem foi recebida e lida.

3. Esta é a *task* que interpreta a mensagem recebida. A mensagem é composta por um código de operação que identifica o papel do cluster na migração e por dois inteiros que identificam os *clusters* envolvidos na migração. Caso o *cluster* seja o destinatário, durante a execução desta *task* é invocada a chamada de sistema *freeze* para congelar a execução do processo de usuário (neste momento o *cluster* remetente já deve ter congelado sua execução também). Caso o *cluster* seja o remetente, é criado um *Portal* para envio de dados, os quais serão recebidos pelo destinatário pelo *Portal default* de recebimento de dados de migração (configurado no momento de *boot*). Essa *task* passa um parâmetro indicando o papel do *cluster* na migração. Nas etapas posteriores esse parâmetro é decisivo na escolha dos subfluxos que o procedimento deve seguir e.g., envio ou recebimento de dados.

4. Esta *task* é responsável pelo gerenciamento do estado atual do envio/recebimento de dados da migração. No total são 10 estados. Abaixo estão os estados e as estruturas que são manejadas em cada um deles:

MSTATE_SECTIONS	Seções do binário identificadas como partes do usuário: `.user.text`, `.user.data`, `.user.bss` e `.user.rodata`.
MSTATE_UAREA	UArea.
MSTATE_SYSBOARD	Tabela de chamadas de sistema.
MSTATE_PAGEDIR	Tabela de diretórios.
MSTATE_PAGETAB	Tabela de páginas.
MSTATE_KSTACKSIDS	Lista responsável pela identificação de quais páginas de *kernel* estão sendo usadas.
MSTATE_KSTACKSPHYS	Páginas de *kernel* que estão sendo usadas.
MSTATE_FRAMES_BITMAP	*Bitmap* responsável pela identificação de quais *frames* estão sendo usados.
MSTATE_FRAMES_PHYS	*Frames* de usuário que estão sendo usados
MSTATE_FINISH	Não manipula estruturas. Apenas reseta o controle interno da migração e invalida TLBs e *caches*.

De acordo com o estado atual, são configurados os *buffers* para o envio/recebimento de dados (o que ocorre respectivamente nas *tasks* 6 e 7), os limites inferior e superior do *loop* que percorre o *buffer* (na *task* 5) e uma função de condição que verifica se elemento em dado índice do *buffer* atual precisa ser enviado.

5. Esta *task* é responsável por atualizar o índice atual do *buffer* configurado na *task* 4 e indicar se a estrutura foi completamente percorrida ou não. Caso o limite superior, também configurado pela *task* 4, foi atingido, a *task* 4 á ativa, atualizando o estado atual. Caso contrário, um novo índice é considerado para o envio/recebimento de dados.

6. Esta *task* é ativa desde que o *cluster* exerça o papel de remetente na migração. A *task* é responsável por verificar se a posição atual na estrutura atual precisa ser enviada. Essa checagem é feita através da função de condição configurada na *task* 4. Esse comportamento é muito útil quando consideramos a lista de páginas de *kernel*, por exemplo. Nesse cenário a verificação (através da função de condição) de quais páginas precisam ser enviadas evita o envio de dados não essenciais ou redundantes. Caso o elemento do *buffer* configurado exija a transferência de dados, a *task* 7 é ativada para tal. Caso contrário, a *task* 5 é ativada para atualizar o índice para o próximo elemento do *buffer*.

7. Esta *task* é responsável pelo envio assíncrono dos dados do *buffer* no índice configurado. Todos os parâmetros para o envio dos dados, tais como o ponteiro para o dado, a quantidade de *bytes* a serem enviadas e o identificador do portal a ser usado, são passados como parâmetro pela *task* 6.

8. Esta *task* é responsável por sinalizar o fim do envio dos dados, ativando a *task* 5 novamente para atualizar o índice do *buffer*.

9. Esta *task* é ativa desde que o *cluster* exerça o papel de destinatario na migração. A *task* é responsável por verificar se a posição atual na estrutura atual precisa ser recebida. Em caso positivo, a *task* 10 é ativada para o recebimento dos dados. Caso contrário, a *task* 5 é ativada para atualizar o índice para o próximo elemento do *buffer*. Essa chegagem é feita de forma semelhante à *task* 6 i.e., através da função de condição configurada na *task* 4.

10. Esta *task* é responsável pela concessão de permissão para o recebimento de dados. Sem essa permissão os dados não poderiam ser lidos.

11. Esta *task* é responsável pelo recebimento assíncrono dos dados do *buffer* no índice configurado. Todos os parâmetros para o recebimento dos dados, tais como o ponteiro para o dado, a quantidade de *bytes* a serem recebidos e o identificador do portal a ser usado, são passados como parâmetro pela *task* 9.

12. Esta *task* é responsável por sinalizar o fim do recebimento dos dados, ativando a *task* 5 novamente para atualizar o índice do *buffer*.

13. Esta *task* é responsável por resetar o controle interno da migração e invalidar TLBs e *caches*. Essa *task* é ativada quando o estado atual é MSTATE_FINISH.

4.5.1.3 Interface com o Daemon

A funcionalidade de migração que o *daemon* de migração provê é acessível através de uma função de sistema chamada `kmigrate_to`. Essa função recebe como parâmetro apenas o identificador do *cluster* para o qual o processo deve ser migrado.

O procedimento da função `kmigrate_to` é detalhado a seguir. Primeiramente, uma *task* é criada para a construção das mensagens de migração e para o envio dessas mensagens às portas *Mailbox* dos *daemons* de migração dos *clusters* envolvidos. Essa *task* envia uma mensagem de migração para o próprio *cluster* que invocou a função `kmigrate_to` (o remetente) e para o *cluster* identificado pelo parâmetro (o destinatário).

É interressante destacar que o *daemon* foi projetado possibilitando expansão das suas funcionalidades sem maiores complicações. Como exemplos de trabalhos futuros temos: implementação de uma funcionalidade em que um terceiro *cluster* peça a migração de um processo entre outros *clusters* distintos; Desenvolvimento de um serviço de escalonamento de processos que considere a melhor distribuíção dos processos entre os *clusters* do processador; implementação de um sistema de *checkpointing* de processos, visando o salvamento do estado de um processo em disco. Todos essas funcionalidades podem utilizar o *daemon* de migração como suporte para suas responsabilidades específicas.

5 METODOLOGIA DE AVALIAÇÃO

Neste capítulo, será apresentado como a solução foi avaliada. Particularmente, as perguntas que guiaram o desenvolvimento dos experimentos desenvolvidos para analisar a virtualização e a migração de processos no Nanvix foram:

(i) Qual o impacto do isolamento da *User Area* (UArea) e do código e dados de usuário sobre a execução do Nanvix?
(ii) Qual a eficiência da migração de processos no Nanvix de acordo com a quantidade de estruturas manipuladas pela aplicação?
(iii) Há sobrecarga no sistema de comunicação quando migramos apicações paralelamente?

Para responder a primeira pergunta, foram desenvolvidos experimentos sobre a manipulação de *threads* no Nanvix, que é o principal subsistema afetado pela UArea. O experimento mensura os impactos na criação e junção de *threads* através de diferentes perspectivas. Trata-se de um teste em que causamos um estresse no subsistema de *threads* através da criação e junção do máximo de *threads* que o sistema suporta. Especificamente, coletamos o tempo de execução, desvios e faltas ocorridas na *cache* de dados e de instrução.

Para responder a segunda pergunta, foram desenvolvidos experimentos sobre a migração de processos no Nanvix. O experimento mensura o tempo de transferência de um processo entre *clusters* de acordo com os recursos utilizados. Neste teste, variamos a quantidade de páginas de memória dinamicamente alocadas entre 0 e 32; e *threads* usadas pela aplicação entre 1 e 17. Em resumo, neste experimento avaliamos como ocorre a progressão do tempo de transferência de um processo desde o mínimo de recursos utilizados (1 *thread* e 0 páginas dinamicamente alocadas) até o máximo (17 *threads* e 32 páginas dinamicamente alocadas).

Para responder a terceira pergunta, foi feito outro teste que mensura o *down time* da aplicação em diversos cenários. Neste teste, mensuramos o *down time* variando a quantidade de processos migrados paralelamente desde o mínimo (1 processo, envolvendo 2 *clusters*) até o máximo (8 processos, envolvendo 16 *clusters*).

Adicionalmente, um quarto teste foi feito com o objetivo de garantir a capacidade do *daemon* suportar múltiplas migrações. Neste experimento, um processo é migrado de *cluster* em *cluster* até que percorra todos os *clusters* do processador i.e., o processo é migrado realizando um movimento circular, passando do *cluster* 0 para o 1, do 1 para o 2, e assim sucessivamente até retornar ao *cluster* 0.

Todos os testes foram realizados no processador Kalray MPPA-256. Realizamos múltiplas replicações para garantir maior confiança estatística (100 replicações para o primeiro experimento e 20 replicações para o segundo e terceiro experimento). Para a medição de tempo no segundo e terceiro experimento, utilizamos a abstração de comunicação *Sync*. Como as estruturas utilizadas por cada experimento variam, a quantidade de

dados enviada também varia. Contudo, podemos generalizar a quantidade de *bytes* enviada através da Equação 5.1. Na Equação 5.1 o identificador `U` representa o tamanho do binário de usuário, `UA` o tamanho da UArea, `SYSB` o tamanho da tabela de gerenciamento das chamadas de sistema, `PGDIR` o tamanho da tabela de diretórios de páginas, `PGTAB` o tamanho da tabela de página, `KSIDS` o tamanho da lista de gerenciamento de páginas de *kernel*, `KSPHYS` a soma do tamanho das páginas de *kernel* em uso, `FBMP` o tamanho do *bitmap* de gerenciamento dos *frames* e `FPHYS` a soma do tamanho dos *frames* usados pela alocação dinâmica de páginas de memória.

$$U+UA+SYSB+PGDIR+PGTAB+KSIDS+KSPHYS+FBMP+FPHYS \quad (5.1)$$

Algumas dessas variáveis são constantes nos experimentos. Substituindo-as pelos seus respectivos valores em *bytes* obtemos a Equação 5.3.

$$10608 + 2112 + 1920 + 4096 + 4096 + 160 + KSPHYS + 16 + FPHYS \quad (5.2)$$

$$23008 + KSPHYS + FPHYS \quad (5.3)$$

Sendo assim, percebemos que a quantidade de dados enviada varia em função da quantidade de páginas de *kernel* em uso e da quantidade de páginas alocadas dinâmicamente. Tendo em vista que as variáveis do experimento são a quantidade de *threads* e a quantidade de páginas alocadas dinamicamente, é importante entender como essas variáveis impactam em `KSPHYS` e `FPHYS`. Em mais detalhes, quando uma *thread* é criada, duas páginas de *kernel* são alocadas para esta nova *thread*: uma para a pilha de execução em espaço de usuário; e outra para a pilha de execução em espaço de *kernel*. Já quando uma página de usuário é dinamicamente criada, aumentamos o espaço do usuário em mais uma página i.e., aumentamos o `FPHYS` em uma página (4096 B). Além disso, quando alocamos a primeira página de usuário, uma página de *kernel* é utilizada como tabela de páginas para essa e as possíveis novas páginas a serem alocadas. Arranjando esses dados na Equação 5.3, obtemos:

$$\begin{cases} 23008 + 4096(2 * NTHREADS), & \text{se } NPAGES = 0 \\ 23008 + 4096(2 * NTHREADS + NPAGES + 1), & \text{se } NPAGES > 0 \end{cases}$$

ou simplesmente:

$$23008 + 4096(2 * NTHREADS + NPAGES + min(NPAGES, 1)) \quad (5.4)$$

Onde `NTHREADS` é a quantidade de *threads* criadas e `NPAGES` a quantidade de páginas criadas.

Quanto ao funcionamento dos experimentos, em resumo, no início do teste todos os *clusters* sincronizavam entre si através do *Sync*. Neste momento, o *I/O Cluster* iniciava uma contagem de ciclos. Ao final do experimento, todos os *clusters* envolvidos sincronizavam novamente. Neste momento, o *I/O Cluster* parava a contagem de ciclos e esse era o tempo que a migração demorou até seu término. Destaca-se que nos experimentos que precisavam de algum *setup* inicial, o tempo de *setup* não foi considerado. Por exemplo, no segundo experimento, o tempo para a criação de *threads*, alocação e manipulação das páginas dinamicamente alocadas não é contabilizado no tempo final. O resultado, portanto, engloba apenas o *down time* da aplicação.

6 RESULTADOS EXPERIMENTAIS

O primeiro experimento, o qual mensurava o impacto do isolamento da aplicação através da UArea e separação do binário de usuário e *kernel*, mostrou que o subsistema de *threads* foi positivamente afetado. A Figura 14(a) ilustra que o Nanvix obteve um leve ganho de desempenho nas operações de criação e junção de *threads*. A Figura 14(b) mostra a origem do aumento da performance. Percebemos que houve uma diminuição na quantidade de faltas em *cache* (tanto de dados como de instruções) e a quantidade de desvios é menor na versão do Nanvix com a UArea. Isso ocorre porque a UArea explora melhor a localidade espacial dos dados, já que os dados estão aglomerados em um espaço menor da memória. Como consequência disso, o número de faltas na *cache* e de desvios diminui, resultando em um aumento de desempenho.

O segundo experimento, que mensura o *down time* da aplicação em cenários variando a quantidade de recursos utilizados, mostrou os seguintes resultados. A Figura 15 e a Figura 16 mostram a progressão de tempo sobre diferentes perspectivas (fixando-se as páginas e *threads*, respectivamente). Como o esperado, o *down time* aumenta quanto maior for o número de páginas e *threads*. Além disso, a quantidade de *threads* se destaca por apresentar maior expressividade no tempo contabilizado em comparação com a quantidade de páginas. Isso acontece porque uma *thread* acarreta em mais dados migrados do que uma página. De maneira geral, o *down time* é aproximadamente descrito pela função 6.1:

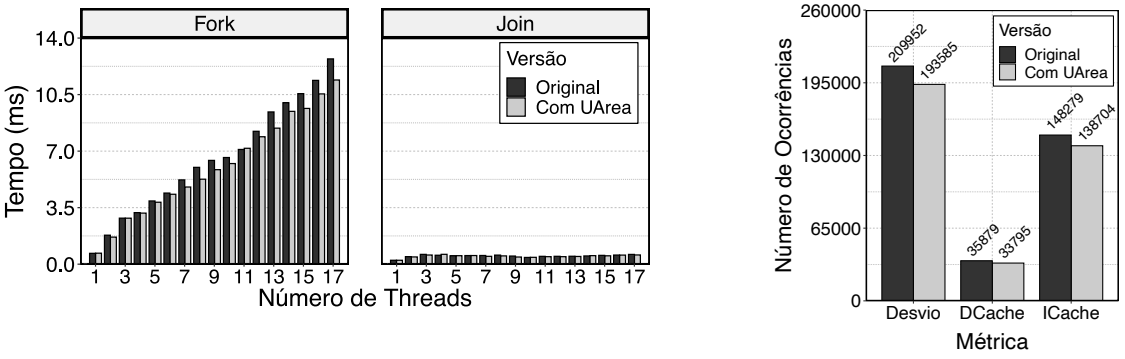
$$f(p,t) = \frac{17}{32} * p + t + 18 \tag{6.1}$$

Essa função descreve relativamente bem a progressão de tempo até 15 *threads*. Como podemos observar nos gráficos, com 16 *threads* há uma disparidade com a natureza

Figura 14 – Impactos da virtualização sobre a manipulação de *threads*.

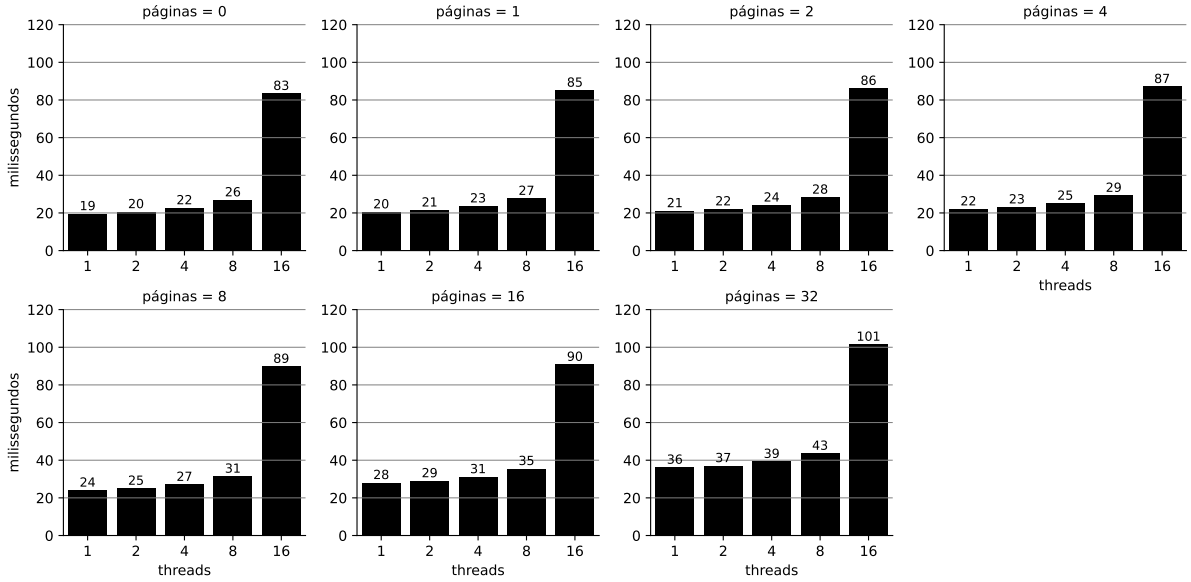
(a) Tempo de criação de *threads*.

(b) Métricas do *kernel*.



Fonte: Desenvolvido pelo autor.

Figura 15 – *Down time* da aplicação durante o experimento de migração fixando a quantidade de páginas alocadas dinamicamente



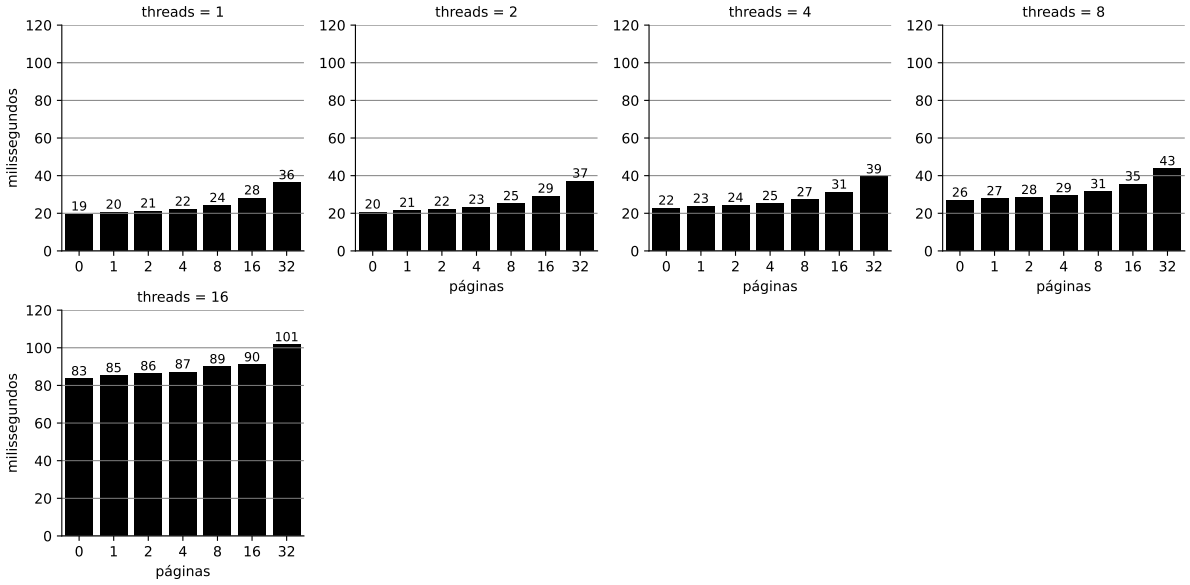
Fonte: Desenvolvido pelo autor.

aparentemente linear que a progressão de tempo aparenta ter. Isso acontece porque o Kalray MPPA-256 possui 16 *cores* por *cluster*, o que significa que simultaneamente apenas 15 *threads* de usuário podem executar, já que em um *core* executam apenas *threads* de sistema, que são responsáveis pelo tratamento de chamadas de sistema e execução das *tasks*. Portanto, essa disparidade numérica que os gráficos apresentam são justificados pelo tempo extra que o teste ocupa com gerenciamento de *threads*. Isso porque o teste finaliza com a junção de todas as *threads* criadas no *cluster* destinatário e tal processo exige tempo extra quando a quantidade de *threads* é maior que 15, uma vez que algumas *threads* precisariam esperar outras terminarem para poderem executar e, finalmente, finalizar sua execução.

O terceiro experimento, que mensurava o *down time* da aplicação quando múltiplas migrações aconteciam simultaneamente, mostrou que a quantidade de migrações paralelas não impacta significativamente o tempo. É importante destacar que nesse experimento foram migradas aplicações que usavam o máximo de recursos de sistema. Mesmo assim, a quantidade de dados não foi suficiente para impactar no *down time*.

Já o quarto experimento, o teste adicional que visava a comprovação da corretude do *daemon*, mostrou que: o *daemon* é capaz de ser reusado no mesmo *cluster* diversas vezes independentemente do papel do *cluster* na migração; e que é possível um processo ser migrado diversas vezes para diferentes *clusters*, até em movimento circular envolvendo todos os *clusters*, como no experimento.

Figura 16 – *Down time* da aplicação durante o experimento de migração fixando a quantidade de *threads*



Fonte: Desenvolvido pelo autor.

7 CONCLUSÕES

Neste trabalho, foi explorado um modelo de virtualização leve baseado em contêineres que considera as restrições arquiteturais dos *lightweight manycores*, adaptando-se as suas restrições, principalmente relacionadas à memória. A virtualização proposta visa melhorar a mobilidade e gerenciamento de processos para *lightweight manycores* no contexto de em um Sistema Operacional (SO) distribuído, o Nanvix.

Os resultados mostraram que a virtualização nesses ambientes é possível, bem como a migração de processos entre os *clusters* do processador. A migração não afeta significativamente o sistema de comunicação, sendo possível realizar múltiplas migrações simultaneamente, e provocou um *down time* relativamente baixo, já que evitamos o envio de dados redundantes e.g., relacionados ao *kernel*. O isolamento das dependências de um processo aumentaram o desempenho de operações do *kernel* na execução normal do SO, em especial no subsistema de *threads*.

7.1 TRABALHOS FUTUROS

Como trabalhos futuros, pretende-se:

(i) Ampliar a virtualização, englobando o subsistema de comunicação;
(ii) Habilitar a execução simultânea de múltiplas aplicações no processador e sua proteção;
(iii) Implementar um escalonador no Nanvix que considere a melhor distribuição de processos entre os *clusters*;
(iv) Implementar um sistema de *checkpointing* que torne possível guardar estados de processos em disco;

REFERÊNCIAS

ALDOSSARY, M.; DJEMAME, K. Performance and energy-based cost prediction of virtual machines live migration in clouds. In: **CLOSER**. [S.l.: s.n.], 2018. p. 384–391.

AMROUCH, H. et al. Negative capacitance transistor to address the fundamental limitations in technology scaling: Processor performance. **IEEE Access**, IEEE, v. 6, p. 52754–52765, 2018.

ASMUSSEN, N. et al. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In: **ASPLOS '16 Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems**. ACM. (ASPLOS '16, v. 44), p. 189–203. ISBN 978-1-4503-4091-5. Disponível em: http://dl.acm.org/citation.cfm?doid=2980024.2872371.

BARBALACE, A. et al. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In: **Proceedings of the Tenth European Conference on Computer Systems**. [S.l.: s.n.], 2015. p. 1–16.

CAMPBELL, S.; JERONIMO, M. An introduction to virtualization. **Published in "Applied Virtualization", Intel**, p. 1–15, 2006.

CASTRO, M. et al. Seismic wave propagation simulations on low-power and performance-centric manycores. **Parallel Computing**, v. 54, p. 108–120, 2016. ISSN 01678191. Disponível em: http://www.sciencedirect.com/science/article/pii/S0167819116000417.

CHOUDHARY, A. et al. A critical survey of live virtual machine migration techniques. **Journal of Cloud Computing**, SpringerOpen, v. 6, n. 1, p. 1–41, 2017.

CLARK, C. et al. Live migration of virtual machines. In: **Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2**. [S.l.: s.n.], 2005. p. 273–286.

COMBE, T.; MARTIN, A.; PIETRO, R. D. To docker or not to docker: A security perspective. **IEEE Cloud Computing**, IEEE, v. 3, n. 5, p. 54–62, 2016.

CROSBY, M. **What is containerd ?** 2017. Disponível em: https://www.docker.com/blog/what-is-containerd-runtime/.

DEVI, L. Y. et al. Security in virtual machine live migration for kvm. In: IEEE. **2011 International Conference on Process Automation, Control and Computing**. [S.l.], 2011. p. 1–6.

DINECHIN, B. D. de et al. A clustered manycore processor architecture for embedded and accelerated applications. In: **2013 IEEE High Performance Extreme Computing Conference (HPEC)**. [S.l.: s.n.], 2013. p. 1–6.

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

FERNANDO, D. et al. Live migration ate my vm: Recovering a virtual machine after failure of post-copy live migration. In: IEEE. **IEEE INFOCOM 2019-IEEE Conference on Computer Communications**. [S.l.], 2019. p. 343–351.

FRANCESQUINI, E. et al. On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. **Journal of Parallel and Distributed Computing (JPDC)**, v. 76, n. C, p. 32–48, fev. 2015. ISSN 0743-7315. Disponível em: http://linkinghub.elsevier.com/retrieve/pii/S0743731514002093.

FU, H. et al. The sunway taihulight supercomputer: system and applications. **Science China Information Sciences**, Springer, v. 59, n. 7, p. 1–16, 2016.

FULLER, S. H.; MILLETT, L. I. Computing performance: Game over or next level? **Computer**, IEEE, v. 44, n. 1, p. 31–38, 2011.

GAO, X. et al. Containerleaks: Emerging security threats of information leakages in container clouds. In: IEEE. **2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.], 2017. p. 237–248.

GEPNER, P.; KOWALIK, M. F. Multi-core processors: New way to achieve high system performance. In: IEEE. **International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)**. [S.l.], 2006. p. 9–13.

HU, L. et al. Magnet: A novel scheduling policy for power reduction in cluster with virtual machines. In: IEEE. **2008 IEEE International Conference on Cluster Computing**. [S.l.], 2008. p. 13–22.

IMRAN, M. et al. Live virtual machine migration: A survey, research challenges, and future directions. **Computers and Electrical Engineering**, Elsevier, v. 103, p. 108297, 2022.

KAHUHA, E. **LXC vs Docker: Which Container Platform Is Right for You?** 2023. Disponível em: https://earthly.dev/blog/lxc-vs-docker/.

KARHULA, P.; JANAK, J.; SCHULZRINNE, H. Checkpointing and migration of iot edge functions. In: **Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking**. [S.l.: s.n.], 2019. p. 60–65.

KELLY, B.; GARDNER, W.; KYO, S. AutoPilot: Message Passing Parallel Programming for a Cache Incoherent Embedded Manycore Processor. In: **Proceedings of the 1st International Workshop on Many-core Embedded Systems**. Tel-Aviv, Israel: ACM, 2013. (MES '13), p. 62–65. ISBN 978-1-4503-2063-4. Disponível em: http://dl.acm.org/citation.cfm?doid=2489068.2491624.

KLUGE, F.; GERDES, M.; UNGERER, T. An operating system for safety-critical applications on manycore processors. In: **2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing**. IEEE. (ISORC '14), p. 238–245. ISBN 978-1-4799-4430-9. Disponível em: http://ieeexplore.ieee.org/document/6899155/.

KOGGE, P. et al. Exascale computing study: Technology challenges in achieving exascale systems. **Defense Advanced Research Projects Agency Information**

Processing Techniques Office (DARPA IPTO), Techinal Representative, v. 15, 01 2008.

MANOHAR, N. A survey of virtualization techniques in cloud computing. In: SPRINGER. **Proceedings of international conference on vlsi, communication, advanced devices, signals & systems and networking (vcasan-2013)**. [S.l.], 2013. p. 461–470.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, v. 38, n. 8, April 1965.

MORABITO, R. et al. Lightweight virtualization as enabling technology for future smart cars. In: **2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. [S.l.: s.n.], 2017. p. 1238–1245.

NAGARAJAN, A. B. et al. Proactive fault tolerance for hpc with xen virtualization. In: **Proceedings of the 21st annual international conference on Supercomputing**. [S.l.: s.n.], 2007. p. 23–32.

PENNA, P. H. **Nanvix: A Distributed Operating System for Lightweight Manycore Processors**. Tese (Doutorado) — Université Grenoble Alpes, 2021.

PENNA, P. H. et al. Using the Nanvix Operating System in Undergraduate Operating System Courses. In: **2017 VII Brazilian Symposium on Computing Systems Engineering**. Curitiba, Brazil: IEEE, 2017. (SBESC '17), p. 193–198. ISBN 978-1-5386-3590-2. Disponível em: http://ieeexplore.ieee.org/document/8116579/.

PENNA, P. H.; FRANCIS, D.; SOUTO, J. The Hardware Abstraction Layer of Nanvix for the Kalray MPPA-256 Lightweight Manycore Processor. In: **Conférence d'Informatique en Parallélisme, Architecture et Système**. Anglet, France: [s.n.], 2019. Disponível em: https://hal.archives-ouvertes.fr/hal-02151274.

PENNA, P. H. et al. On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In: **SBESC 2019 - IX Brazilian Symposium on Computing Systems Engineering**. Natal, Brazil: [s.n.], 2019.

PENNA, P. H. et al. Inter-kernel communication facility of a distributed operating system for noc-based lightweight manycores. **Journal of Parallel and Distributed Computing**, Elsevier, v. 154, p. 1–15, 2021.

PENNA, P. H. et al. RMem: An OS Service for Transparent Remote Memory Access in Lightweight Manycores. In: **MultiProg 2019 - 25th International Workshop on Programmability and Architectures for Heterogeneous Multicores**. Valencia, Spain: [s.n.], 2019. (High-Performance and Embedded Architectures and Compilers Workshops (HiPEAC Workshops)), p. 1–16. Disponível em: https://hal.archives-ouvertes.fr/hal-01986366.

PINTO, S. et al. Virtualization on trustzone-enabled microcontrollers? voilà! In: IEEE. **2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)**. [S.l.], 2019. p. 293–304.

ROSSI, D. et al. Energy-efficient near-threshold parallel computing: The pulpv2 cluster. **IEEE Micro**, v. 37, n. 5, p. 20–31, 2017.

SHARMA, P. et al. Containers and virtual machines at scale: A comparative study. In: **Proceedings of the 17th International Middleware Conference**. [S.l.: s.n.], 2016. p. 1–13.

SINGH, G. et al. A predictive checkpoint technique for iterative phase of container migration. **Sustainability**, MDPI, v. 14, n. 11, p. 6538, 2022.

STOYANOV, R.; KOLLINGBAUM, M. J. Efficient live migration of linux containers. In: SPRINGER. **International Conference on High Performance Computing**. [S.l.], 2018. p. 184–193.

SWEENEY, J. Virtualization: An overview. **Encyclopedia of Cloud Computing**, Wiley Online Library, p. 89–101, 2016.

SYNYTSKY, R. **Containers Live Migration: Behind the Scenes**. 2016. Disponível em: https://www.infoq.com/articles/container-live-migration/.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN 013359162X, 9780133591620.

THALHEIM, J. et al. Cntr: Lightweight os containers. In: **2018 USENIX Annual Technical Conference**. [S.l.: s.n.], 2018. p. 199–212.

VANZ, N.; SOUTO, J. V.; CASTRO, M. Virtualização e migração de processos em um sistema operacional distribuído para lightweight manycores. In: SBC. **Anais da XXII Escola Regional de Alto Desempenho da Região Sul**. [S.l.], 2022. p. 45–48.

WANG, Z. et al. Ada-things: An adaptive virtual machine monitoring and migration strategy for internet of things applications. **Journal of Parallel and Distributed Computing**, Elsevier, v. 132, p. 164–176, 2019.

WATADA, J. et al. Emerging trends, techniques and open issues of containerization: a review. **IEEE Access**, IEEE, v. 7, p. 152443–152472, 2019.

WOOD, T. et al. Black-box and gray-box strategies for virtual machine migration. In: **NSDI**. [S.l.: s.n.], 2007. v. 7, p. 17–17.

ZHANG, Q. et al. A comparative study of containers and virtual machines in big data environment. In: IEEE. **2018 IEEE 11th International Conference on Cloud Computing (CLOUD)**. [S.l.], 2018. p. 178–185.