

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
7 de julho de 2023

Nicolas Vanz

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

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 7 de julho de 2023.

Prof^a. Lúcia Helena Martins Pacheco, Dr.
Coordenadora do Curso

Banca Examinadora:

Prof. Márcio Bastos Castro, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Giovani Gracioli
Avaliador
Universidade Federal de Santa Catarina

Prof. Odorico Machado Mendizabal
Avaliador
Universidade Federal de Santa Catarina

Nicolas Vanz

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

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Márcio Bastos Castro, Dr.

Coorientador: João Vicente Souto, Me.

Florianópolis

7 de julho 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 a funcionalidade de 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 baseado em uma abordagem *multikernel* de código aberto projetado para *lightweight manycores*. Os resultados experimentais mostram que a virtualização impactou positivamente o desempenho do SO. Houve aumento de desempenho no subsistema de *threads* e redução de desvios, faltas na *cache* de dados e faltas na *cache* de instruções. Os processos puderam ser transferidos entre *clusters* do processador em um *downtime* que varia entre 19 ms e 101 ms, dependendo da quantidade de recursos utilizados.

Palavras-chave: Lightweight Manycores. Sistemas Operacionais. Migração de Processos. Virtualização. Containerização

ABSTRACT

The lightweight manycore processor class emerged to provide a high degree of parallelism and energy efficiency. However, developing applications for these processors faces various programmability issues stemming from their architectural peculiarities. Particularly, process management needs to mitigate problems arising from small local memories and the lack of robust virtualization support. In this context, this work aims to develop a process migration functionality in a distributed operating system for lightweight manycores through a lightweight container based virtualization approach. Specifically, this work is part of the Nanvix project, which is an open-source distributed operating system based on a multikernel approach designed for lightweight manycores. Experimental results show that virtualization positively impacted the operating system's performance. There was an increase in performance in the thread subsystem and a reduction in branches, in data cache misses and instruction cache misses. The processes were able to be transferred between processor clusters with a down time ranging from 19 ms to 101 ms, depending on the amount of resources used.

Keywords: Lightweight Manycores. Operating Systems. Process Migration. Virtualization. Containerization

LISTA DE FIGURAS

Figura 1 – Visão conceitual de um processador <i>lightweight manycore</i>	16
Figura 2 – Exemplos de arquiteturas <i>Multiple Instruction Multiple Data</i> (MIMD).	20
Figura 3 – Visão arquitetural do processador Kalray MPPA-256.	21
Figura 4 – Estrutura interna da <i>Hardware Abstraction Layer</i> (HAL) do Nanvix.	22
Figura 5 – Estrutura interna do <i>microkernel</i> do Nanvix.	23
Figura 6 – Estrutura interna do <i>multikernel</i> do Nanvix.	23
Figura 7 – Comparação de modelos de ambientes de execução de aplicação.	28
Figura 8 – Fluxo de execução da <i>cold migration</i>	30
Figura 9 – Fluxo de execução da <i>pre-copy migration</i>	31
Figura 10 – Fluxo de execução da <i>post-copy migration</i>	31
Figura 11 – Fluxo de execução da migração híbrida.	32
Figura 12 – Visão simplificada da estrutura do binário gerado na compilação.	43
Figura 13 – Diferença da estrutura do Nanvix com e sem a <i>User Area</i>	44
Figura 14 – Fluxo de <i>tasks</i> de migração	48
Figura 15 – Impactos da virtualização sobre a manipulação de <i>threads</i>	57
Figura 16 – <i>Downtime</i> da aplicação durante o experimento de migração fixando a quantidade de páginas alocadas dinamicamente	58
Figura 17 – <i>Downtime</i> da aplicação durante o experimento de migração fixando a quantidade de <i>threads</i>	59
Figura 18 – <i>Downtime</i> da aplicação em função da quantidade de migrações paralelas	60

LISTA DE ABREVIATURAS E SIGLAS

cgroup	<i>control group.</i>	27, 28, 32, 37, 38
CPU	<i>Central Processing Unit.</i>	25, 27
CRIU	<i>Checkpoint/Restore In Userspace.</i>	32, 36, 46
DARPA/IPTO	Departamento de Defesa do Governo dos Estados Unidos. . .	15
ERAD/RS	Escola Regional de Alto Desempenho da Região Sul.	18
FaaS	<i>Functions as a Service.</i>	36
FLOPS	<i>Floating-point Operations per Second.</i>	15
HAL	<i>Hardware Abstraction Layer.</i>	9, 21, 22, 43
IBM	<i>International Business Machines Corporation.</i>	25
IoT	<i>Internet of Things.</i>	36, 37
IPC	<i>Inter-Process Communication.</i>	27
LXC	<i>Linux Containers.</i>	27, 28, 37, 38
LXD	<i>Linux Containers Daemon.</i>	27
MCU	<i>Microcontroller Unit.</i>	36
MIMD	<i>Multiple Instruction Multiple Data.</i>	9, 15, 19, 20
MISD	<i>Multiple Instruction Single Data.</i>	19
MMU	<i>Memory Management Unit.</i>	21
MPSoC	<i>Multiprocessor System-on-Chip.</i>	15, 20
NoC	<i>Network-on-Chip.</i>	15, 16, 20, 21, 22, 24, 42, 46, 47, 59
PID	<i>Process ID.</i>	27, 32
SIMD	<i>Single Instruction Multiple Data.</i>	19
SISD	<i>Single Instruction Single Data.</i>	19
SO	Sistema Operacional. .	5, 17, 18, 19, 21, 22, 24, 25, 26, 28, 30, 33, 38, 39, 41, 42, 43, 44, 45, 46, 47, 61
SRAM	<i>Static Random Access Memory.</i>	20
TLB	<i>Translation Lookaside Buffer.</i>	21, 42, 50, 51

UArea	<i>User Area.</i>	44, 45, 48, 50, 53, 54, 57
UTS	<i>Unix Timesharing System.</i>	27
VDI	<i>Virtual Desktop Infrastructure.</i>	28
VM	Máquina Virtual.	25, 28, 29, 30, 31, 32, 35, 36, 38, 41
VMM	<i>Virtual Machine Monitor.</i>	25, 26
WAN	<i>Wide-Area Network.</i>	19

SUMÁRIO

1	INTRODUÇÃO	15
1.1	OBJETIVOS	17
1.1.1	Objetivo Principal	18
1.1.2	Objetivos Específicos	18
1.2	CONTRIBUIÇÕES	18
1.3	ORGANIZAÇÃO DO TRABALHO	18
2	REFERENCIAL TEÓRICO	19
2.1	DOS <i>SINGLE-CORES</i> AOS <i>LIGHTWEIGHT MANYCORES</i>	19
2.2	NANVIX OS	21
2.2.1	Abstrações de Comunicação do Nanvix	23
2.2.2	<i>Tasks</i> no Nanvix	24
2.3	VIRTUALIZAÇÃO	24
2.3.1	Virtualização total	25
2.3.2	Para-virtualização	26
2.3.3	Virtualização a Nível de Sistema Operacional e Containerização	26
2.3.4	Outros Tipos de Virtualização	28
2.4	MIGRAÇÃO	29
2.4.1	<i>Cold migration</i>	29
2.4.2	<i>Hot Migration</i>	30
2.4.2.1	<i>Pre-Copy migration</i>	30
2.4.2.2	<i>Post-Copy migration</i>	31
2.4.2.3	<i>Migração Híbrida</i>	31
2.5	<i>CHECKPOINT/RESTORE IN USERSPACE</i>	32
2.6	TÉCNICAS UTILIZADAS NO TRABALHO	33
3	TRABALHOS RELACIONADOS	35
3.1	"VIRTUALIZATION ON TRUSTZONE-ENABLED MICROCONTROLLERS? VOILÀ!"	35
3.2	"CHECKPOINTING AND MIGRATION OF IOT EDGE FUNCTIONS"	36
3.3	"LIGHTWEIGHT VIRTUALIZATION AS ENABLING TECHNOLOGY FOR FUTURE SMART CARS"	37
3.4	"CONTAINER-BASED REAL-TIME SCHEDULING IN THE LINUX KERNEL"	37
3.5	COMPARAÇÃO DO PRESENTE TRABALHO COM OS TRABALHOS RELACIONADOS	38

4	PROPOSTA DE VIRTUALIZAÇÃO E MIGRAÇÃO DE PROCESSOS PARA <i>LIGHTWEIGHT MANYCORES</i>	41
4.1	VIRTUALIZAÇÃO NO NANVIX	41
4.2	CONTEXTO DE UM PROCESSO NO NANVIX	41
4.3	ISOLAMENTO DO CONTEXTO DE UM PROCESSO DE USUÁRIO	42
4.3.1	Divisão de Dados e Instruções	43
4.3.2	<i>User Area</i>	44
4.4	CONTEINERIZAÇÃO	45
4.5	MIGRAÇÃO DE PROCESSOS	45
4.5.1	Rotina de migração	46
4.5.1.1	<i>Daemon de Migração</i>	46
4.5.1.2	<i>Fluxo de Migração</i>	47
4.5.1.3	<i>Interface com o Daemon</i>	51
5	METODOLOGIA DE AVALIAÇÃO	53
6	RESULTADOS EXPERIMENTAIS	57
6.1	ANÁLISE DO IMPACTO DA UAREA E DO ISOLAMENTO DE CÓDIGO E DADOS DE USUÁRIO SOBRE A MANIPULAÇÃO DE <i>TH-READS</i>	57
6.2	ANÁLISE DO <i>DOWNTIME</i> DA MIGRAÇÃO DE ACORDO COM A QUANTIDADE DE DADOS MANIPULADOS	57
6.3	ANÁLISE DO IMPACTO DE MIGRAÇÕES PARALELAS SOBRE O SISTEMA DE COMUNICAÇÃO	59
6.4	ANÁLISE DA CAPACIDADE DO <i>DAEMON</i> REALIZAR MÚLTIPLAS MIGRAÇÕES UTILIZANDO UM MESMO <i>CLUSTER</i>	59
7	CONCLUSÕES	61
7.1	TRABALHOS FUTUROS	61
	REFERÊNCIAS	63
	APÊNDICE A – ARTIGO CIENTÍFICO	67
	APÊNDICE B – CÓDIGO FONTE	73

1 INTRODUÇÃO

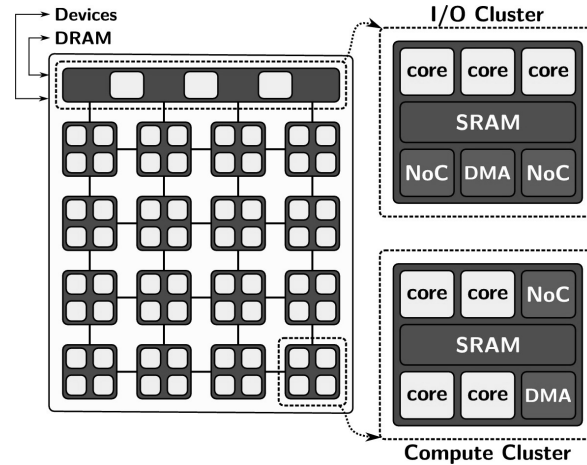
Durante muitos anos, o aumento do desempenho de sistemas computacionais esteve intrinsecamente associado ao aumento da frequência de relógio dos processadores e avanços na tecnologia dos semicondutores. Essas técnicas se mantiveram eficientes até o momento em que a dissipação de calor interna dos *chips* 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-Chips* (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 os *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 explícita 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 aplicações (BARBALACE et al., 2015).

Atualmente, estudos exploram soluções para amenizar o impacto arquitetural 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, nas próximas seções serão expostos o objetivo principal do trabalho e os objetivos específicos.

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 exhibe 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*

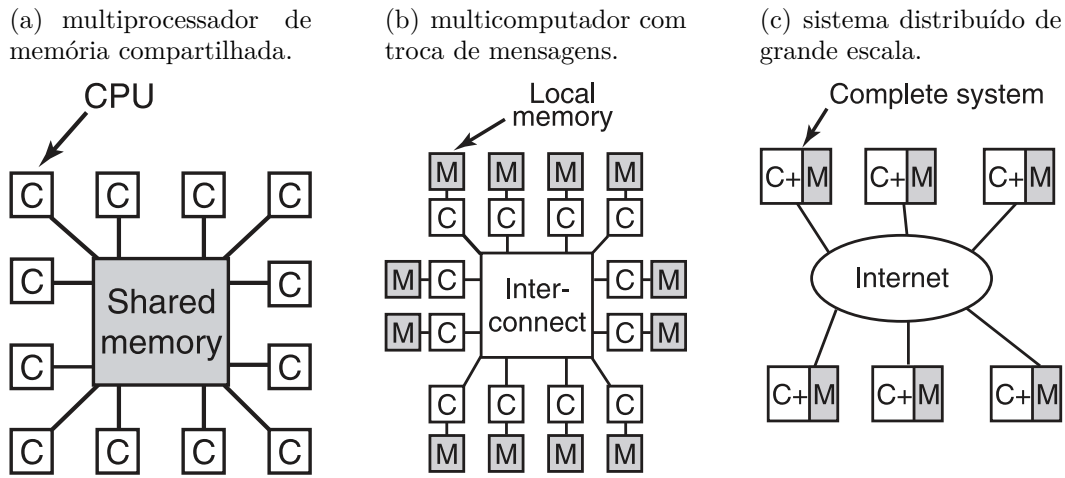
O aumento de desempenho dos sistemas computacionais manteve-se como uma necessidade constante para o avanço da ciência em vários setores: astrologia, biologia, engenharia, etc. Até tempos atrás, esse objetivo era alcançado através do aumento da frequência de relógios do núcleo de processamento, do avanço na tecnologia dos semicondutores e do acréscimo do número de transistores em um *chip* (AMROUCH et al., 2018). Atualmente, chegou-se 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 se consegue colocar em uma mesma área de um *chip* chegou a um limite físico, i.e., o tamanho dos transistores 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.



Fonte: Adaptado de Tanenbaum e Bos (2014)

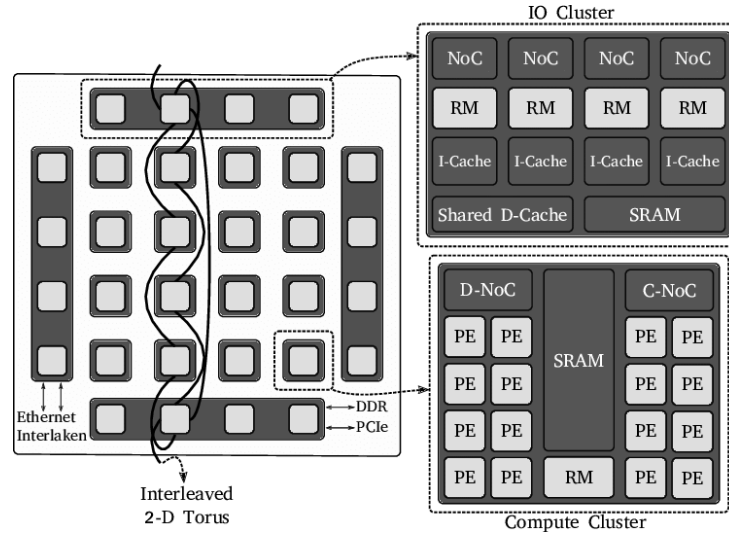
2015). Os *lightweight manycores* são classificados como *Multiprocessor System-on-Chip* (MPSoC) e suas arquiteturas apresentam as seguintes características:

- (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 Nanvix¹ 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). A Figura 4 ilustra a estrutura interna da HAL do Nanvix. Entre os recursos geridos estão: *cores*, *Translation Lookaside Buffers* (TLBs), *cache*, *Memory Management Unit* (MMU), NoC, interrupções, memória virtual e recursos de *I/O*. De maneira geral, esta camada provê abstrações ao nível do *core*, *cluster* e comunicação/sincronização entre *clusters* (PENNA, 2021).

Nanvix *Microkernel* é a camada intermediária que provê gerenciamento de recursos e os serviços mínimos de um SO em um *cluster*. A Figura 5 ilustra a estrutura interna do *microkernel* do Nanvix. Pode-se observar que entre os serviços se encontram o gerenciamento de *threads* e memória, controle de acesso à memória, interface para chamadas de sistema e a comunicação entre processos. As chamadas de sistema podem ser executadas: (i) localmente, caso acessem dados *read-only* ou alterem estruturas internas do *core*, ou (ii) remotamente pelo *master core* (i.e., o *core* exclusivo

¹ Disponível em <https://github.com/nanvix>

Figura 4 – Estrutura interna da HAL do Nanvix.

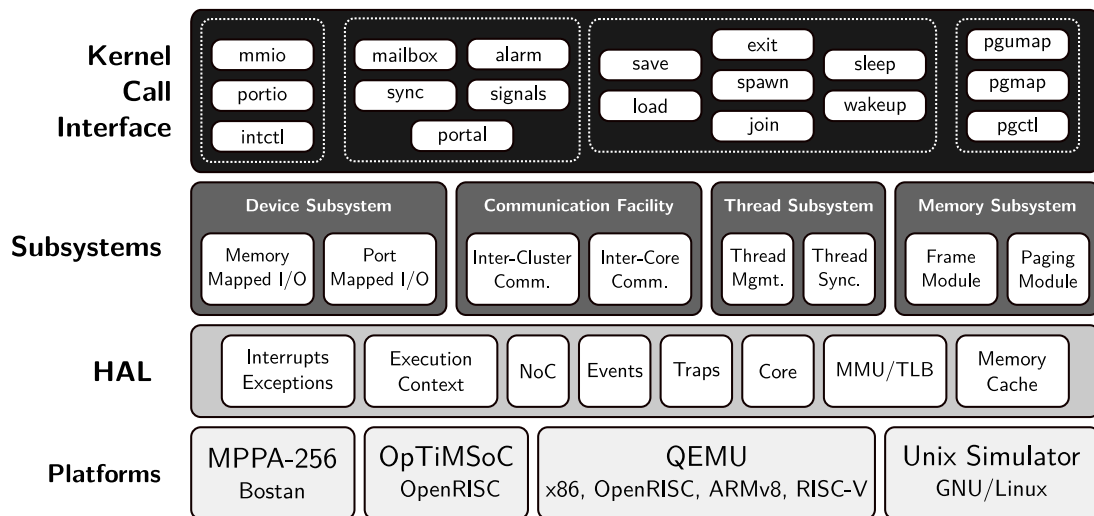


Fonte: Penna (2021)

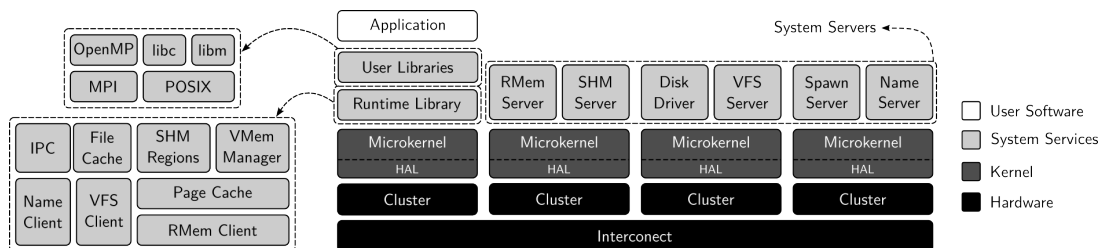
para execução de tarefas do SO), que atende à requisição e libera o *slave core* (i.e., o *core* destinado às *threads* de usuário) requisitante ao término da chamada (PENNA, 2021). Essa característica adjetiva o *microkernel* como assimétrico.

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. A Figura 6 ilustra o *multikernel* do Nanvix. 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. Como pode-se observar na Figura 6, alguns serviços incluídos no Nanvix são mecanismos de *spawn* de processos e gerenciamento de nomes lógicos dos processos a fim de abstrair a localização dos processos no processador.

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

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

Fonte: Penna (2021)

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

Fonte: Penna (2021)

clusters se torna importante, pois a comunicação entre *clusters* próximos é mais rápida e 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) possui 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).

2.2.2 Tasks no Nanvix

No Nanvix há um sistema de *tasks* leves cooperativas (SOUTO, 2022). Nesse sistema, uma *task* é definida como uma sub-rotina que pode ser executada periodicamente e que constitui um passo em um fluxo de execução. As *tasks* podem ser conectadas umas as outras, o que torna possível a criação *loops* ou condicionais, gerando um grafo de execução.

As *tasks* são escalonadas pelo *Dispatcher*, que é executado no *master core*. O *Dispatcher* utiliza uma mesma área de memória i.e., pilha, para a execução das *tasks*. Isso reduz o consumo de memória, já que não é necessário um processo ou *thread* dedicado à *task* (SOUTO, 2022).

Sendo assim, o sistema de *tasks* leves cooperativas se torna uma alternativa boa para o desenvolvimento de protocolos, comunicação, interfaces de *I/O* e *daemons*. Isso porque evita o uso de recursos que são escassos no ambiente em que são utilizadas: *thread* e memória.

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 virtualização 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 Sistema Operacional e Containerização

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 containerizada. 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 containerização 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 containerização 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 personalizadas 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);

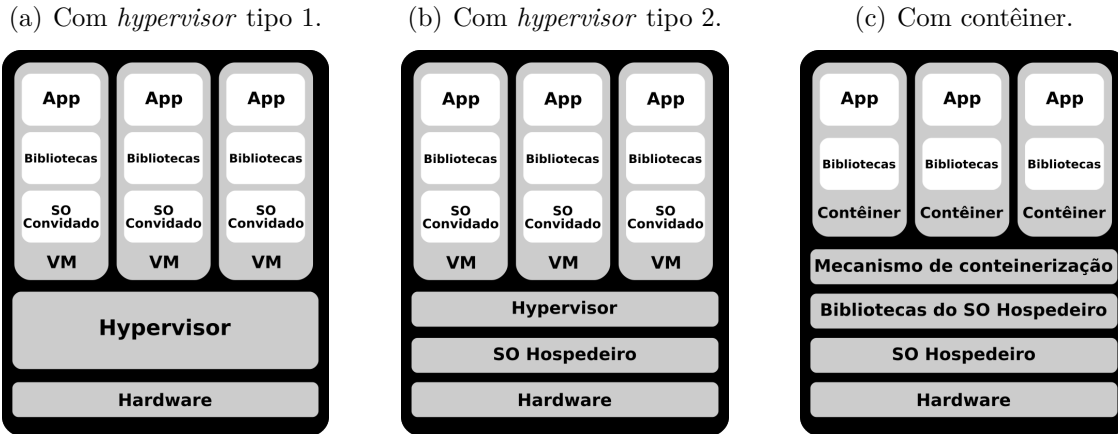
- (ii) *Unix Timesharing System* (UTS): garante que cada contêiner possa ver e alterar seu próprio *hostname*, tornando-o um nó independentemente nomeado na rede (WATADA et al., 2019; DUA; RAJA; KAKADIA, 2014);
- (iii) *Process ID* (PID): virtualiza os identificadores dos processos de forma que cada 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);
- (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 containerização, como o *Docker* e *Linux Containers* (LXC), para garantir que um contêiner não utilize mais recursos do que o necessário.

Várias ferramentas de containerização 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 containerização. 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 portátil para outros sistemas operacionais; há ainda problemas de segurança em aberto; e os contêineres compartilham o mesmo *kernel* (DUA; RAJA; KAKADIA, 2014).

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



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

Outra ferramenta muito conhecida é o *Docker*. O *Docker* traz uma abordagem um pouco diferente do LXC. Enquanto no LXC é possível 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 *Containerd*, 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 ferramentas de containerização 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 uni-

ficados 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

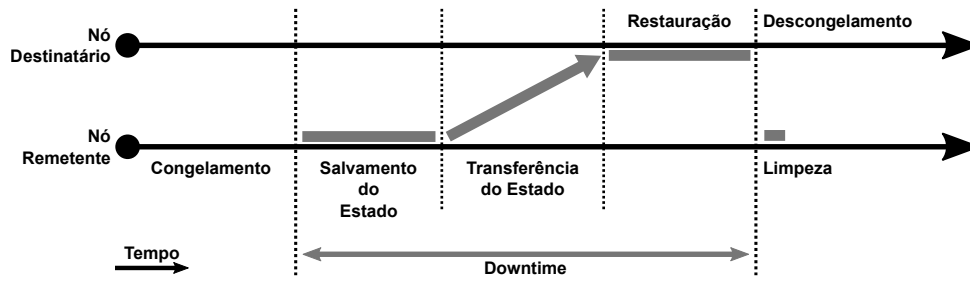
A migração é o processo de transferência de uma aplicação ou VM de um ambiente a outro. A migração é muito usada atualmente principalmente em ambientes *cloud* (IMRAN et al., 2022). A migração traz diversos benefícios e vantagens aos serviços em um ambiente computacional, tais como:

- (i) Balanceamento de carga: é uma técnica que permite que a carga de trabalho de servidores sobrecarregados 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 transferidas 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).

A migração pode ser classificada de acordo com em dois tipos principais. São eles: *cold migration* (*non-live migration*) e *hot migration* (*live migration*) (IMRAN et al., 2022).

2.4.1 Cold migration

A Figura 8 ilustra o fluxo de execução da *cold migration*, também é conhecida como *non-live migration*. Neste tipo de migração, a VM (ou contêiner) a ser migrada precisa ser desligada antes do processo de migração começar. Detalhadamente, o sistema

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

Fonte: Adaptado de Synytsky (2016)

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 downtime*, é alto. O *downtime* 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 aceitável atualmente, haja vista a grande quantidade de serviços que não podem parar sua execução (SINGH et al., 2022; IMRAN et al., 2022).

2.4.2 Hot Migration

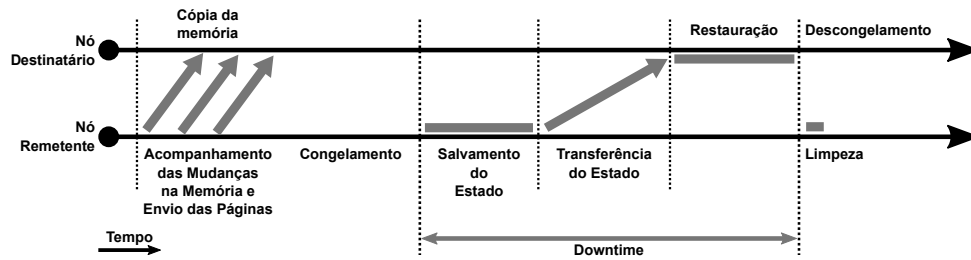
Em contraste com a *cold migration*, na *hot migration*, também conhecida como *live migration*, a VM (ou contêiner) a ser migrada não precisa ser desligada antes do procedimento começar. Os principais objetivos desse processo são: maximizar a performance do sistema durante a migração; melhorar o uso da rede; e reduzir o *downtime* do sistema (IMRAN et al., 2022).

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

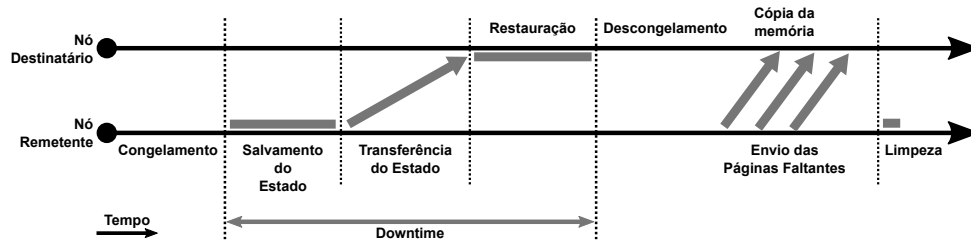
2.4.2.1 Pre-Copy migration

A Figura 9 ilustra o fluxo de execução da *pre-copy migration*. Neste modelo, ao receber a requisição de migração, o remetente cria uma pré-imagem do estado atual da VM (ou contêiner) e a envia ao destinatário enquanto continua executando normalmente. Nesta estrutura está contido o esquema de paginação atual 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 da VM (ou contêiner), que, então, é enviada ao destino, onde é restaurada.

Esse fluxo de migração faz com que o tempo total de migração seja maior que o

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

Fonte: Adaptado de Synytsky (2016)

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

Fonte: Adaptado de Synytsky (2016)

do *cold migration*, já que há a retransmissão de dados, em especial páginas de memória. Em contrapartida, o *downtime* 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).

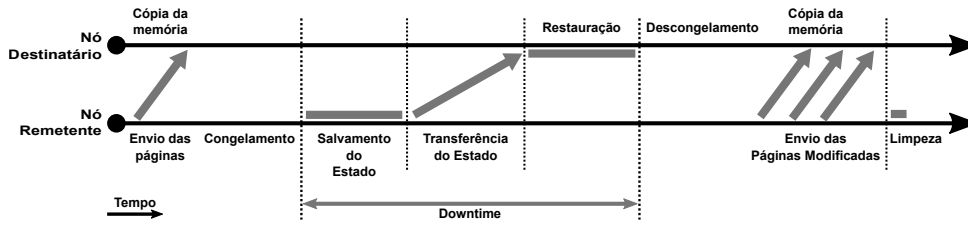
2.4.2.2 *Post-Copy migration*

Tal como ilustrado pela Figura 10, no modelo *post-copy migration* o remetente cria um estado parcial da VM (ou contêiner) 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, a VM (ou contêiner) é apagada do remetente (SINGH et al., 2022; IMRAN et al., 2022).

2.4.2.3 *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 da VM (ou contêiner) 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

Figura 11 – Fluxo de execução da migração híbrida.



Fonte: Adaptado de Synytsky (2016)

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, a VM (ou contêiner) é apagada do remetente (SINGH et al., 2022; IMRAN et al., 2022).

2.5 CHECKPOINT/RESTORE IN USERSPACE

O *Checkpoint/Restore In Userspace* (CRIU) é uma ferramenta de auxílio para a migração de processos entre nós de um sistema computacional. Através do CRIU é possível tirar uma *snapshot* do estado atual de um contêiner, salvar o estado em disco e restaurá-lo novamente (VENKATESH et al., 2019). Mais detalhadamente a migração funciona da seguinte forma:

- (i) A aplicação ou grupo de processos é congelada. O CRIU congela toda a árvore de processos a partir do PID do processo pai da aplicação;
- (ii) O estado da aplicação é salva em um conjunto de arquivos. O CRIU captura as diversas informações do processo. Estão incluídos nessas informações os descritores de arquivos, *pipes* e mapeamento de memória. Nesta etapa, a injeção de um código parasita na aplicação auxilia durante a copia do sistema de memória;
- (iii) Os dados extraídos são copiados para o nó destinatário;
- (iv) A aplicação ou grupo de processos é reconstruída. Nesta etapa, o CRIU se transforma no processo pai da aplicação. A partir dele, toda a árvore de processos é recriada através de *forks*. Neste momento, os processos, *namespaces*, cgroups, mapeamento de memória, *timers* e *threads* são restaurados e são limpos os vestígios do CRIU no processo pai (DASH, 2022).

Todo esse procedimento garante que a migração de um contêiner seja transparente a este e sem perda de contexto. Ou seja, do ponto de vista da aplicação, não é perceptível que ocorreu um congelamento ou migração.

2.6 TÉCNICAS UTILIZADAS NO TRABALHO

O desenvolvimento deste trabalho foi inspirado por duas técnicas citadas acima: a virtualização a nível de SO e a *pre-copy migration*. Ambas as técnicas foram adaptadas ao contexto dos *lightweight manycores* e ao Nanvix.

A virtualização a nível de SO é atrativa para o desenvolvimento deste trabalho pois neste tipo de virtualização não é necessária a presença de um SO convidado, em contraste com a virtualização total e a para-virtualização. Por esse motivo, este modelo não demanda tanto uso de memória. Essa característica atende as exigências dos *lightweight manycores*, nos quais a memória é limitada.

O principal motivo da utilização da *pre-copy migration* neste trabalho é a redução de complexidade do sistema de migração. Os *lightweight manycores* têm pouca memória a sua disposição. Isso significa que as aplicações de usuário não podem ocupar muita memória. Logo, durante a migração de uma aplicação, o tempo dedicado à transferência de memória é menor. Ou seja, considera-se que a transferência de páginas não é tão expressiva no *downtime* a ponto de exigir um sistema mais complexo de migração.

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 *downtime* (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 *downtime* em ambientes *cloud*, há alguns autores preocupados com o desenvolvimento de soluções envolvendo virtualização e migração em ambientes de recursos restritos, como sistemas de tempo real e sistemas críticos, nos quais há restrições de tempo também. 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 possibilidade 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 contêineres 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 promover 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 têm 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 destruição 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 "CONTAINER-BASED REAL-TIME SCHEDULING IN THE LINUX KERNEL"

O artigo "*Container-based real-time scheduling in the linux kernel*" (ABENI; BALSINI; CUCINOTTA, 2019) aborda o tema de virtualização com contêineres em sistemas de tempo real. Os autores exploram a implementação de um escalonador de tarefas em sistemas de tempo real utilizando *Linux Containers* (LXC) (i.e., com o auxílio de *control groups* (cgroups) e *namespaces*).

Abeni, Balsini e Cucinotta (2019) explicam que os componentes que constituem os ambientes de sistemas de tempo real tradicionalmente eram executados por uma VM dedicada, em que as VMs eram gerenciadas por um *hypervisor* escalonador. Os autores afirmam que em algumas situações (e.g., em sistemas embarcados) pode ser vantajosa a utilização de uma virtualização mais leve, baseada em contêineres, já que este tipo de virtualização sobrecarrega menos o sistema quando comparada com a abordagem de virtualização total tradicionalmente utilizada.

Desse modo os autores propõem um escalonador que estende um escalonador já presente no *kernel* do Linux, o `SCHED_DEADLINE`. A implementação consiste em um sistema de escalonamento em uma hierarquia de dois níveis. No primeiro nível, o qual é responsável pelo escalonamento dos *cgroups*, é utilizada a política de escalonamento do `SCHED_DEADLINE` e no segundo nível é utilizada uma política de escalonamento com prioridade fixa, como o `SCHED_FIFO` ou `SCHED_RR`. Resumidamente, no momento em que a política de escalonamento `SCHED_DEADLINE` escalona uma entidade que está associada a uma fila de *tasks* de tempo real, então o escalonador de prioridade fixa seleciona a *task* de maior prioridade.

Os experimentos mostraram que o escalonador proposto fez com que os *cores* fossem utilizados de uma forma melhor, quando compara-se com escalonadores que utilizam a virtualização total. Isso porque, em contraste com a virtualização total, na virtualização com contêineres, os componentes compartilham o mesmo *kernel*. Sendo assim, é possível a migração de *tasks* para filas de *cores* menos utilizados.

3.5 COMPARAÇÃO DO PRESENTE TRABALHO COM OS TRABALHOS RELACIONADOS

O presente trabalho se assemelha com os trabalhos relacionados apresentados no sentido de aplicar a virtualização e migração em um sistema com recursos restritos. De maneira similar aos trabalhos detalhados nas Seções 3.2, 3.3 e 3.4 e em contraste com o trabalho detalhado na Seção 3.1, neste trabalho foi explorado um modelo de virtualização baseado na utilização de contêineres. Isso foi feito com o intuito de evitar a sobrecarga que a virtualização total tem sobre o ambiente, em especial na memória, que nos *lightweight manycores* é escassa.

Neste trabalho, não foi utilizada qualquer ferramenta de auxílio na manipulação de contêineres, em contraste com os trabalhos apresentados, que usufruem de mecanismos de gerenciamento de contêineres, como o Docker (MORABITO et al., 2017; KARHULA; JANAK; SCHULZRINNE, 2019) ou o LXC (ABENI; BALSINI; CUCINOTTA, 2019). Isso porque o trabalho é feito sobre um ambiente não usual, os *lightweight manycores*, utilizando um Sistema Operacional (SO) ainda incomum para a prática dessas técnicas, o Nanvix. Dessa forma, a criação de um contêiner e a forma como este é manipulado foram desenvolvidas desde o início e especificamente para o Nanvix. Isso traz à implementação

algumas características peculiares. Por exemplo, os contêineres executam sobre uma estrutura de *kernel* idêntica, se diferenciando pelo contexto do processo, o que é uma característica comumente vista na virtualização a nível de SO. Contudo, o *kernel* não é compartilhado entre os contêineres, já que em um *cluster* temos exatamente um contêiner e os *clusters* são independentes entre si. Logo, um contêiner tem acesso completo aos recursos do *cluster* em que executa e o *cluster* tem visão apenas do contêiner que hospeda. Então, o gerenciamento desses contêineres precisa ser feito em um nível de abstração maior, em que se tem uma visão do processador como um todo, e não somente de um *cluster*.

Além disso, 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.

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 dependência que um processo tem com o *cluster* em que executa. Em outras palavras, apenas um processo executa por *cluster* e os serviços envolvendo o processo têm dependências em recursos físicos do *cluster*.

Essa falta de mobilidade dos processos 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*.

4.1 VIRTUALIZAÇÃO NO NANVIX

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 em conjunto com 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* e o Nanvix. 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; ZHANG et al., 2018).

4.2 CONTEXTO DE UM PROCESSO NO NANVIX

Para o desenvolvimento da virtualização no Nanvix, é necessário: isolar o que é *kernel* do que é usuário; e diferenciar as partes internas do *kernel* que são contexto do processo das que são contexto do *cluster*. Para isso, é preciso 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 do Nanvix que sustentam um processo são: *threads*, *syscalls*, sistema de memória e comunicação. Todos esses módulos de alguma forma têm dependências no *kernel* ou *hardware* do *cluster* que executa o processo. Essas dependências são ilustradas conceitualmente na Figura 13(a) e algumas delas estão listadas abaixo:

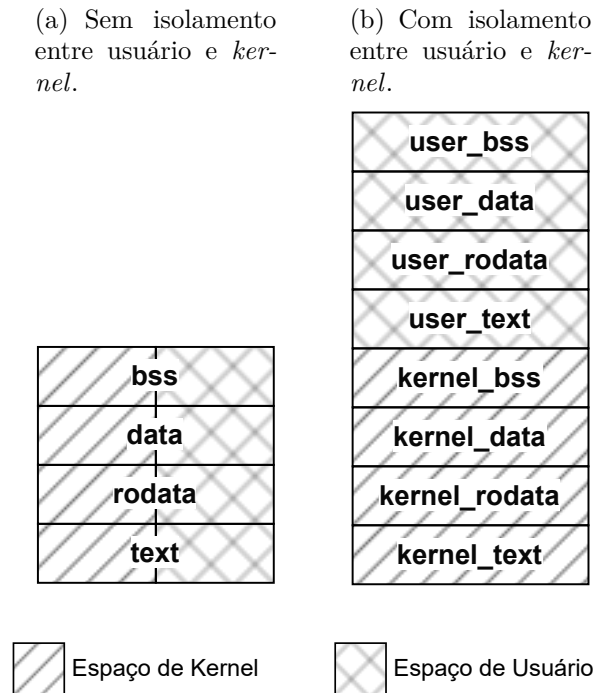
- (i) As estruturas das *threads* de usuário estão armazenadas em listas internas de *kernel*, assim como variáveis de sincronização (para junção de *threads*, por exemplo), estruturas de escalonamento, referências às pilhas de execução e outras variáveis/estrutura de controle. É importante destacar que na abordagem inicial do Nanvix não havia separação explícita nessas estruturas para identificar quais variáveis são relacionadas 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 Buffers* (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 incluídos no contêiner sejam suficientes para o processo executar. Isso inclui todos os dados de usuário, 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 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 containerização, 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 13(a) ilustra como os subsistemas do Nanvix são originalmente estruturados. Não há uma divisão explícita do que são

Figura 12 – Visão simplificada da estrutura do binário gerado na compilação.



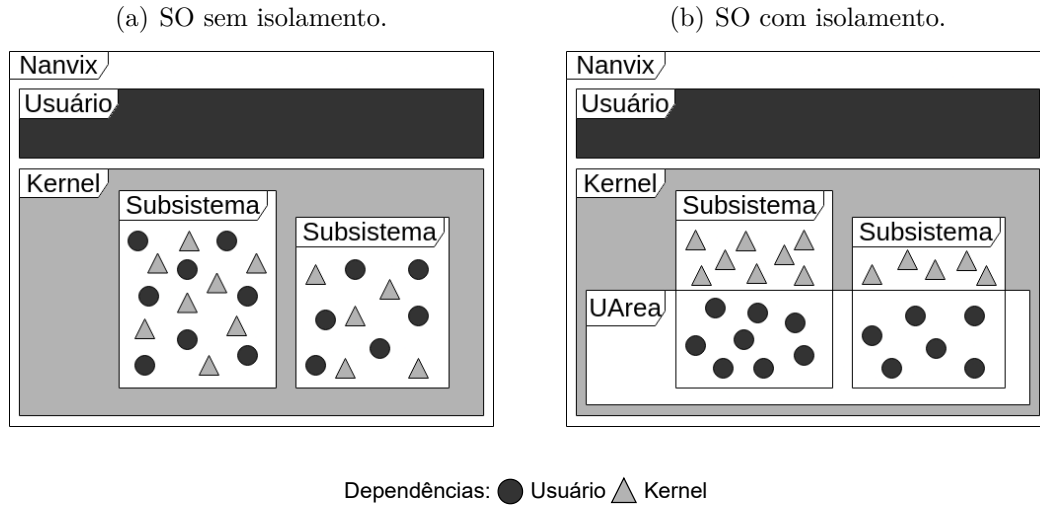
Fonte: Desenvolvido pelo autor.

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. A Figura 12(a) ilustra como o binário era estruturado originalmente e a Figura 12(b), como o é estruturado depois das modificações. 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

Figura 13 – Diferença da estrutura do Nanvix com e sem a *User Area*.

Fonte: Desenvolvido pelo autor.

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* passem a ter a mesma organização interna de *kernel*, haja vista que a ligação é estática. Como consequência disso, não precisamos mover códigos de *kernel*, já que todos os *clusters* possuem *kernels* idênticos. 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 containerização 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

É importante destacar que apesar dessa estrutura armazenar informações de usuário, ela é armazenada em espaço de *kernel*. Além disso, tem tamanho fixo, o que garante a uniformidade do *kernel* entre os *clusters*. Essa estrutura foi projetada para englobar as várias arquiteturas suportadas pelo Nanvix. Adicionalmente, 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 deste trabalho está focada em isolar o processo 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 é definido como a junção do código de usuário, dados de usuário e UArea. Essa união abrange o essencial para a execução do processo: código e dados de usuário; e as dependências internas do processo.

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 factível. Especificamente,

nós eliminamos a necessidade de descobrir quais são e onde estão as informações que compõem o estado de um processo dentro do Nanvix. Todo o estado do processo está agora isolado via containerização, 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 e ligação detalhados na Subseção 4.3.1).

Como consequência da uniformidade do *kernel* entre os *clusters*, eliminamos o envio de dados redundantes durante a migração. Isso porque os dados referentes à instância local do SO não precisam ser enviados, já que o *kernel* é idêntico em todos os *clusters*. Dessa forma, apenas os dados relacionados ao contêiner são migrados/enviados, o que atenua 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 Seção 2.4.2, utilizando a técnica *pre-copy*, vista na Seção 2.4.2.1. Em outras palavras, 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 a entidade responsável por gerenciar as migrações. O *daemon* é inicializado durante o *boot* do sistema e é composto por um fluxo de *tasks* (SOUTO, 2022), 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 e um *Portal default* para recebimento dos dados durante a migração. Em contraste, o *Portal* de envio de dados e as *Mailboxes* para envio das mensagens de migração são criados sob demanda durante a migração e destruídos após o término dela. Isso é feito com o intuito de economizar os recursos de comunicação do sistema.

O fluxo de migração se 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 ativada. 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: (i) perda de retornos de chamadas de sistemas; (ii) perda de sinais de sincronização; (iii) inconsistência de valores de variáveis de usuário; (iv) inconsistência na alocação de memória; 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. Isso é importante pois 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. Portanto, é imprescindível que o SO não seja pausado como a aplicação. Ou seja, é essencial que as *threads* de sistema continuem 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.

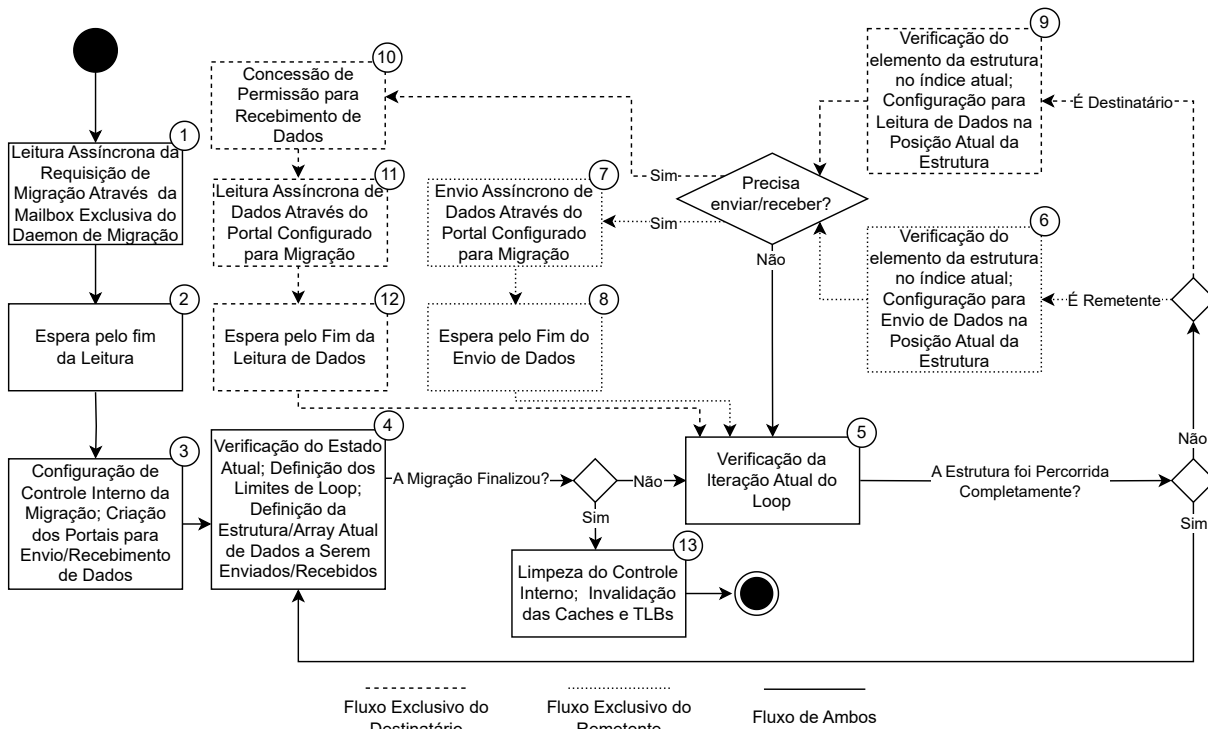
3. Restauração da execução do processo no *cluster* destino.

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 14 ilustra o fluxo de execução da migração. Nela, cada quadro corresponde a uma *task* e a descrição de 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.

Esse *design* de migração baseado em *tasks* foi adotado porque: (i) garante o

Figura 14 – Fluxo de *tasks* de migração



isolamento das etapas de migração em passos bem definidos; (ii) permite a construção de um *design* de *daemon* mais leve, que não consuma muitos recursos; (iii) o funcionamento da comunicação assíncrona está intrinsecamente ligado ao sistema de *tasks* do Nanvix.

Abaixo estão descritos o funcionamento mais detalhado de cada *task* de migração de acordo com o número identificador ilustrado na Figura 14.

1. Esta é a *task* responsável pela leitura assíncrona de uma mensagem 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 independentemente do papel do *cluster* (remetente ou destinatário).
2. Esta *task* é responsável por esperar o término da leitura assíncrona disparada pela *task* 1. Portanto, esta é a *task* que indica ao *daemon* quando uma mensagem é recebida e lida completamente.
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 números 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, durante a execução desta *task* 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. Na Tabela 1 estão os estados e as estruturas que são manejadas em cada um deles. 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 é liberada, atualizando o estado atual. Caso contrário, um novo índice é considerado para o envio/recebimento de dados.

MSTATE_SECTIONS	Seções do binário identificadas como partes do usuário: <code>.user.text</code> , <code>.user.data</code> , <code>.user.bss</code> e <code>.user.rodata</code> .
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 <i>kernel</i> estão sendo usadas.
MSTATE_KSTACKSPHYS	Páginas de <i>kernel</i> que estão sendo usadas.
MSTATE_FRAMES_BITMAP	<i>Bitmap</i> responsável pela identificação de quais <i>frames</i> estão sendo usados.
MSTATE_FRAMES_PHYS	<i>Frames</i> de usuário que estão sendo usados
MSTATE_FINISH	Não manipula estruturas. Apenas reseta o controle interno da migração e invalida TLBs e <i>caches</i> .

Tabela 1 – Estados da migração e estruturas manipuladas em cada um deles.

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 destinatário 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 checagem 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`, a qual é chamada ativamente pela aplicação que requisita a migração. 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 o envio da requisição de migração. Depois disso, o *cluster* requisitante é congelado através da chamada de sistema *freeze*. É importante lembrar que esta *task* recém-criada não é impedida de executar, já que as *tasks* são executadas por uma *thread* de sistema. No momento em que a *task* é escalonada, são construídas e enviadas duas mensagens de migração: uma o para o próprio *cluster* que invocou a função `kmigrate_to` (o remetente) e uma para o *cluster* identificado pelo parâmetro (o destinatário). Essas mensagens são enviadas para as portas *Mailbox* dos *daemons* dos *clusters* envolvidos. No momento em que as mensagens são lidas, o processo de migração inicia.

É interessante destacar que o *daemon* foi projetado possibilitando expansão das suas funcionalidades sem maiores complicações. Por exemplo: (i) Novas interfaces podem ser desenvolvidas, já que o funcionamento do *daemon* é independente; (ii) A estrutura do sistema de migração é de fácil manipulação, permitindo a adição de novas funcionalidades e (iii) A adição/remoção de uma estrutura na migração implica na adição/remoção de um estado em uma estrutura condicional do tipo *switch*.

5 METODOLOGIA DE AVALIAÇÃO

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

- Q1 Qual impacto o isolamento da *User Area* (UArea) e do código e dados de usuário teve sobre o tempo de execução de operações do subsistema de *threads* no Nanvix?
- Q2 Qual a eficiência da migração de processos no Nanvix de acordo com a quantidade de dados manipulados pela aplicação?
- Q3 Há sobrecarga no sistema de comunicação quando migramos aplicações paralelamente?
- Q4 O *daemon*, em um mesmo *cluster*, é capaz de atender requisições de migração variando o papel do *cluster* na migração?

Para responder a pergunta Q1, foi desenvolvido um experimento 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. Este experimento estressa o subsistema de threads através da criação e junção do máximo de *threads* que o sistema suporta i.e., 18 *threads*. Especificamente, coletamos o tempo de execução, desvios e faltas ocorridas na *cache* de dados e de instrução.

Para responder a pergunta Q2, foi desenvolvido outro experimento. O experimento mensura o tempo de transferência de um processo entre *clusters* de acordo com os recursos utilizados i.e., *threads* e quantidade de páginas de memória alocadas dinamicamente. Neste teste, variamos a quantidade de páginas de memória alocadas dinamicamente entre 0 e 32; e *threads* usadas pela aplicação entre 1 e 17. Dessa forma, nós avaliamos como ocorre a progressão do tempo de transferência de um processo desde a alocação mínima de recursos até a alocação máxima i.e., desde 1 *thread* e 0 páginas de memória até 17 *threads* e 32 páginas de memória, em que cada página ocupa 4 KB.

Para responder a pergunta Q3, nós desenvolvemos um experimento que mensura o *downtime* da aplicação quando variamos a quantidade de processos migrados paralelamente. Neste experimento são considerados quatro cenários de migração paralela em que cada processo é migrado uma vez entre um par de *clusters*. Os cenários se diferenciam pela quantidade de processos e *clusters*: 1, 2, 4 e 8 processos, o que abrange, respectivamente, 2, 4, 8 e 16 *Compute Clusters*. Um *I/O Cluster* é utilizado em todos os cenários com o intuito de registrar o tempo obtido.

Para responder a pergunta Q4, desenvolvemos um experimento que avalia a corretude do *daemon* quando realizamos mais de uma migração envolvendo o mesmo *cluster* e variando o papel deste na migração. Neste experimento, um processo é migrado de *cluster* em *cluster* até que percorra todos os *Compute 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. O movimento circular garante que todos os *clusters* façam duas migrações: uma como remetente e outra como destinatário, totalizando 16 migrações.

Todos os testes foram realizados no processador Kalray MPPA-256. Realizamos múltiplas replicações para garantir maior confiança estatística. Foram feitas 100 replicações para o primeiro experimento e 20 replicações para os demais. A medição de tempo no segundo e terceiro experimento foi feita utilizando 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. A soma de todas essas variáveis resulta na quantidade de dados manipulados (*DM*) em *bytes*.

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

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

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

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

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 dinamicamente. 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*. Primeiro, o *KSPHYS* muda quando uma *thread* é criada. Nesta operação, 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*. Segundo, o *FPHYS* muda quando página de usuário é dinamicamente criada. Quando isso acontece, 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:

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

ou simplesmente:

$$DM = 23008 + 4096(2 \times NTHREADS + NPAGES + \min(NPAGES, 1)) \quad (5.4)$$

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

Quanto ao funcionamento dos experimentos, 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 *downtime* da aplicação.

6 RESULTADOS EXPERIMENTAIS

Neste capítulo, serão apresentados os resultados obtidos através dos experimentos desenvolvidos.

6.1 ANÁLISE DO IMPACTO DA UAREA E DO ISOLAMENTO DE CÓDIGO E DADOS DE USUÁRIO SOBRE A MANIPULAÇÃO DE *THREADS*

Ao analisar o isolamento da aplicação através da UArea e separação do binário de usuário e kernel, percebemos um impacto positivo no subsistema de threads do Nanvix. A Figura 15(a) ilustra que o Nanvix obteve um leve ganho de desempenho na operação de criação de *threads* e não introduziu mudanças significativas na operação de junção de *threads*. A Figura 15(b) mostra a origem do aumento da performance na operação de criação de *threads*, exibindo a quantidade de desvios e faltas nas *caches* de dados e instrução. Percebemos que houve uma diminuição de: (i) 7,8 % na quantidade de desvios; (ii) 5,8 % na quantidade de faltas na *cache* de dados; e (iii) 6,45 % na quantidade de faltas na *cache* de instruções. 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.

6.2 ANÁLISE DO *DOWNTIME* DA MIGRAÇÃO DE ACORDO COM A QUANTIDADE DE DADOS MANIPULADOS

A Figura 16 e a Figura 17 mostram a progressão de tempo sobre diferentes perspectivas (fixando-se as páginas e *threads*, respectivamente). Como o esperado, o *downtime*

Figura 15 – Impactos da virtualização sobre a manipulação de *threads*.

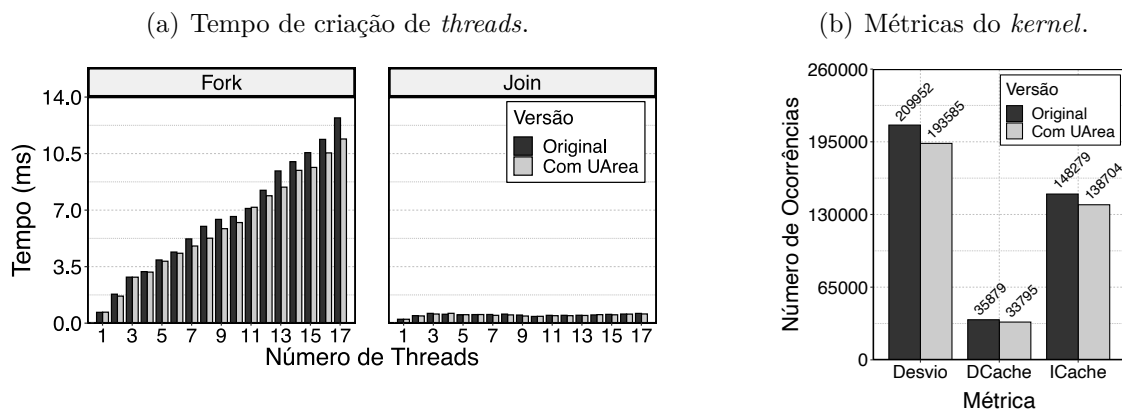
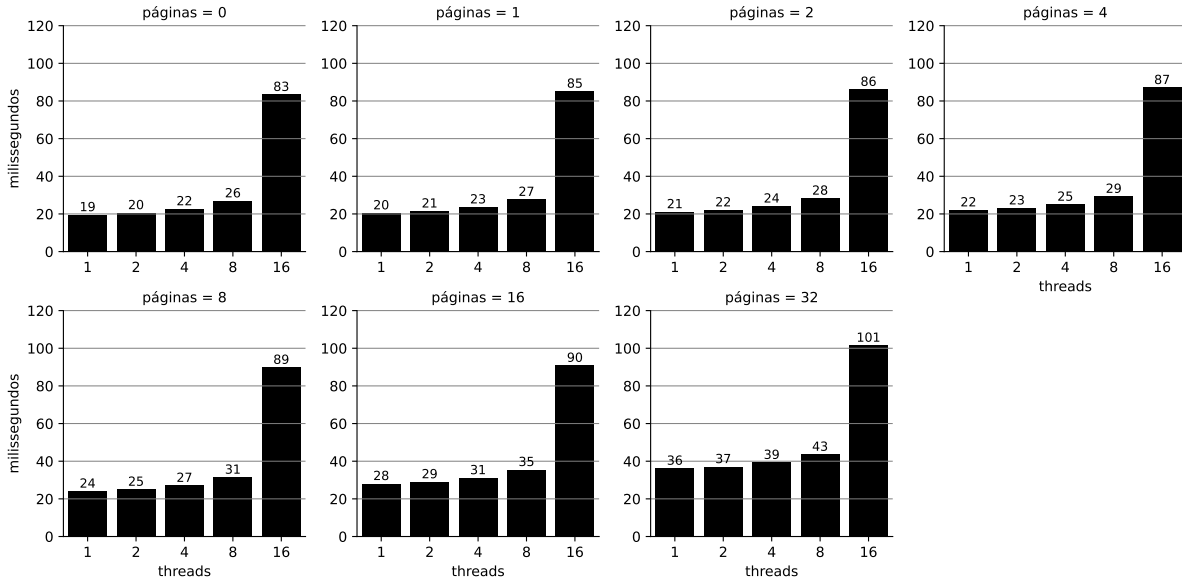


Figura 16 – *Downtime* da aplicação durante o experimento de migração fixando a quantidade de páginas alocadas dinamicamente



Fonte: Desenvolvido pelo autor.

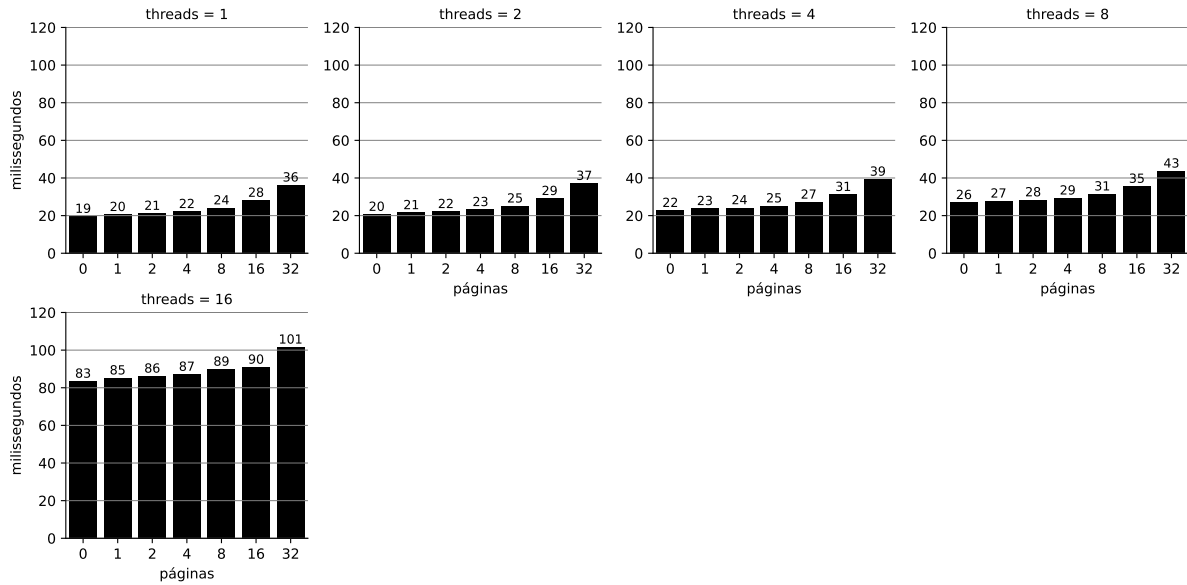
aumenta quanto maior for o número de páginas e *threads*, com um mínimo de 5,26 % e uma máximo de 431 %. Ou seja, quanto maior for a memória utilizada, maior o tempo de comunicação entre os *clusters*. Além disso, a quantidade de *threads* se destaca por apresentar maior expressividade no tempo contabilizado em comparação com a quantidade de páginas. Isso acontece porque uma *thread* requer mais dados migrados além de uma página e.g., é preciso migrar duas pilhas de execução, que totalizam duas páginas de memória e as estruturas de manipulação dessa *thread*.

De maneira geral, o *downtime* é aproximadamente descrito pela Função 6.1:

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

A Função 6.1 aproxima a progressão de tempo até 15 *threads*. Contudo, como podemos observar nas Figuras 16 e 17, ocorre uma disparidade com 16 *threads*, afetando a natureza linear observada até 15 *threads*. Isso acontece porque o Kalray MPPA-256 possui 16 núcleos em um *cluster* e, como o Nanvix apresenta um *microkernel* assimétrico, um núcleo é reservado exclusivamente para a execução de *threads* e tarefas do sistema. Desta forma, ao criar 16 *threads*, nós extrapolamos a quantidade de núcleos disponíveis ao usuário. A competição de recursos entre duas *threads* introduz, em média, 230 % de sobrecarga ao *downtime*, variando entre 198 % e 246 %. Ou seja, o salto numérico observado nas Figuras 16 e 17 é resultado do tempo extra utilizado no experimento para o gerenciamento das *threads*, já que temos mais *threads* do que núcleos em que estas podem executar.

Figura 17 – *Downtime* da aplicação durante o experimento de migração fixando a quantidade de *threads*



Fonte: Desenvolvido pelo autor.

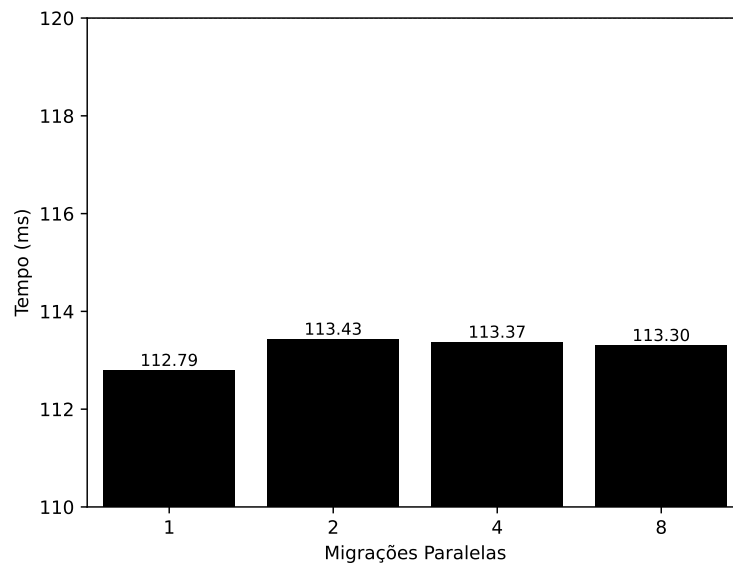
6.3 ANÁLISE DO IMPACTO DE MIGRAÇÕES PARALELAS SOBRE O SISTEMA DE COMUNICAÇÃO

Como ilustrado pela Figura 18, o experimento que mensurava o *downtime* da aplicação quando múltiplas migrações aconteciam simultaneamente, mostrou que a quantidade de migrações paralelas não impacta significativamente o tempo. Destaca-se 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 *downtime*, o que significa que a topologia e a taxa de transferência da *Network-on-Chip* (NoC) do Kalray MPPA-256 supre a demanda de transferência de dados. Considerando os cenários em que há 1, 2, 4 ou 8 migrações simultâneas, o *downtime* médio foi em torno de 113 ms.

6.4 ANÁLISE DA CAPACIDADE DO DAEMON REALIZAR MÚLTIPLAS MIGRAÇÕES UTILIZANDO UM MESMO CLUSTER

O teste que visava a comprovação da consistência e corretude do *daemon* executou completamente sem ocorrência de erros. Isso significa que: (i) o *daemon* é capaz de ser reusado no mesmo *cluster* diversas vezes independentemente do papel do *cluster* na migração; e (ii) é possível um processo ser migrado diversas vezes para diferentes *clusters*, até mesmo em movimento circular envolvendo todos os *clusters*.

Esse comportamento é possível porque o *daemon* é totalmente separado da aplicação e está associado ao *kernel*. Isso porque está contido nas seções de *kernel* do binário

Figura 18 – *Downtime* da aplicação em função da quantidade de migrações paralelas

Fonte: Desenvolvido pelo autor.

e sua execução é sustentada por *tasks*, que são executadas por uma *thread* de sistema, e não de usuário. Dessa forma, é possível que o *daemon* conclua a migração como receptor e, logo em seguida, a partir da requisição da aplicação que acordou, volte a migrar.

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 um Sistema Operacional (SO) distribuído, o Nanvix. Todas as mudanças realizadas neste trabalho são de código aberto e estão disponíveis no repositório oficial do Nanvix¹.

Os resultados mostraram que a virtualização nesses ambientes é possível, bem como a migração de processos entre os *clusters* do processador. O sistema de comunicação não é um gargalo para as migrações, sendo possível realizar múltiplas migrações simultaneamente. Neste contexto, a containerização exerceu o papel principal ao evitar o envio de dados redundantes relativos ao *kernel* e melhor organizar os dados internos do *kernel* e do usuário.

A migração provocou um *downtime* que varia entre 19 ms e 101 ms, dependendo da quantidade de recursos utilizados. O isolamento das dependências de um processo não impactou negativamente o sistema. Pelo contrário, a containerização aumentou o desempenho da operação de criação de *threads*. Além disso, aumentou o desempenho do *kernel* na execução normal do SO, diminuindo a quantidade de desvios, faltas na *cache* de dados e faltas na *cache* de instruções em 7,8 %, 5,8 % e 6,45 %, respectivamente.

7.1 TRABALHOS FUTUROS

A containerização desenvolvida neste trabalho engloba parcialmente o contexto que um processo pode ter no Nanvix. Estão incluídos no contêiner informações relacionadas ao gerenciamento de *threads*, gerenciamento de memória e gerenciamento de chamadas de sistema. Naturalmente, a expansão da containerização com o intuito de abranger todos os subsistemas do Nanvix é um possível trabalho futuro. Nesta expansão, pode-se considerar a inclusão do sistema de comunicação e possivelmente de *tasks*.

Além disso, as chamadas de sistema *freeze* e *unfreeze* podem ser utilizadas com um propósito diferente do proposto por este trabalho. Essas funcionalidades foram criadas para habilitar ou desabilitar a execução de *threads* de usuário em um *cluster* a fim de tornar um processo apto para a migração. Contudo, essas chamadas de sistema podem ser utilizadas com outra finalidade. Através do *freeze* e *unfreeze*, pode-se pausar um processo e salvar em disco uma *snapshot*, que representa o estado do processo em dado momento. Com isso, é viável, por exemplo, criar um sistema de *checkpointing* objetivando tornar as aplicações tolerantes a falhas, sendo possível recarregar um estado antigo do processo.

¹ Disponível em <https://github.com/nanvix>

Adicionalmente, uma aplicação direta da migração de processos é o rearranjo dos processos no processador de modo a organizar os processos nos *clusters* de maneira ótima. Sendo assim, evidencia-se o desenvolvimento de um escalonador de processos no Nanvix que considere a melhor distribuição de processos entre os *clusters*. Além disso, destaca-se como trabalho futuro a habilitação da execução simultânea de múltiplas aplicações no processador e da sua proteção.

REFERÊNCIAS

- ABENI, L.; BALSINI, A.; CUCINOTTA, T. Container-based real-time scheduling in the linux kernel. **ACM SIGBED Review**, ACM New York, NY, USA, v. 16, n. 3, p. 33–38, 2019.
- 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/>.
- DASH, A. K. Understanding Migration Mechanisms of Containers using CRIU. 2022.
- 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.

SOUTO, J. V. **A task-based execution engine for distributed operating systems tailored to lightweight manycores with limited on-chip memory**. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 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.

VENKATESH, R. S. et al. Fast in-memory CRIU for docker containers. In: **Proceedings of the International Symposium on Memory Systems**. [S.l.: s.n.], 2019. p. 53–65.

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.

APÊNDICE A – ARTIGO CIENTÍFICO

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

Nicolas Vanz¹, João Vicente Souto¹, Márcio Castro¹

¹Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD)
Universidade Federal de Santa Catarina (UFSC) - Florianópolis/SC

nicolas.vanz@grad.ufsc.br, joao.vicente.souto@posgrad.ufsc.br,
marcio.castro@ufsc.br

Resumo. *Lightweight manycores apresentam memória interna limitada e falta de um suporte robusto para virtualização, dificultando o gerenciamento de processos nessas arquiteturas. Neste contexto, este trabalho explora um suporte inicial a uma virtualização leve baseada em contêineres para um sistema operacional distribuído. Os resultados mostram que o método proposto provê uma melhor localidade dos dados e contribui para a migração de processos.*

1. Introdução

Atualmente, a eficiência energética de sistemas computacionais revela-se tão importante quanto seu desempenho. Processadores *lightweight manycores* surgem para aliar alto desempenho à eficiência energética. Contudo, as escolhas arquiteturais necessárias para atingir esse objetivo dificultam o desenvolvimento de aplicações para essa classe de processadores [Castro et al. 2016].

O Kalray MPPA-256 é um exemplo comercial de *lightweight manycore* e exemplifica as características dessa classe de processadores. A Figura 1 apresenta uma visão geral do Kalray MPPA-256 e suas peculiaridades, tais como: (i) integrar 288 núcleos de baixa frequência em um único chip; (ii) organizar os núcleos em 20 conjuntos (*clusters*) para compartilhamento de recursos locais; (iii) utilizar 2 *Networks-on-Chip* (NoCs) para transferência de dados entre *clusters*; (iv) possuir um sistema de memória distribuída composto por pequenas memórias locais, e.g., *Static Random Access Memory* (SRAM) de 2 MB; (v) não dispõe de coerência de *cache*; e (vi) apresentar componentes heterogêneos, e.g., *clusters* destinados à computação ou comunicação com periféricos.

Tais características, especialmente relacionadas à memória, inviabilizam um suporte complexo para virtualização. Por exemplo, máquinas virtuais utilizadas em ambientes *cloud* possuem à disposição centenas de GBs para isolar duplicatas inteiras de Sistemas Operacionais (SOs) com a ajuda de virtualização no nível de instrução [Sharma et al. 2016]. As pequenas memórias locais e a simplificação do *hardware* para redução do consumo energético restringem os tipos de virtualização suportados. Neste contexto, o presente trabalho explora um modelo de virtualização baseado em contêineres adaptado para *lightweight manycores*. Contêineres são executados pelo SO como aplicações virtuais e não incluem um SO convidado, resultando em um menor impacto no sistema de memória e requerendo menor complexidade do *hardware* [Thalheim et al. 2018, Sharma et al. 2016].

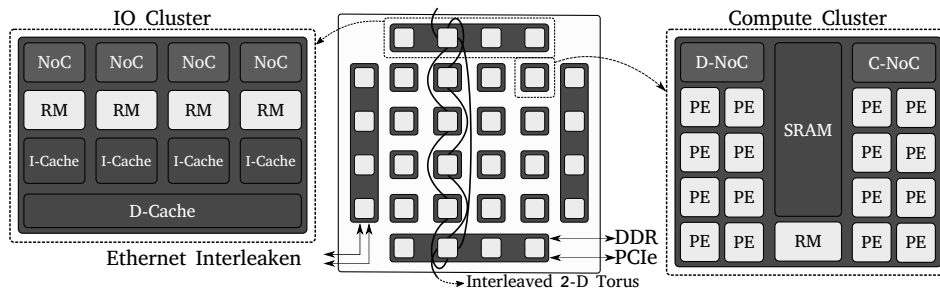


Figura 1. Visão arquitetural do processador Kalray MPPA-256 [Penna et al. 2019].

2. Sistema Operacional Nanvix

O Nanvix¹ é um SO distribuído e de propósito geral que busca equilibrar desempenho, portabilidade e programabilidade para *lightweight manycores* [Penna et al. 2019]. O Nanvix é estruturado em três camadas de abstração. Do nível de abstração mais baixo ao mais alto, são elas: (i) *Hardware Abstraction Layer (HAL)* abstrai os recursos de *hardware* sobre uma visão comum. (ii) *Microkernel* gerencia os recursos de um *cluster* e fornece serviços básicos do SO, e.g., *threads*. (iii) *Multikernel* provê serviços complexos do SO de uma forma distribuída através de um modelo cliente-servidor, e.g., memória distribuída.

Em sua abordagem original, os processos no Nanvix são isolados e estaticamente alocados a um único *cluster*. Desta forma, todas as informações do processo são dependentes do *cluster* que o executa. Por exemplo, comunicações são atreladas aos *clusters* envolvidos e não aos processos. A falta de isolamento e virtualização dos recursos dos processos impõe restrições à utilização do processador, por exemplo, afetando o suporte a multi-aplicação, algo necessário a um SO de propósito geral. Melhorar a mobilidade e a disposição dos processos no processador possibilitaria a utilização mais inteligente dos recursos do mesmo. Por exemplo, reduzir o consumo energético aproximando processos que se comunicam intensamente.

3. Virtualização e Migração de Processos

Visando aumentar a independência dos processos no processador, este trabalho explora um modelo de virtualização leve baseada em contêineres. Nesse modelo, o processo é definido pelos recursos lógicos que compõem o contêiner e não mais pelos recursos físicos e.g., interfaces NoC dos *clusters*. Este trabalho é o resultado de um suporte inicial a esse modelo e engloba apenas o subsistema de *threads*.

Neste contexto, 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 2a ilustra como os subsistemas do Nanvix são 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.

Para tornar a manipulação de processos mais eficiente, introduzimos conceitos de containerização, isolando as dependências que o usuário possui dentro do *cluster* (dados que são gerenciados pelo *kernel* mas pertencem ao contexto do processo de usuário). A

¹Disponível em <https://github.com/nanvix>

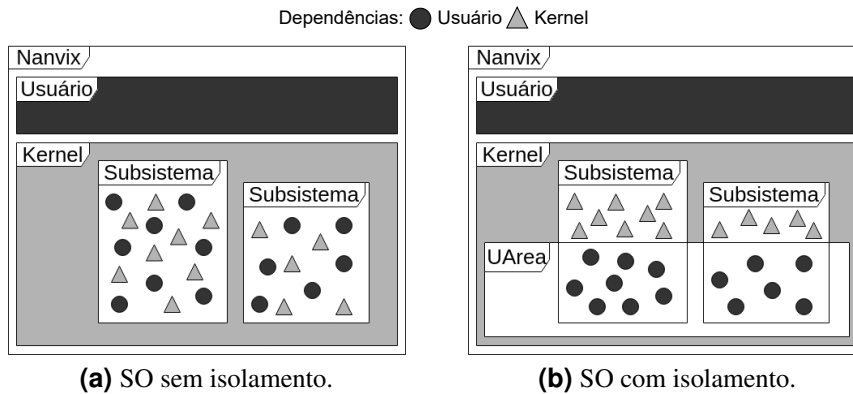


Figura 2. Diferença da estrutura do Nanvix com e sem a User Area.

distinção entre usuário e SO ocorre pela separação das instruções e dados de cada um em segmentos de memória diferentes. Além disso, isolamos as dependências internas do processo em uma área de memória bem definida, separada das demais estruturas do *kernel*, denominada *User Area (UArea)*. A Figura 2b ilustra conceitualmente a divisão das dependências fornecida pela UArea.

Especificamente, a UArea mantém informações sobre (i) *threads* ativas (identificadores e contextos); (ii) ponteiros para suas pilhas de execução; e (iii) variáveis de controle e filas de escalonamento. Inicialmente, a UArea em conjunto com o segmento de dados e instruções do usuário compõem o estado atual do usuário, encapsulando a execução de um processo dentro do *cluster*. Futuramente, introduziremos o módulo de memória e comunicação junto de um processo monitor para manter as conexões consistentes.

Como aplicação direta do isolamento do processo, a migração de processos torna-se mais eficiente. Com a criação de uma instância isolada do espaço do usuário via containerização, eliminamos a necessidade de descobrir quais são e onde estão as dependências do processo, facilitando a transferência de seu contexto. Isso só é possível porque os *clusters* possuem uma estrutura de *kernel* idêntica, descartando a necessidade do envio de dados relacionados ao SO. Ao evitar o envio de dados redundantes entre *clusters*, atenuamos o impacto da migração sobre a NoC.

A funcionalidade inicial de migração é similar ao *Checkpoint/Restore In Userspace (CRIU)*, ferramenta utilizada por *softwares* de gerenciamento de contêineres como o Docker. Porém, a migração será executada por intermédio *daemons* do SO. A migração acontece da seguinte maneira: (i) a execução do processo em um *cluster* é congelada em um estado consistente; (ii) o contexto do processo é enviado via NoC para outro *cluster*. (inclui-se nessa etapa o envio dos segmentos de dados e código do usuário, da UArea e das pilhas de execução); e (iii) a execução do processo é restaurada em outro *cluster*. Trata-se de uma *hot migration*, em que a aplicação é migrada durante sua execução, com cópia das páginas de memória da aplicação.

4. Resultados

Para avaliar o impacto das mudanças feitas para a virtualização, foram desenvolvidos experimentos sobre a manipulação de *threads* e suporte à migração de processos no Nanvix. Todos os experimentos foram executados no processador Kalray MPPA-256 e os resultados mostrados são médias de 100 replicações de cada experimento para garantir 95% de confiança estatística, resultando em um desvio padrão máximo inferior a 1%.

O experimento de manipulação de *threads* mensura os impactos na criação e

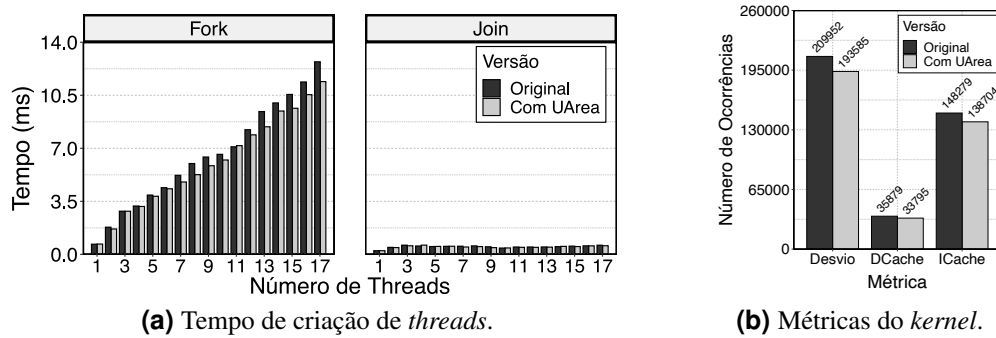


Figura 3. Impactos da virtualização sobre a manipulação de threads.

junção através de diferentes perspectivas. Especificamente, coletamos o tempo de execução, desvios e faltas ocorridas na *cache* de dados e de instrução (Figura 3). Os resultados apresentam um aumento no desempenho das operações de manipulação quando utilizamos a UArea porque exploramos melhor a localidade espacial dos dados, consequentemente, diminuindo o número de faltas na *cache*.

O experimento de migração avaliou o tempo de transferência de um processo entre *clusters*. A aplicação de usuário migrada contém 352,8 KB. Detalhadamente, foram transferidos instruções e dados (342,8 KB), a UArea (2 KB) e uma pilha de execução (8 KB). O tempo médio para migração da aplicação foi de 226 ms.

5. Conclusão

Neste trabalho foi explorado um modelo de virtualização leve baseada em contêineres que considera as restrições arquiteturais de *lightweight manycores* para melhorar o suporte de processos em um SO distribuído. Os resultados mostraram que o isolamento das dependências de um processo aumentaram o desempenho de operações do *kernel* e suportaram a migração de processos de forma eficiente. Como trabalhos futuros, pretende-se (i) ampliar a virtualização, englobando outros subsistemas do Nanvix; (ii) habilitar a execução simultânea de múltiplas aplicações no processador e sua proteção.

Referências

- Castro, M., Franceschini, E., Dupros, F., Aochi, H., Navaux, P. O., and Méhaut, J.-F. (2016). Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54:108–120.
- Choudhary, A., Govil, M. C., Singh, G., Awasthi, L. K., Pilli, E. S., and Kapil, D. (2017). A critical survey of live virtual machine migration techniques. *Journal of Cloud Computing*, 6(1):1–41.
- Penna, P. H., Souto, J., Lima, D. F., Castro, M., Broquedis, F., Freitas, H., and Mehaut, J.-F. (2019). On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In *SBESC 2019 - IX Brazilian Symposium on Computing Systems Engineering*, Natal, Brazil.
- Sharma, P., Chaufournier, L., Shenoy, P., and Tay, Y. (2016). Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, pages 1–13.
- Thalheim, J., Bhatotia, P., Fonseca, P., and Kasikci, B. (2018). Cntr: Lightweight os containers. In *2018 USENIX Annual Technical Conference*, pages 199–212.

APÊNDICE B – CÓDIGO FONTE

Tudo desenvolvido neste trabalho é de código aberto e pode ser acessado no repositório oficial do Nanvix: <https://github.com/nanvix>.