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
4 de abril de 2023

RESUMO

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

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

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

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

Como alternativa à limitação do aumento da frequência de relógio, foram desenvolvidos processadores com vários núcleos de processamento, os *multicores*. O desempenho dos processadores *multicore* não dependem mais apenas das altas frequências de relógio, recorrendo ao paralelismo como principal vantagem aos processadores *single-core*. Deste modo, mesmo com a estagnação da frequência de relógio nos processadores, esse aumento na quantidade de *cores* em conjunto com outras melhorias no *hardware*, 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 dos processadores chamada *lightweight manycore*. Esses processadores são classificados como *Multiprocessor System-on-Chip*s (MPSoCs) e têm como objetivo atrelar alto desempenho à eficiência energética (FRANCESQUINI et al., 2015). Para atingir esse objetivo, a arquitetura dos *lightweight manycores* é caracterizada por:

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

A Figura 1 ilustra uma visão conceitual da arquitetura de um *lightweight manycore*. Os

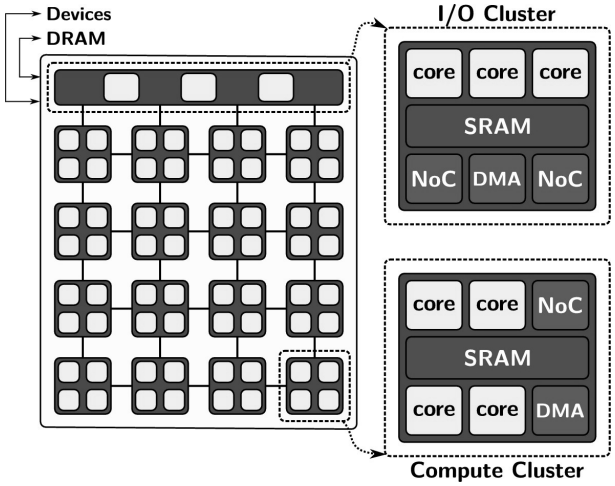


Figura 1 – Visão conceitual de um processador *lightweight manycore* (PENNA et al., 2021)

processadores Kalray MPPA-256 (DINECHIN et al., 2013), PULP (ROSSI et al., 2017) e Sunway SW26010 (FU et al., 2016) são exemplos comerciais dessa classe de processadores.

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

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

Atualmente, estudos exploram soluções para amenizar o impacto das arquiteturais sobre o desenvolvimento de *software*. Sistemas Operacionais (SOs) distribuídos sobresaem-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* destaca-se

por aderir a natureza distribuída e restritiva dos *lightweight manycores* (PENNA et al., 2017; PENNA et al., 2017; PENNA et al., 2019).

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

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

1.1 OBJETIVOS

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

1.1.1 Objetivo Principal

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

1.1.2 Objetivos Específicos

(i) Propor um modelo de virtualização adaptado às necessidades e restrições dos *lightweight manycores*;
(ii) Implementar o modelo proposto no Nanvix, um SO distribuído para *lightweight manycores*;
(iii) Analisar a corretude da solução através do desenvolvimento de *benchmarks* que avaliem a migração de processos;
(iv) Analisar o impacto do modelo de virtualização no Nanvix;
(v) Propor, implementar e avaliar um modelo de migração de processos para o Nanvix.

1.2 CONTRIBUIÇÕES

Esse trabalho de conclusão propõe o suporte à virtualização e migração de processos no Nanvix. Parte desse 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 do trabalho estão organizadas da seguinte maneira. O Capítulo 2 apresenta conceitos fundameitais para o entendimento do trabalho, tais como um detalhamento da arquitetura dos *lightweight manycores* e do Nanvix. O Capítulo 3 discute os trabalhos relacionados. O Capítulo 4 expõe a proposta deste trabalho de conclusão e os detalhes de desenvolvimento da solução a ser implementada. O Capítulo 6 exibe e discute alguns resultados preliminares já obtidos. 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 Operational (SO) 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*. Atualmente, nós estamos chegando ao limite físico que impede a aplicação de parte dessas técnicas. Além da dificuldade de garantir a dissipação de calor à medida que a frequência aumenta, o número de transistores que conseguimos colocar em uma mesma área de um *chip* está chegando ao seu limite físico, i.e., o tamanho dos transitores alcançou a escala atômica.

Como alternativa para a continuidade nos avanços de poder computacional, foram exploradas novas técnicas. Em especial, foram desenvolvidas arquiteturas paralelas, que exploram o poder de processamento paralelo, o qual é atingido pela execução de múltiplos *cores* simultaneamente. 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, as quais ainda podem ser divididas em multiprocessadores ou multicomputadores, como mostrado na Figura 2 (TANENBAUM; BOS, 2014).

Neste contexto, a classe de processadores *lightweight manycores* destacam-se por atrelar alto poder de processamento paralelo com eficiência energética. 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;

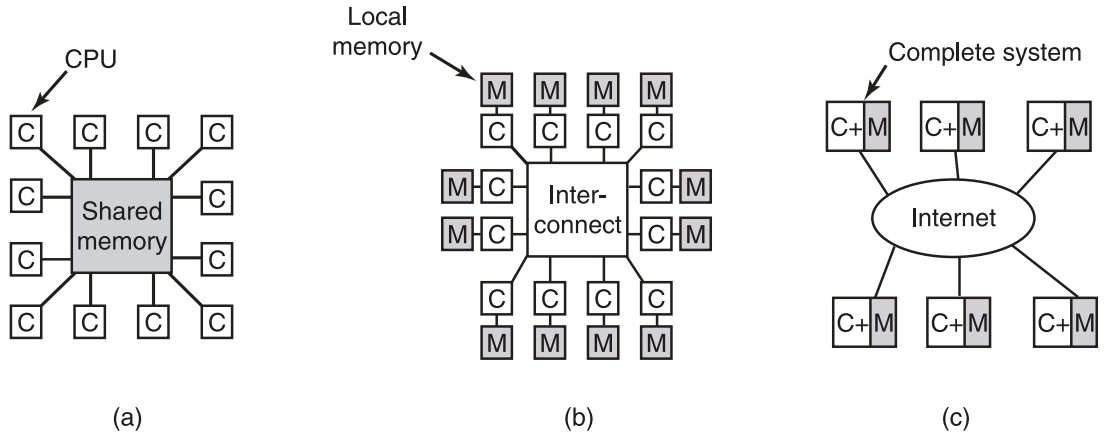


Figura 2 – (a) um multiprocessador de memória compartilhada. (b) um multicomputator com troca de mensagens. (c) um sistema distribuído de grande escala (TANENBAUM; BOS, 2014).

(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, para o desenvolvimento deste trabalho foi utilizado o processador Kalray MPPA-256. A Figura 3 apresenta uma visão geral do processador 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;
(v) Não dispor de coerência de *cache*;
(vi) Apresentar heterogeneidade: *clusters* destinados à computação (*Compute Clusters*) e *clusters* destinados à comunicação com periféricos (*I/O Clusters*).

2.2 NANVIX OS

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

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

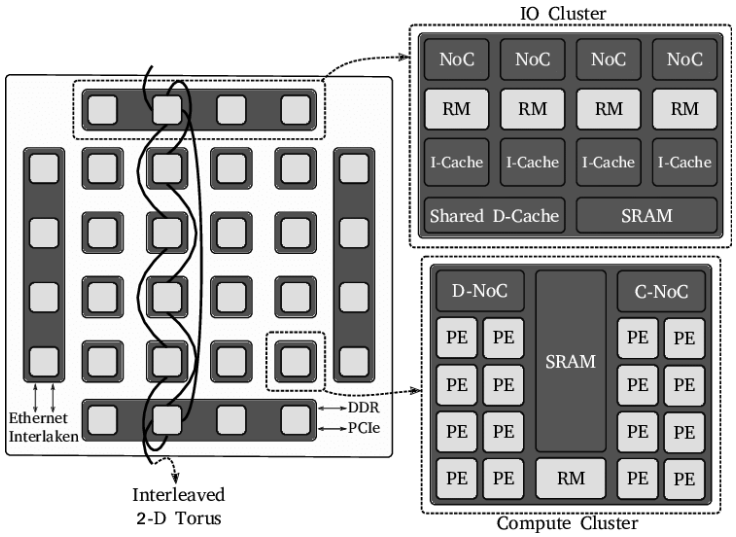


Figura 3 – Visão arquitetural do processador Kalray MPPA-256 (PENNA et al., 2019).

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

Nanvix *Microkernel* é a camada intermediária que provê gerenciamento de recursos e os serviços mínimos de um SO em um *cluster*. Entre esses serviços se encontram a comunição entre processos, gerenciamento de *threads* e memória, controle de acesso à memória e interface para chamadas de sistema. As chamadas de sistema podem ser executadas localmente, caso acessem dados *read-only* ou alterem estruturas internas do *core*, ou remotamente pelo *master core*, que atende à requisição e libera o *slave core* requisitante ao término da chamada (PENNA, 2021). Essa característica adjetiva o *microkernel* como assimétrico. A Figura 5 ilustra a estrutura interna do *microkernel* do Nanvix.

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

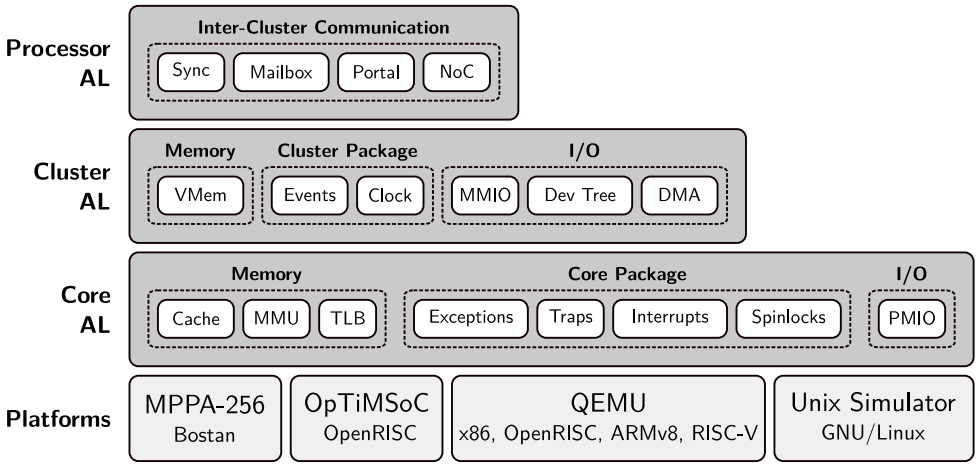


Figura 4 – Estrutura interna da HAL do Nanvix (PENNA, 2021).

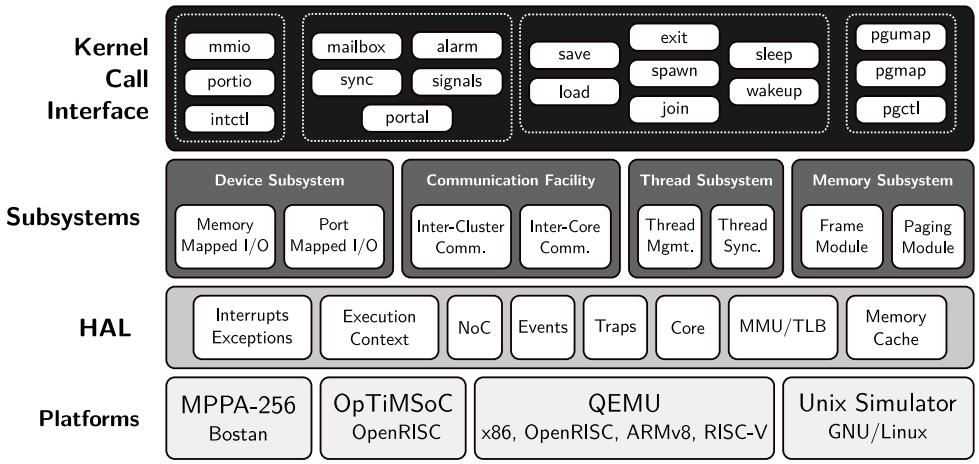
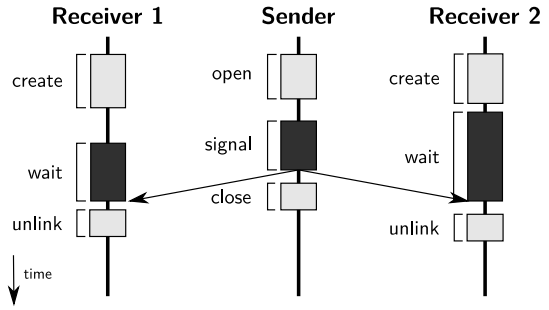


Figura 5 – Estrutura interna do *microkernel* do Nanvix (PENNA, 2021).

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 o desempenho do sistema quando múltiplas aplicações estão em execução simultânea no processador. No caso de aplicações paralelas, compostas por múltiplos processos (ou *threads*) que se comunicam, a disposição dos processos (ou *threads*) nos *clusters* se torna importante, pois a comunicação entre *clusters* próximos é mais rápida e 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,

(a) Modo $1 : N$.



(b) Modo $N : 1$.

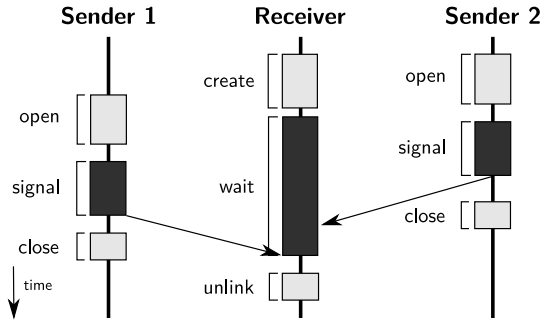


Figura 6 – Fluxo de execução da abstração *Sync* (PENNA, 2021).

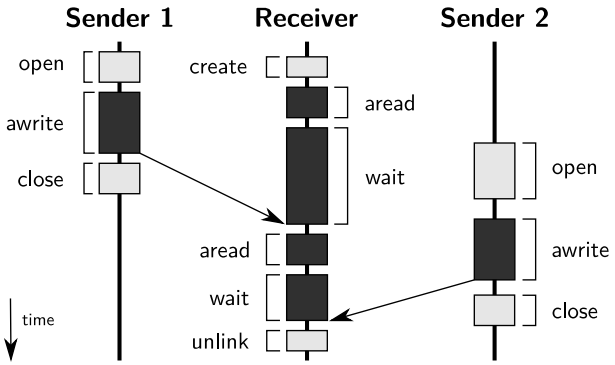


Figura 7 – Fluxo de execução da abstração *Mailbox* (PENNA, 2021)

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.

2.2.1.1 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).

O *Sync* pode ser operado duas maneiras distintas: o modo $1 : N$ e $N : 1$. No modo $1 : N$ (Figura 6(a)) um nó envia uma notificação a múltiplos nós, que estão esperando pelo sinal e são liberados após o recebimento do sinal. Em contraste, no modo $N : 1$ (Figura 6(b)), múltiplos nós enviam uma notificação a um único nó, que é liberado após o recebimento do sinal de todos os outros nós envolvidos (PENNA, 2021).

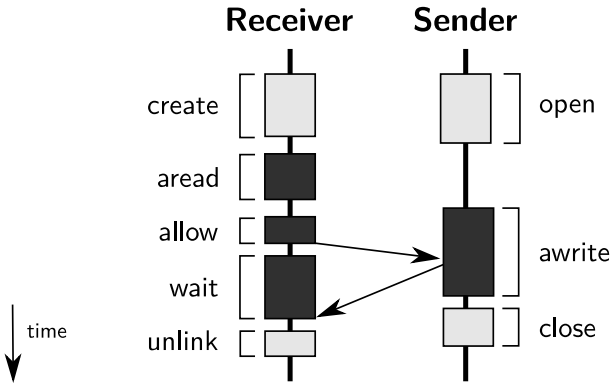


Figura 8 – Fluxo de execução da abstração *Portal* (PENNA, 2021)

2.2.1.2 Mailbox

A abstração *Mailbox* é responsável pelo suporte ao envio de mensagens de controle através da troca de pequenas mensagens de tamanho fixo. A abstração segue a semântica $N : 1$ e funciona da seguinte forma: um nó (destinatário da mensagem) possuí uma *Mailbox*, da qual lê mensagens, e múltiplos nós (remetentes da mensagem) podem escrever nessa *Mailbox* (PENNA, 2021). A Figura 7 ilustra o fluxo de execução da *Mailbox*.

2.2.1.3 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). A Figura 8 ilustra o fluxo de execução da abstração *Portal*.

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. O isolamento e independência das instâncias virtuais garantem à virtualização algumas vantagens, que são muito exploradas hoje, especialmente em ambientes *cloud* (MANOHAR, 2013). São elas: flexibilidade, portabilidade, escalabilidade e segurança.

O conceito de virtualização pode ser traçado desde a década de 50, durante a época dos *mainframes* e da memória virtual (ainda emergente na época) (CAMPBELL; JERONIMO, 2006). Nesse período, a preocupação era tornar um recurso físico acessível a múltiplos usuários simultaneamente. Essa motivação sustentou a evuloção da virtua-

lização, levando ao surgimento das Máquinas Virtuais (MVs) 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 MVs, que eram utilizadas por múltiplos usuários simultaneamente.

O retorno das pesquisas sobre virtualização ocorreu mais recentemente, na década de 90. 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 se mostrou uma solução viável por permitir que em um único servidor executassem diversas MVs rodando diferentes serviços, mantendo ainda assim a independência de execução entre eles. Isso significa que a interrupção ou quebra de um serviço não afeta os demais. Sendo assim, a virtualização garante uma maior flexibilidade e escalabilidade, além de reduzir os custos de manutenção e operação, afinal, dessa forma os recursos de *hardware* são utilizados de maneira mais eficiente e há a redução na quantidade de máquinas físicas para gerenciar.

Nos dias de hoje, a tendência de uso de MVs em servidores continua cresente. 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).

Nas próximas seções serão apresentados alguns tipos de virtualização e suas peculiaridades.

2.3.1 Virtualização total

Neste tipo de virtualização o objetivo é 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

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

2.3.3 Virtualização a Nível de Processo e Conteinerização

Virtualizar um processo ou aplicação é o processo de desacoplar a execução de um processo do sistema que o executa. Nesse contexto, o processo tem uma visão virtual única do sistema, de modo que a execução de cada aplicação ocorre independentemente uma da outra.

Neste tipo de virtualização, cada aplicação é isolada em um ambiente virtual, também conhecido como contêiner, no qual 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. Isso torna esse modelo uma fonte de inspiração para o desenvolvimento da virtualização em sistemas de memória limitados, que é o caso dos *lightweight manycores*.

2.3.4 Outros Tipos de Virtualização

(i) 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).

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

(iii) 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 otimizção.

2.3.5 Virtualização no Nanvix

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

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

Na abordagem original do Nanvix, o processo é dependente do *cluster* em que é alocado, o que afeta o suporte a migração e diminui a eficiência computacional, como detalhado na Seção 2.2. Nesse contexto, a virtualização torna-se útil por aumentar a mobilidade dos processos, o que possibilitaria o gerenciamento da distribuição dos processos no processador. Especificamente, este trabalho explora um modelo mais leve de virtualização para *lightweight manycores* baseada em contêineres. Contêineres são executados pelo SO como aplicações virtuais e não incluem um SO convidado, não sendo necessária a criação de duplicatas de SOs e resultando em um menor impacto no sistema de memória e requisitando menor complexidade do *hardware* (THALHEIM et al., 2018; SHARMA et al., 2016; ZHANG et al., 2018).

2.4 MIGRAÇÃO

A migração é o processo de transferência de uma aplicação ou MV de um ambiente a outro. A migração tem sido muito utilizada hoje em dia principalmente em ambientes *cloud* (IMRAN et al., 2022). Através da migração, os serviços são beneficiados com as suas vantagens, tais como:

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

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

(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/MVs são tranferidas a outro servidor durante esses intervalos de manutenção sem que os serviços sejam afetados.

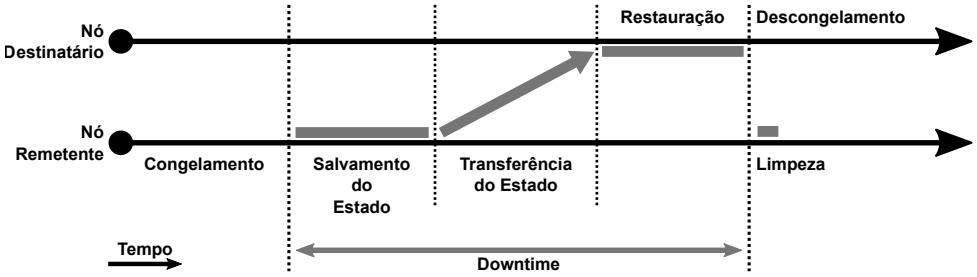


Figura 9 – Fluxo de execução da *cold migration*. Adaptado de (SYNYTSKY, 2016)

(iv) Gerenciamento de energia: através da migração, é possível gerenciar a alocação de MVs 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.

Nas próximas seções serão discutidos os principais tipos de migração.

2.4.1 Tipos de Migração

A migração pode ser classificada em dois tipos distintos: *cold migration* (*non-live migration*) e *hot migration* (*live migration*) (IMRAN et al., 2022). Esses tipos serão detalhados nas seções 2.4.1.1 e 2.4.1.2, respectivamente.

2.4.1.1 Cold migration

Também é conhecido como *non-live migration*. Neste tipo de migração, o sistema a ser migrado (contêiner ou MV) precisa ser desligado antes do processo começar. Este é o tipo de migração cujo fluxo de execução segue o seguinte. O sistema é desligado, o estado (*checkpoint*) do sistema é salvo e enviado ao SO ou MV 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 implica em um alto *down time* do sistema migrado devido à alta quantidade de dados a serem transferidos e pelo tempo extra para desligar e ligar o sistema novamente. Isso não é considerado aveitável atualmente, haja vista a grande quantidade de serviços que não podem parar sua execução (SINGH et al., 2022; IMRAN et al., 2022). A Figura 9 ilustra o fluxo de execução da *cold migration*.

2.4.1.2 Hot Migration

Em contraste com a *cold migration*, na *hot migration*, também conhecida como *live migration*, o sistema a ser migrado não precisa ser desligado antes do processo co-

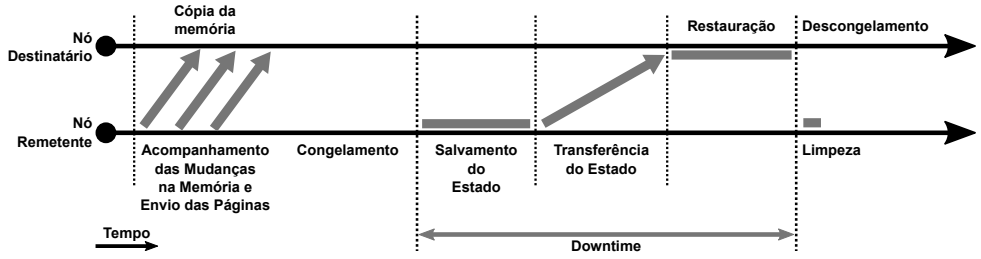


Figura 10 – Fluxo de execução da *pre-copy migration*. Adaptado de (SYNYTSKY, 2016)

meçar. Isso com o objetivo de maximizar a performance do sistema durante a migração, melhorar o uso da rede e reduzir o *down time* do sistema (IMRAN et al., 2022).

Este modelo a migração ainda pode ser classificado de acordo com a técnica utilizada para a transferência dos dados. As técnicas são: *pre-copy migration*, *post-copy migration* e migração híbrida, as quais serão detahadas, respecitvamente, nas Seções 2.4.1.2.1, 2.4.1.2.2 e 2.4.1.2.3.

2.4.1.2.1 *Pre-Copy migration*

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

Esse fluxo de migração faz com que o tempo total de migração seja maior que o do *cold migration*, já que há a retransmissão de dados, em especial páginas de memória. Em contrapartida, o *down time* é reduzido, pois o sistema executa normalmente na parte inicial da migração, quando é feita e enviada a pré-imagem ao destinatário (SINGH et al., 2022; IMRAN et al., 2022). A Figura 10 ilustra o fluxo de execução da *pre-copy migration*.

2.4.1.2.2 *Post-Copy migration*

Neste modelo, o remetente cria um estado parcial do sistema em execução e o envia ao destinatário. Nesse estado, está incluso apenas o essencial para a execução inicial do sistema, não abrangendo o esquema de paginação. Quando o sistema é restaurado no destinatário, ocorrerão várias faltas de páginas, as quais resultam em requisições de páginas ao remetente, o qual as envia ao destinatário. Quando todas as páginas são transferidas ao destinatário, o sistema é apagado do remetente (SINGH et al., 2022; IMRAN et al., 2022). A Figura 11 ilustra o fluxo de execução da *post-copy migration*.

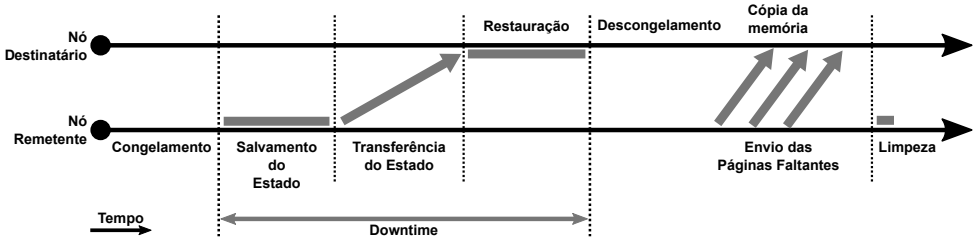


Figura 11 – Fluxo de execução da *post-copy migration*. Adaptado de (SYNYTSKY, 2016)

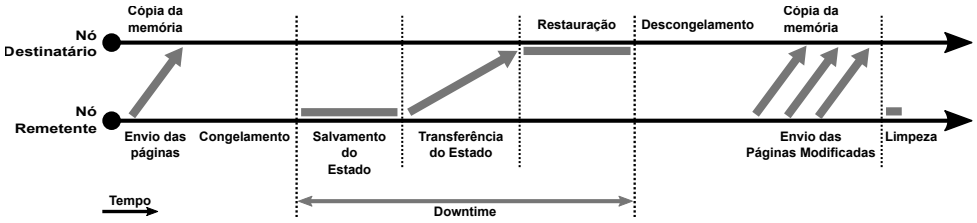


Figura 12 – Fluxo de execução da migração híbrida. Adaptado de (SYNYTSKY, 2016)

2.4.1.2.3 Migração Híbrida

Neste modelo, são utilizadas as técnicas de *pre-copy* e *post-copy* em conjunto. Inicialmente, tal como na *pre-copy migration*, é feita uma pré-imagem do sistema, que é enviada ao destinatário, enquanto o sistema executa normalmente. Em contraste com o método *pre-copy* nesta etapa não são reenviadas as páginas modificadas/atualizadas. Depois, o estado completo do sistema é enviado e as páginas modificadas/atualizadas são requisitadas ao remetente tal como na *post-copy migration*. Quando todas as páginas são transferidas ao destinatário, o sistema é apagado do remetente (SINGH et al., 2022; IMRAN et al., 2022). A Figura 12 ilustra o fluxo de execução da migração híbrida.

adicinar seção detalhando o sistema de tasks no background?

3 TRABALHOS RELACIONADOS

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

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

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

Apesar da maioria das pesquisas estarem voltadas à exploração desses benefícios e diminuição do tempo de migração e *down time* em ambientes *cloud*, há alguns autores preocupados com o desenvolvimento de soluções envolvendo virtualização e migração em ambientes de recursos restritos. Dessa forma, como a temática de limitação de recursos, especialmente de memória, é muito presente neste trabalho, serão abordados nas próximas seções algumas pesquisas desses autores i.e., pesquisas voltadas à busca pelo uso da virtualização/migração de forma mais leve e cujo impacto no *hardware* seja reduzido, se adaptando 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) adressa a possiblidade de implementação da virtualização em microcontroladores que utilizam *TrustZone*. *TrustZone* é uma tecnologia de *hardware* voltada à segurança, em que a execução de um sistema pode ser dividida entre normal e segura. Os autores afirmam que essa tecnologia pode ser explorada além das suas propriedades de segurança. Isso porque o *TrustZone* também provê certo nível de isolamento dos recursos, o que o torna viável de ser usado para virtualização, afinal o isolamento cria um ambiente seguro e propício para a execução simultânea e isolada de múltiplas Máquinas Virtuais (MVs).

O artigo 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 MVs nesses ambientes utilizando a tecnologia *TrustZone* para garantir o isolamento das MVs.

Os testes foram feitos num microcontrolador *Cortex-M4*. Conforme descrito pelos autores, a solução, de fato, garante o suporte à execução múltipla de MVs em microcontroladores.

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

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

Os autores evidenciam que os aparelhos IoT são usados na computação de borda para provomover o que chamamos de *Functions as a Service* (FaaS), que é um tipo de serviço oferecido por diversas plataformas, como a *Amazon AWS Lambda* e *Google Cloud Functions*. O problema é que esses dispositivos 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, os autores propõe um esquema de *checkpointing* utilizando *Docker* e *Checkpoint/Restore In Userspace* (CRIU). Através dessas tecnologias, os contêineres que não estão executando computação útil são interrompidos e salvos em disco, liberando espaço da memória para a execução de outro contêiner. Isso se torna extremamente útil quando consideramos funções de longa duração bloqueantes, já que durante o tempo de espera pelo sinal, a aplicação pode ser interrompida. Além disso, com o estado salvo em disco, a migração de contêineres entre dispositivos IoT de borda se torna possível. Dessa forma, além de reduzir o uso de recursos nos dispositivos de computação em borda, através da migração dos contêineres, outros benefícios surgem, como o balanceamento de carga e tolerância a falhas entre aparelhos IoT de borda.

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

possível.

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

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

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

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

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

3.4 COMPARAÇÃO DO PRESENTE TRABALHO COM OS TRABALHOS RELACIONADOS

O presente trabalho se asemelha com os trabalhos relacionados apresentados no sentido de aplicar a virtualização e migração em um sistema com recursos restritos. Contudo, em contraste com os trabalhos apresentados, a principal vantagem explorada com a virtualização é o aumento da mobilidade dos processos, possibilitando a migração de processos. A utilização eficiente dos recursos promovida pela virtualização, mesmo que necessária nos *lightweight manycores* pela limitação de recursos computacionais (em es-

pecial a memória) se torna uma vantagem indireta da virtualização. Isso porque o uso eficiente de *hardware* é provido mais pela migração (através da melhor disposição dos processos entre os *clusters*) do que pela virtualização em si.

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

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

Este trabalho de conclusão propõe-se a aumentar a independência dos processos no processador através do projeto e desenvolvimento do suporte à virtualização e migração de processos em *lightweight manycores*. Ambientes *cloud*, nos quais o sistema de memória é de alta capacidade, usufruem da utilização de Máquinas Virtuais (MVs) para isolar duplicatas inteiras de Sistemas Operacionais (SOs) com o auxílio da virtualização a nível de instrução (SHARMA et al., 2016). Em oposição, *lightweight manycores* não dispõem de centenas de GBs de memória, mas sim pequenas memórias locais. Isso associado a outras simplificações de *hardware* faz com que algumas técnicas de virtualização sejam impraticáveis nesses ambientes computacionais.

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

4.1 CONTEXTO DE UM PROCESSO

Para o desenvolvimento da virtualização, é necessário entender o contexto de um processo no Nanvix e as relações que atrelam o processo ao *cluster* e ao SO i.e., as dependências que o processo tem com os recursos reais de um *cluster* e com a estrutura interna do SO. De maneira geral, os módulos que compõem um processo são: *threads*, *syscalls*, sistema de memória e comunicação. Todos esses módulos de alguma forma têm dependências no *kernel* ou *cluster*. Essas dependências são ilustradas genericamente na Figura 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 explicita nessas estruturas para identificar quais variáveis são relacinadas a *kernel* ou usuário.

(ii) As estruturas responsáveis por armazenar as *syscalls*, seus parâmetros e retornos requisitadas pelos *slave cores* ao *master core* estão em espaço de *kernel*.

(iii) Todo o sistema de memória está armazenado no *kernel*. Tabelas de diretórios, tabelas de página, *Translation Lookaside Buffer* (TLB), 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

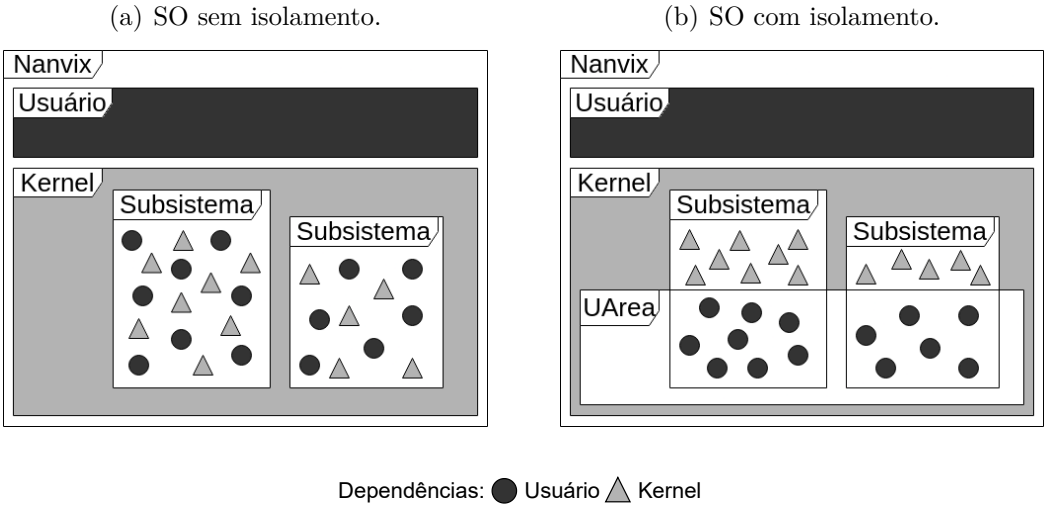


Figura 13 – Diferença da estrutura do Nanvix com e sem a *User Area*.

comunicação entre *clusters* estão armazenadas em espaço de *kernel* e referenciam uma interface física do *cluster* em que o processo está executando.

Sendo assim, o desenvolvimento da virtualização nesse sistema implica no isolamento dessas dependências em um arranjo, que é o contêiner. A ideia principal é garantir que o contêiner contenha tudo o que é necessário para o processo executar. Isso inclui todos os dados e códigos de usuário e todas as dependências do processo com o *kernel* e *cluster*. Isso deve ser feito de uma maneira que permita com que o *kernel* execute qualquer contêiner como uma aplicação virtual i.e., deve ser possível que o contêiner se conecte ao *kernel* de forma que consiga utilizar os recursos e serviços de *kernel* sem que interfira em sua estrutura interna.

4.2 ISOLAMENTO DO CONTEXTO DE UM PROCESSO DE USUÁRIO

Para a virtualização de processos através da conteinerização, é recomendável que as informações relevantes para a manipulação dos processos em execução estejam isoladas das informações internas do próprio SO para que os recursos de *hardware* sejam utilizados de maneira eficiente (CHOUDHARY et al., 2017). A Figura 13(a) ilustra como os subsistemas do Nanvix são originalmente estruturados. Não há uma divisão explícita do que são dados para funcionamento interno do SO ou dependências locais do processo. Esta abordagem torna algumas das funcionalidades do SO onerosas porque ela dificulta o acesso às informações do processo e impacta partes independentes do sistema, e.g., migração e segurança dos processos.

4.2.1 Divisão de Dados e Instruções

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

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

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

4.2.2 *User Area*

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

Detalhadamente, a UArea mantém informações sobre:

(i) *Threads* ativas, incluindo identificadores, pilhas de execução e contextos;
(ii) Filas de escalonamento de *threads*;
(iii) Variáveis de controle interno do sistema de *threads*, como quantidade de *threads*

ativas;

(iv) Tabela de gerenciamento de chamadas de sistema; e

(v) Estruturas de gerenciamento de memória e.g., sistema de paginação

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

read again from here

4.3 MIGRAÇÃO DE PROCESSOS

Como aplicação direta do isolamento do processo, a migração de processos torna-se viável. Especificamente, nós eliminamos a necessidade de descobrir quais são e onde estão as informações que compõem o estado de um processo dentro do Nanvix através da criação de uma instância isolada do espaço do usuário via conteinerização, facilitando a transferência de seu contexto. Isso só é possível porque os *clusters* possuem uma estrutura de *kernel* idêntica (devido às mudanças desenvolvidas no processo de compilação detalhados na Subseção 4.2.1). Por este motivo, eliminamos o envio de dados redundantes entre *clusters* referentes à instância local do SO, atenuando o impacto da migração sobre a NoC.

4.3.1 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 do projeto, implementamos o algoritmo *hot migration* para migração de processos. A técnica de *hot migration* migra a aplicação durante sua execução, copiando as páginas de memória e o estado de execução da aplicação e restaurando a aplicação depois da transferência completa dos dados. A seguir é detalhado o fluxo de execução da migração:

1. **Congelamento da execução do processo em um estado consistente.**
   Antes do envio da aplicação a outro *cluster*, é necessário que o processo esteja em um estado consistente e estático. Isso significa que durante o processo de migração é preciso que todas as operações dele sejam pausadas. Isso é feito objetivando evitar inconsistências que podem ser causadas por condições de corrida e.g., impedir perda de instruções, retornos de chamadas de sistemas, sinais de sincronização, etc. Para atingir esse estado consistente, a chamada de sistema *freeze* é invocada. Esta é

uma chamada de sistema que é tratada apenas pelo *master core*. Especificamente, esta chamada ativa uma variável interna do SO que impede o escalonamento de *threads* de aplicação (*threads* que não executam no *master core*) e envia um sinal de reescalonamento para todos os *slave cores*, para que as *threads* de usuário saiam de execução o mais rápido possível. Isso garante uma pausa na aplicação sem que o SO seja impedido de executar, o que é imprescindível para a migração, já que as informações do processo precisam ser enviadas pelas interfaces NoC do *cluster* remetente, o que exige que o SO atenda às requisições de envio de dados. 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 *task* de sistema, que é executada no *master core*, é criada para o envio dos dados ao *cluster* destinatário. Através das abstrações de comunicação *Mailbox* e *Portal*, as seções de dados e instruções do processo são enviadas ao *cluster* destinatário. Logo após, a UArea é enviada. 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*.

detalhar como funcionam as tasks e threads em cada cluster: remetente e destinatario

como funciona a migração para um cluster com nenhum processo alocado

5 METODOLOGIA DE AVALIAÇÃO

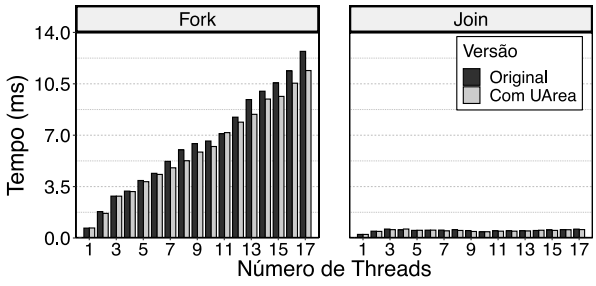
6 RESULTADOS PARCIAIS

A solução foi avaliada em etapas anteriores ao desenvolvimento atual do trabalho e os resultados seguintes englobam apenas o susbsistema de *threads* do Nanvix. 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 valores médios de 100 replicações de cada experimento para garantir 95% de confiança estatística, resultando em um desvio padrão inferior a 1%.

O experimento de manipulação de *threads* mensura os impactos na criação e junção através de diferentes perspectivas. Especificamente, coletamos o tempo de execução, desvios e faltas ocorridas na *cache* de dados e de instrução (Figura 14). Os resultados apresentam um aumento no desempenho das operações de manipulação quando utilizamos a *User Area* (UArea) porque exploramos melhor a localidade espacial dos dados, o que, consequentemente, diminui 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 *down time* médio da aplicação, i.e., o tempo que a aplicação demorou para restaurar a execução no *cluster* destinatário após a migração, foi de 226 ms. A média de tempo para o *cluster* remetente enviar todos os dados foi de 218 ms.

(a) Tempo de criação de *threads*.



(b) Métricas do *kernel*.

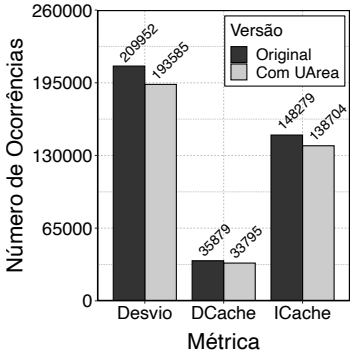


Figura 14 – Impactos da virtualização sobre a manipulação de *threads*.

7 CONCLUSÕES

Neste trabalho, foi explorado um modelo de virtualização leve baseada em contêineres que considera as restrições arquiteturais dos *lightweight manycores*, adaptando-se as suas restrições, principalmente relacionadas à memória. A virtualização proposta visa melhorar a mobilidade e gerenciamento de processos para *lightweight manycores* no contexto de em um Sistema Operational (SO) distribuído. Os resultados mostraram que o isolamento das dependências de um processo aumentaram o desempenho de operações do *kernel* e suportaram de fato 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;
(iii) Realizar uma maior quantidade de experimentos para avaliar os sobrecustos introduzidos pela abordagem proposta.

REFERÊNCIAS

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

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.

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.

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.

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.

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.

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.

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. Using The Nanvix Operating System in Undergraduate Operating System Courses. In: **VII Brazilian Symposium on Computing Systems Engineering**. Curitiba, Brazil: [s.n.], 2017. Disponível em: https://hal.archives-ouvertes.fr/hal-01635880.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.