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

Nicolas Vanz[1], **João Vicente Souto**[1], **Márcio Castro**[1]

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

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

Resumo. *Este artigo apresenta a proposta e desenvolvimento da funcionalidade de migração de processos no Nanvix, um sistema operacional distribuído projetado para lightweight manycores, através de uma abordagem de virtualização leve baseada em contêineres. Os resultados experimentais mostram que a solução permite melhorar o desempenho do subsistema de threads em comparação a solução padrão implementada no Nanvix, assim como reduzir os desvios de instruções e faltas de cache, atingindo um downtime entre 19 ms e 113 ms durante as migrações.*

1. Introdução

Atualmente, a eficiência energética dos sistemas computacionais revela-se tão importante quanto seu desempenho. Nesse contexto, processadores do tipo *lightweight manycore* foram desenvolvidos de forma a prover um melhor compromisso entre desempenho e consumo energético [Francesquini et al. 2015]. Esses processadores são classificados como *Multiprocessor System-on-Chip*s (MPSoCs) e são caracterizados por: (i) integrar 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 de memória distribuída restritivos, compostos por pequenas memórias locais; e (vi) apresentar *clusters* heterogêneos.

Apesar dos processadores *lightweight manycores* oferecerem uma melhor eficiência energética que os *multicores*, suas características arquiteturais introduzem severos desafios de programabilidade [Castro et al. 2016]. Nesse contexto, Sistemas Operacionais (SOs) distribuídos foram propostos para esta classe de processadores, destacando-se por proverem um ambiente de programação mais robusto e rico [Penna et al. 2019a]. 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. 2019b].

Porém, o mecanismo de gerenciamento de processos do Nanvix impossibilita a mobilidade dos processos dentro do processador, pois cada processo passa todo seu ciclo de vida em um 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, migrar processos que se comunicam intensamente para *clusters* próximos permite reduzir a latência de comunicação, aumentando, assim, o desempenho das aplicações paralelas [Vanz et al. 2022].

Neste contexto, este artigo explora um modelo leve de virtualização baseado no conceito de contêineres para fornecer as abstrações necessárias de forma a viabilizar a migração de processos no Nanvix. Ao desvincular os recursos locais utilizados por um processo dentro do Nanvix, a abordagem proposta permite prover maior controle e mobilidade de processos no processador. A solução proposta foi implementada no Nanvix[1] e compreende as seguintes contribuições: (i) modificações no *kernel* para permitir o isolamento do contexto dos processos; (ii) alteração na estrutura do binário do SO, separando dados do *kernel* e do usuário; (iii) isolamento dos dados que são gerenciados pelo *kernel* mas pertencem ao contexto do processo de usuário em uma região de memória denominada *User Area* (UArea); e (iv) criação de um novo *daemon* para controlar a migração dos processos, o qual provê uma nova chamada de sistema para migração de processos. Os experimentos foram executados no Kalray MPPA-256 [Penna et al. 2021], um *lightweight manycore* que integra 288 *cores* de baixa frequência em um único *chip*. Neste processador, os *cores* são organizados em 20 *clusters*, os quais possuem somente 2 MB de memória local, e a comunicação entre *clusters* é feita através de NoCs. Os resultados obtidos mostraram que a solução proposta permitiu melhorar o desempenho no subsistema de *threads* e reduzir os desvios de instruções, faltas na *cache* de dados e faltas na *cache* de instruções, atingindo um *downtime* entre 19 ms e 113 ms durante as migrações, dependendo da quantidade de recursos utilizados pelos processos.

Este artigo está organizado da seguinte forma. Na Seção 2 são apresentados alguns trabalhos relacionados. A Seção 3 expõe as contribuições deste trabalho, destacando as modificações realizadas no Nanvix para viabilizar a migração de processos. Então, a Seção 4 apresenta e discute os resultados obtidos. Por fim, na Seção 5 são apresentadas as conclusões do trabalho e apontados possíveis trabalhos futuros.

2. Trabalhos Relacionados

A literatura apresenta algumas soluções envolvendo virtualização e migração em sistemas mais restritivos, como sistemas de tempo real e sistemas críticos. Nesses trabalhos, as pesquisas são 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.

Pinto et al. [Pinto et al. 2019] abordaram a possibilidade de implementação da virtualização em microcontroladores que utilizam *TrustZone*, uma tecnologia de *hardware* voltada à segurança que provê certo nível de isolamento dos recursos. Os autores propuseram uma solução que usa um *hypervisor* mais leve para gerenciar as Máquinas Virtuais (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.

O trabalho desenvolvido por *Karhula et al.* [Karhula et al. 2019] envolveu migração de contêineres entre dispositivos *Internet of Things* (IoT) de borda como solução para a diminuição do uso de recursos nesses sistemas restritivos. Os autores propuseram um esquema de *checkpointing* utilizando *Docker* e *Checkpoint/Restore In Userspace* (CRIU), em que contêineres que estão em estado de espera são temporariamente

[1]O código-fonte da solução proposta está disponível no repositório oficial do Nanvix: https://github.com/nanvix

interrompidos e salvos em disco, liberando espaço na memória para execução de outros contêineres. Os testes foram feitos em uma *Raspberry Pi 2 Model B*, a qual executava diversos contêineres com aplicações de longa duração e que simulavam o comportamento bloqueante. Os resultados mostraram uma economia no uso de recursos, em especial da memória.

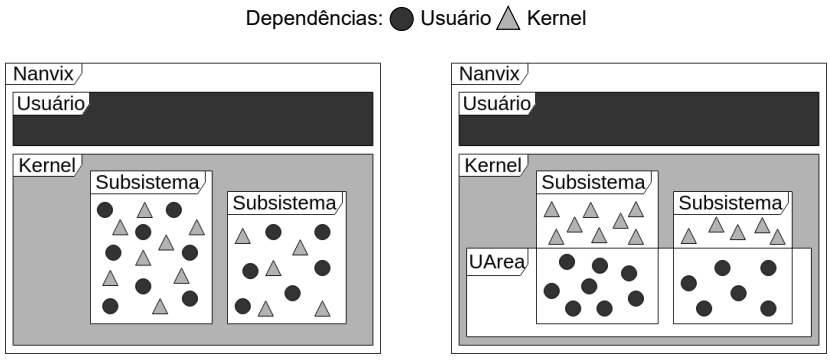
Morabito et al. [Morabito et al. 2017] abordaram o uso de virtualização no desenvolvimento de aplicações para carros inteligentes. Os autores desenvolveram um sistema que utiliza contêineres *Docker* para criar uma camada de abstração a nível de processo. A solução proposta inclui um escalonador de contêineres, capaz de gerenciar os recursos de *hardware* de forma a garantir que os contêineres sejam executados de maneira eficiente em função do nível de criticalidade das aplicações, sem que haja desperdício de recursos. A proposta foi testada em um *Raspberry Pi 3* e os resultados mostram que o escalonador foi capaz de gerenciar os recursos de maneira eficiente, priorizando as tarefas de maior prioridade.

A pesquisa desenvolvida por *Abeni et al.* [Abeni et al. 2019] abordou 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*). Os autores propuseram um escalonador que estende um escalonador já presente no *kernel* do Linux. Os experimentos mostraram que o escalonador proposto utiliza melhor os *cores* quando comparado 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 de forma mais eficiente.

De maneira similar a [Karhula et al. 2019, Morabito et al. 2017, Abeni et al. 2019], e em contraste com [Pinto et al. 2019], neste artigo exploramos 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. Diferentemente dos trabalhos citados, a solução proposta neste artigo não depende de nenhuma ferramenta externa para a manipulação de contêineres como o *Docker* [Morabito et al. 2017, Karhula et al. 2019] ou o LXC [Abeni et al. 2019]. Isso porque o trabalho foi feito sobre um ambiente não usual (*lightweight manycores*) utilizando um SO que não possui algum suporte para conteinerização (Nanvix). Na solução proposta, 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* há 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. A solução proposta neste trabalho é detalhada na próxima seção.

3. Virtualização e Migração de Processos no Nanvix

O Nanvix surgiu com a proposta de resolver os problemas de programabilidade e portabilidade em *lightweight manycores* [Penna et al. 2019b]. Apesar de ser uma abordagem promissora, o SO ainda possui limitações, como a dependência que um processo tem com o *cluster* em que executa. Apenas um processo pode ser executado por *cluster* e



(a) Original. **(b)** Com UArea.

Figura 1. Diferença da estrutura do Nanvix com e sem UArea.

os serviços envolvendo o processo têm dependências com recursos físicos do *cluster*. Essa falta de mobilidade dos processos afeta o desempenho do sistema porque impede a redistribuição dos processos de forma dinâmica. Neste contexto, 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*.

3.1. Limitações da Versão Original do Nanvix

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 1a e algumas delas são listadas a seguir: (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; (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*; (iv) o sistema de comunicação tem dependências tanto no *kernel* quanto a recursos físicos do *cluster*.

A geração original de um executável do Nanvix compilava todos os níveis em bibliotecas estáticas (*Hardware Abstraction Layer* (HAL), *microkernel*, *libnanvix*, *ulibc* e *multikernel*) e as juntava com a aplicação do usuário de forma a misturar o que é *kernel* do que é usuário. Para viabilizar a implementação de algum tipo de virtualização no Nanvix, é importante isolar essas dependências em um arranjo que guarde todas as informações necessárias para a execução de um processo, sejam elas manipuladas pelo usuário ou pelo *kernel*. Neste artigo, chamamos este 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.

A Figura 1a 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.

3.2. Isolamento do Contexto de Processos e *User Area* (UArea)

Visando a separação das informações entre usuário e *kernel*, nós adaptamos o *script* de ligação original do Nanvix. Na nova versão, as seções `.text`, `.data`, `.bss` e `.rodata` dos arquivos binários compilados são renomeados, especificando a qual camada de abstração tal arquivo pertence (*kernel* ou usuário). 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.

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 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.

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 UArea (Figura 1b), a qual 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* com 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.

Devido às limitações de memória e simplificações de *hardware* existentes em *lightweight manycores*, as técnicas clássicas de virtualização utilizadas em nuvens computacionais são impraticáveis de serem implementadas. Visando atenuar o impacto da virtualização na memória, o presente trabalho explora um modelo de virtualização mais leve, baseado em contêineres adaptado para *lightweight manycores* e Nanvix. Nessa solução, o SO executa os contêineres como aplicações virtuais, sem 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].

Diferentemente da abordagem original de conteinerização, 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.

3.3. 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 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 e ligação. Como consequência da uniformidade do *kernel* entre os *clusters*, eliminamos o envio de dados redundantes relacionados à instância local do SO durante a migração. Dessa forma, apenas os dados relacionados ao contêiner são migrados/enviados, o que atenua o impacto da migração sobre a NoC.

Para a migração de um processo entre *clusters* foi desenvolvida uma rotina de migração. A funcionalidade é similar ao 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 implementada a técnica *hot migration*, em que a aplicação é migrada enquanto é executada, utilizando a técnica *pre-copy*. 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.

3.3.1. *Daemon* e Fluxo 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 and Castro 2022], as quais são inicializadas e conectadas durante a inicialização do módulo de migração, que ocorre durante o *boot*. Além disso, nesse período ainda são criados a porta de *Mailbox* por onde o *daemon* recebe as requisições de migração e o *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

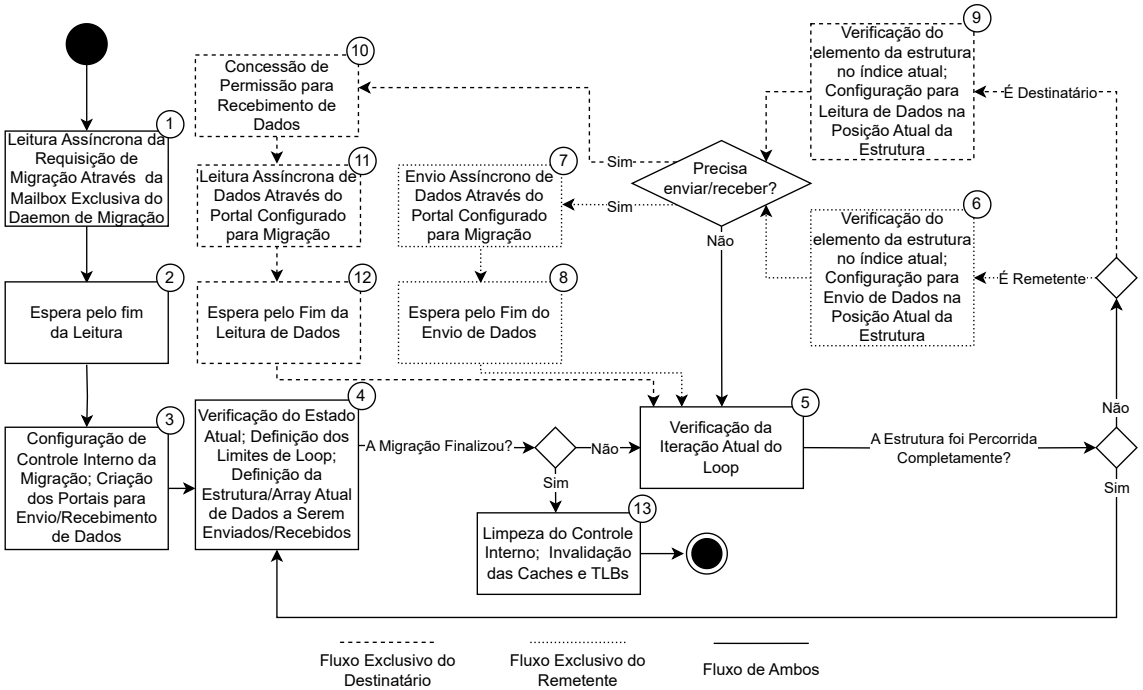


Figura 2. Fluxo de *tasks* de migração.

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. Genericamente, podemos dizer que o fluxo de migração é composto por três passos principais:

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 a fim de evitar inconsistências no sistema. Para prover tal garantia, uma nova chamada de sistema, *freeze*, foi desenvolvida, a qual é invocada no início do processo de migração. Ela ativa uma variável interna do SO que impede o escalonamento de *threads* de aplicação e envia um sinal de reescalonamento para todos os *slave cores*, para que as *threads* de usuário saiam de execução o mais rápido possível. Após o travamento no escalonamento de *threads* de usuário, novas chamadas de sistema requisitadas pela aplicação não podem ocorrer. Após o congelamento, o processo é considerado consistente e seu contexto está pronto 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*.

Todo o controle do processo de migração é feito com uso do mecanismo de *tasks*, onde são descritas as tarefas e suas dependências. O subsistema de *tasks* se responsabiliza por executá-las. A Figura 2 ilustra o fluxo de execução da migração, a qual é composta por 13 *tasks*. Nela, cada quadro corresponde a uma *task* e a descrição de cada quadro indica o que a *task* faz. É importante salientar que apenas um subconjunto das *tasks* é executado pelo *cluster* de origem e de destino do processo a ser migrado: *tasks* representadas com linha contínua são executadas pelo remetente e destinatário ao passo que *tasks* representadas com linha tracejada são executadas ou pelo remetente ou pelo destinatário. Por fim, destaca-se que neste fluxo todas as comunicações são assíncronas, ou seja, 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.

3.3.2. 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* (este *cluster* é descongelado ao fim da migração). 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. Por fim, é importante destacar que o *daemon* foi projetado de forma a possibilitar a expansão das suas funcionalidades.

4. Resultados Experimentais

4.1. Metodologia de Avaliação

Três perguntas guiaram o desenvolvimento dos experimentos para analisar a virtualização e a migração de processos no Nanvix:

Q1 Qual impacto o isolamento da 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 são realizadas migrações em paralelo?

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 (18 *threads*). Especificamente, coletamos o tempo de execução, desvios e faltas ocorridas na *cache* de dados e de instrução.

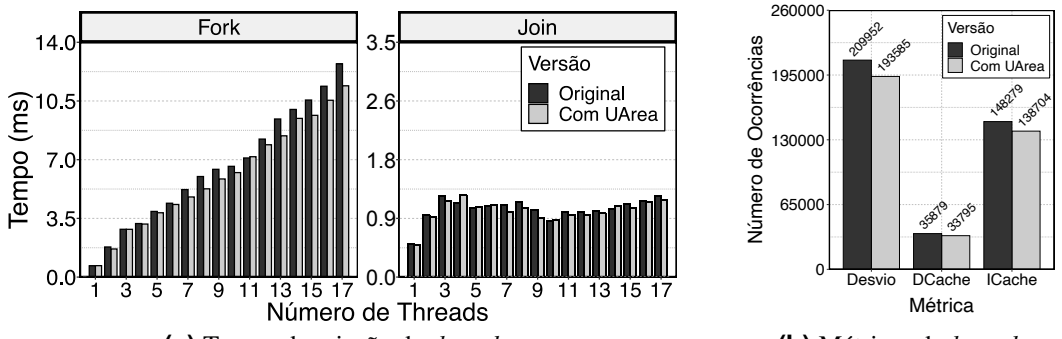
Para responder a pergunta **Q2**, foi desenvolvido um experimento que mensura o tempo de transferência de um processo entre *clusters* de acordo com os recursos utilizados i.e., *threads* e quantidade de páginas de memória alocadas dinamicamente. Neste teste, variou-se a quantidade de páginas de memória alocadas dinamicamente entre 0 e 32; e *threads* usadas pela aplicação entre 1 e 17.

Por fim, para responder a pergunta **Q3**, foi desenvolvido um experimento que mensura o *downtime* da aplicação quando varia-se 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*.

Todos os testes foram realizados no processador Kalray MPPA-256 [Penna et al. 2021]. Para obter uma maior confiança estatística foram executadas 20 repetições de cada experimento. Os resultados apresentados correspondem a média dos valores obtidos em cada experimento.

4.2. Análise do Impacto da UArea e do Isolamento de Código e Dados de Usuário sobre a Manipulação de *threads* (Q1)

Ao analisar o isolamento da aplicação através da UArea e separação do binário de usuário e *kernel*, percebe-se um impacto positivo no subsistema de *threads* do Nanvix. A Figura 3a comprova que o Nanvix obteve um leve ganho de desempenho na operação de



(a) Tempo de criação de *threads*.

(b) Métricas do *kernel*.

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

criação de *threads* e não introduziu mudanças significativas na operação de junção de *threads*. A Figura 3b apresenta a origem do aumento da desempenho na operação de criação de *threads*, exibindo a quantidade de desvios e faltas nas *caches* de dados e instrução. Nota-se 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.

4.3. Análise do *downtime* da Migração (Q2)

A Figura 4 apresenta os tempos de migração de processos para um número variável de *threads* e páginas alocadas por elas. Como o esperado, o *downtime* aumenta quanto maior for o número de páginas e *threads*, com um mínimo de 5,26% e um 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 (é preciso migrar também duas pilhas de execução, que totalizam duas páginas de memória e as estruturas de manipulação dessa *thread*).

Como podemos observar na Figura 4, ocorre uma disparidade no tempo de migração 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%.

4.4. Análise do Impacto de Migrações Paralelas (Q3)

Como ilustrado pela Figura 5, o experimento que mensurava o *downtime* da aplicação quando múltiplas migrações aconteciam simultaneamente, mostrou que a quantidade de

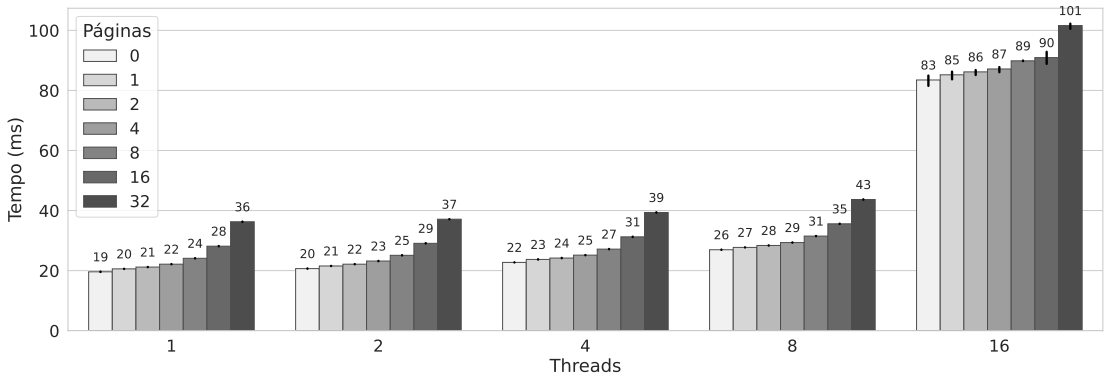


Figura 4. *Downtime* do teste durante o experimento de migração.

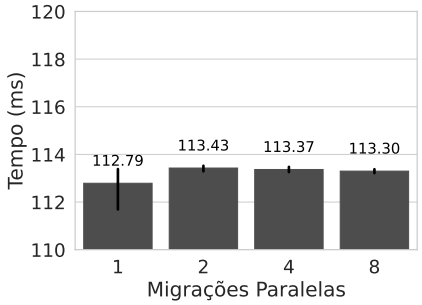


Figura 5. *Downtime* do teste com migrações em paralelo.

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 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.

5. Conclusão

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

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 conteinerização 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 113 ms, dependendo da quantidade de recursos utilizados. O isolamento das dependências de um processo impactou positivamente o sistema, aumentando 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.

Como trabalho futuro pretende-se desenvolver um escalonador de processos no Nanvix que utilize o *daemon* de migração de processos proposto para realizar, de forma transparente e automática, a migração dos processos em tempo de execução levando em consideração as necessidades de processamento e de comunicação entre processos.

Referências

Abeni, L., Balsini, A., and Cucinotta, T. (2019). Container-based real-time scheduling in the linux kernel. *ACM SIGBED Review*, 16(3):33–38.

Castro, M., Francesquini, 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.

Francesquini, E., Castro, M., Penna, P. H., Dupros, F., Freitas, H., Navaux, P., and Méhaut, J.-F. (2015). On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. *Journal of Parallel and Distributed Computing (JPDC)*, 76(C):32–48.

Karhula, P., Janak, J., and Schulzrinne, H. (2019). Checkpointing and migration of IoT edge functions. In *International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*, pages 60–65.

Morabito, R., Petrolo, R., Loscrì, V., Mitton, N., Ruggeri, G., and Molinaro, A. (2017). Lightweight virtualization as enabling technology for future smart cars. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1238–1245.

Penna, P. H., Souto, J., Lima, D. F., Castro, M., Broquedis, F., Freitas, H., and Méhaut, J.-F. (2019a). On the performance and isolation of asymmetric microkernel design for lightweight manycores. In *Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8, Natal, Brazil.

Penna, P. H., Souto, J. V., Uller, J. F., Castro, M., Freitas, H., and Méhaut, J.-F. (2021). Inter-kernel communication facility of a distributed operating system for NoC-based lightweight manycores. *Journal of Parallel and Distributed Computing*, 154:1–15.

Penna, P. H., Souza, M., Junior, E. P., Souto, J., Castro, M., Broquedis, F., Cota de Freitas, H., and Mehaut, J.-F. (2019b). RMem: An OS Service for Transparent Remote Memory Access in Lightweight Manycores. In *International Workshop on Programmability and Architectures for Heterogeneous Multicores (MultiProg)*, pages 1–16, Valencia, Spain.

Pinto, S., Araujo, H., Oliveira, D., Martins, J., and Tavares, A. (2019). Virtualization on TrustZone-enabled microcontrollers? Voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304. IEEE.

Sharma, P., Chaufournier, L., Shenoy, P., and Tay, Y. (2016). Containers and virtual machines at scale: A comparative study. In *International Middleware Conference (Middleware)*, pages 1–13.

Souto, J. V. and Castro, M. (2022). Improving concurrency and memory usage in distributed operating systems for lightweight manycores via cooperative time-sharing lightweight tasks. *Journal of Parallel and Distributed Computing*, 174:2–18.

Thalheim, J., Bhatotia, P., Fonseca, P., and Kasikci, B. (2018). Cntr: Lightweight OS Containers. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 199–212.

Vanz, N., Souto, J. V., and Castro, M. (2022). Virtualização e migração de processos em um sistema operacional distribuído para lightweight manycores. In *Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)*, pages 45–48. SBC.

Zhang, Q., Liu, L., Pu, C., Dou, Q., Wu, L., and Zhou, W. (2018). A comparative study of containers and virtual machines in big data environment. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 178–185. IEEE.