UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

MARCOS ANTONIO MOREIRA VIEIRA

ANÁLISE DO DESEMPENHO DE SISTEMAS OPERACIONAIS DE TEMPO REAL PARA DISPOSITIVOS IOT LOW-END NO CONTEXTO MULTI-CORE

QUIXADÁ

2026

MARCOS ANTONIO MOREIRA VIEIRA

ANÁLISE DO DESEMPENHO DE SISTEMAS OPERACIONAIS DE TEMPO REAL PARA DISPOSITIVOS IOT LOW-END NO CONTEXTO MULTI-CORE

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. André Ribeiro Braga.

QUIXADÁ

2026

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

V716a Vieira, Marcos Antonio Moreira.
Análise do desempenho de Sistemas Operacionais de Tempo Real para dispositivos IoT no contexto Multi-core / Marcos Antonio Moreira Vieira. – 2026.
47 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Computação, Quixadá, 2026.
Orientação: Prof. Dr. André Ribeiro Braga.

1. Sistemas Operacionais de Tempo Real. 2. Multiprocessamento simétrico. 3. Internet Das Coisas. 4. benchmarking. I. Título.

CDD 621.39

Dedico este trabalho à minha mãe.

AGRADECIMENTOS

Agradeço à minha mãe Francisca Iraneide Moreira, por ser o pilar que me manteve de pé e me sustentou durante toda a minha graduação, e também pelo apoio incondicional que me deu.

À minha irmã Kamilly Vitória Moreira Lima, cuja presença em minha vida foi o combustivel que não permitiu que eu desistisse em momento algum.

À minha família, em especial à minha Avó Ivaniza Martins, por me apoiar nessa jornada desde antes mesmo de eu iniciá-la.

Ao meu amigo e irmão, Guilherme Barros Vieira de Araújo, pelo apoio imensurável que me prestou, por estar ao meu lado nos momentos mais difíceis e por cada lembrança construída ao longo desse período.

Aos meus grandes amigos que construí durante essa tragetória, pelo apoio, troca de conhecimento e incentivo durante os desafios enfrentados ao longo da graduação, em especial, a José Adrian, Felipe Feitosa, Arthur e Maria Eduarda.

À minha companheira, Jaiane Sampaio França, por seu suporte emocional, paciência e cumplicidade ao longo dessa jornada acadêmica, sendo uma presença essencial nessa caminhada.

Por fim, agradeço a Deus, por me conceder a força necessária para enfrentar todos os desafios dessa jornada.

RESUMO

Este trabalho avalia o desempenho de dois *Real-Time Operating Systems* / Sistemas Operacionais de Tempo Real (*RTOS*s) de código aberto, *FreeRTOS* e *NuttX*, na *Raspberry Pi Pico* (RP2040), comparando execução em *single-core* e em *Symmetric Multiprocessing* / Multiprocessamento Simétrico (*SMP*). O objetivo é verificar como decisões internas de sincronização do *kernel* afetam a escalabilidade em microcontroladores *low-end multi-core*. Para isso, implementam-se três cenários complementares: execução de primitivas do *kernel* de forma isolada (para caracterizar custo estrutural em *multi-core*), um conjunto de cargas integradas que exercita serviços típicos do sistema e um cenário de contenção controlada em que tarefas concorrem pelo mesmo recurso. Em todos os testes, executa-se cada rotina em janelas temporais fixas, obtendo *throughput* por contagem de iterações, e registra-se o pico de uso de memória dinâmica (*heap*) como métrica complementar. Os resultados mostram comportamento semelhante em cargas predominantemente computacionais, porém diferenças marcantes quando o custo passa a ser dominado por escalonamento e sincronização. Em *SMP*, o *FreeRTOS* tende a perder desempenho com maior intensidade quando há entradas frequentes em região crítica, pois a proteção interna em *multi-core* pode serializar trechos de execução mesmo sem contenção explícita do mesmo objeto. Já o *NuttX* mantém boa escalabilidade em baixa contenção ao se beneficiar de caminhos rápidos em algumas operações, mas reduz gradativamente sua vantagem à medida que a contenção cresce e o caminho completo de sincronização é acionado com maior frequência. Observa-se também maior pico de *heap* no *NuttX*, compatível com a execução com mais serviços do sistema ativos durante os experimentos.

Palavras-chave: *RTOS*; *multi-core*; *SMP*; sincronização; contenção de recursos.

ABSTRACT

This work evaluates the performance of two open-source *RTOS*s, *FreeRTOS* and *NuttX*, on the *Raspberry Pi Pico* (RP2040), comparing *single-core* and *SMP* execution. The goal is to assess how internal *kernel* synchronization decisions impact scalability on low-end multicore microcontrollers. Three complementary scenarios are implemented: isolated *kernel* primitives (to characterize structural overhead under *multicore*), integrated workloads exercising typical system services, and a controlled contention scenario in which tasks compete for the same shared resource. Each routine runs in fixed time windows, computing throughput by iteration counting, while peak dynamic memory usage (*heap*) is recorded as a complementary metric. Results show similar behavior for compute-bound workloads, but clear differences when scheduling and synchronization dominate the cost. Under *SMP*, *FreeRTOS* tends to degrade more sharply with frequent critical-section entries, since multicore protection may serialize execution even without explicit contention on the same *kernel* object. In contrast, *NuttX* preserves scalability under low contention by benefiting from fast paths in some operations, but gradually loses this advantage as contention increases and full synchronization paths are triggered more often. *NuttX* also reaches a higher peak heap usage, consistent with running with more system services enabled during the experiments.

Keywords: *RTOS*s; multicore; *SMP*; synchronization; resource contention.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AMP	*Asymmetric Multiprocessing* / Multiprocessamento Assimétrico
API	*Application Programming Interface* / Interface de Programação da Aplicação
GPOS	*General-Purpose Operating System* / Sistema Operacional de Propósito Geral
IoT	*Internet of Things* / Internet das Coisas
RTOS	*Real-Time Operating System* / Sistema Operacional de Tempo Real
SMP	*Symmetric Multiprocessing* / Multiprocessamento Simétrico
SO	Sistema Operacional

SUMÁRIO

1 INTRODUÇÃO

A *Internet of Things* / Internet das Coisas (*IoT*) vem se estabelecendo como uma das mais significativas revoluções tecnológicas dos últimos anos, interligando bilhões de aparelhos que recolhem, processam e compartilham informações em tempo real (Amara *et al.*, 2022). Prevê-se que até 2030, a quantidade de dispositivos IoT conectados ultrapassará 30 bilhões, abrangendo setores como a indústria, energia, saúde e comércio (Amara *et al.*, 2022).

Esses dispositivos podem ser classificados em *low-end*, *middle-end* e *high-end*, de acordo com sua capacidade de processamento, consumo de energia e aplicações. Enquanto dispositivos *high-end* são empregados em tarefas complexas e com alta demanda computacional, e *middle-end* oferecem um equilíbrio entre desempenho e eficiência, os *low-end* são projetados para aplicações com recursos limitados, priorizando baixo consumo de energia e simplicidade de operação (Ojo *et al.*, 2018).

Quando se trata de dispositivos *IoT low-end*, grande parte de suas aplicações enfrentam restrições de tempo real, exigindo tempos de execução consistentes para atender às demandas de diversas aplicações (Belleza; Pignaton, 2018). Para lidar com essas limitações, o uso de *RTOS* tem se tornado essencial, oferecendo escalabilidade, segurança, modularidade e maior previsibilidade (Abdelsamea *et al.*, 2016). Além das funcionalidades básicas de um Sistema Operacional (SO), como escalonadores de tarefas e gerenciamento de memória, esses sistemas fornecem capacidades em tempo real, garantindo execução eficiente e multitarefa com preempção, baixa latência e alto desempenho energético (Nikolov *et al.*, 2021).

Além das restrições de tempo, há também, nesses sistemas, limitações de processamento e energia, tornando essencial o melhor aproveitamento dos recursos computacionais. A adoção de arquiteturas *multi-core* tem se mostrado uma solução viável, proporcionando melhor distribuição de carga, e maior eficiência na execução de tarefas (Oliveira *et al.*, 2023). No entanto essa abordagem também impõe uma série de desafios, especialmente relacionados à execução concorrente e ao compartilhamento de recursos, como contenção, sincronização entre tarefas, interferência entre núcleos e impactos na previsibilidade temporal (Oliveira *et al.*, 2023), (Xing *et al.*, 2024).

Embora *RTOS*s sejam amplamente utilizados em dispositivos IoT *low-end*, ainda há pouca clareza sobre como eles realmente aproveitam arquiteturas *multi-core* nesse tipo de sistema. Apesar de alguns SOs voltados para esse contexto implementarem suporte a múltiplos núcleos, eles possuem diferenças em suas abordagens que podem trazer mudanças significativas

em seu desempenho e eficiência. Além disso, faltam estudos que investiguem como o uso de múltiplos núcleos impacta métricas essenciais para esse tipo de sistema, como throughput e consumo de memória.

Diante disso, este trabalho propõe realizar uma análise comparativa do suporte à arquitetura *multi-core* dos *RTOS*s *NuttX* e *FreeRTOS*, utilizando como plataforma de testes a *Raspberry Pi Pico* e avaliando métricas fundamentais para aplicações *IoT*.

1.1 Objetivos

Nesta seção, são apresentados o objetivo geral e os objetivos específicos deste trabalho.

1.1.1 Objetivo Geral

Avaliar o desempenho dos *RTOS*s *NuttX* e *FreeRTOS* no contexto *multi-core*.

1.1.2 Objetivos Específicos

Os objetivos específicos são:

- Implementar uma metodologia de benchmarking capaz de avaliar RTOS em sistemas *single-core* e *multi-core*
- Investigar o impacto do multiprocessamento na execução de tarefas nesses sistemas, considerando desafios como sincronização, chamadas de sistema e contenção de recursos.
- Analisar os mecanismos internos do *kernel* de cada *RTOS* a fim de compreender os efeitos observados na avaliação
- Fornecer insights que auxiliem na escolha da arquitetura e do RTOS mais adequado para diferentes aplicações.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos fundamentais necessários para o desenvolvimento deste trabalho, abordando as tecnologias e metodologias que sustentam a análise proposta. Inicialmente, são explorados os conceitos de *RTOS*, destacando suas características e sua arquitetura. Em seguida, é abordado o conceito de *IoT*, apresentando sua estrutura, camadas e principais aplicações, além da categorização dos dispositivos *IoT* em *low-end*, *middle-end* e *high-end*, conforme suas capacidades computacionais e restrições de *hardware*. Por fim, são discutidos o multiprocessamento e os sistemas *multi-core*, abordando as arquiteturas *SMP* e *Asymmetric Multiprocessing* / Multiprocessamento Assimétrico (*AMP*), suas diferenças e aplicações, bem como o papel dos *RTOS*s em ambientes *multi-core*, considerando como esses sistemas operacionais gerenciam a execução de tarefas e otimizam a comunicação entre núcleos de processamento.

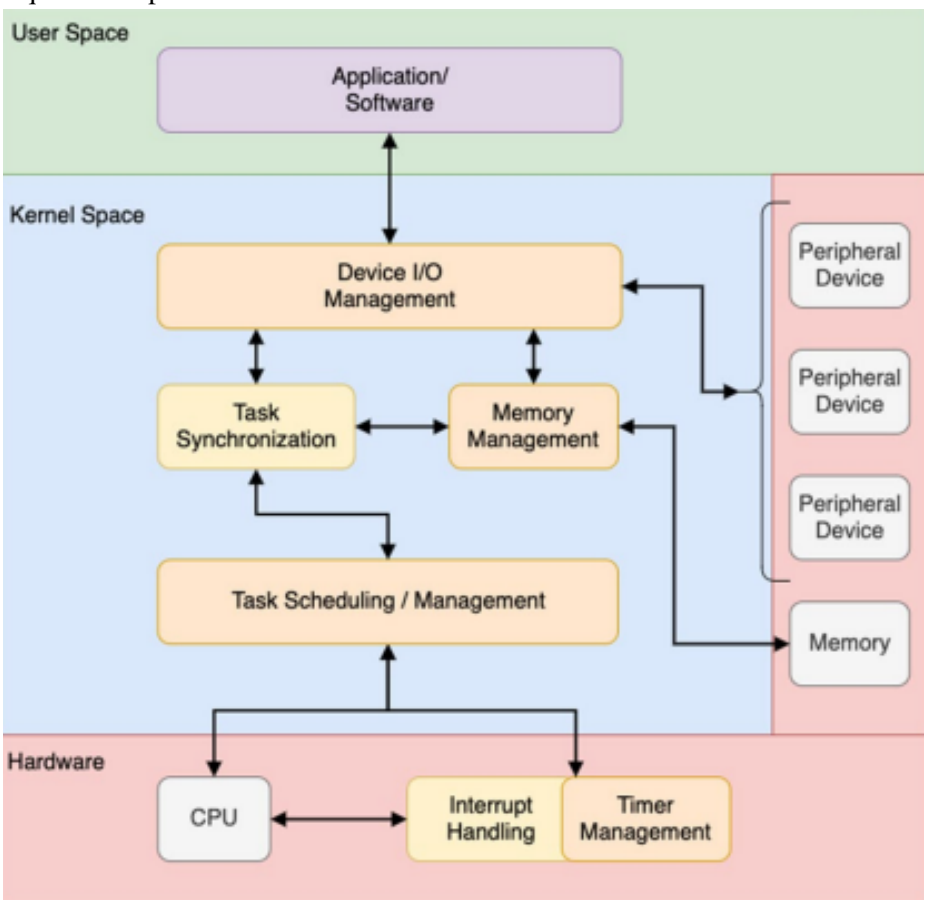
2.1 *RTOS*

O avanço dos sistemas embarcados impulsionou a necessidade de soluções computacionais capazes de gerenciar tarefas com precisão, garantindo que operações críticas sejam executadas dentro de prazos rigorosos. Nesse contexto, surgiram os *RTOS*s, projetados para oferecer previsibilidade e controle rigoroso na execução de processos, assegurando tempos de resposta bem definidos e consistentes (Hambarde *et al.*, 2014). Esses sistemas desempenham um papel crucial em aplicações onde falhas temporais podem comprometer a funcionalidade ou a segurança, como controle de tráfego aéreo, dispositivos médicos, sistemas automotivos e telecomunicações (Reddy *et al.*, 2023).

A arquitetura de um *RTOS* pode ser representada conforme a Figura 1, onde diferentes camadas desempenham funções específicas para garantir a previsibilidade na execução de tarefas. No nível mais básico, encontra-se o *hardware*, que inclui a CPU, o gerenciador de interrupções e os temporizadores, responsáveis por fornecer suporte à execução de operações em tempo real. Acima dessa camada, o *kernel* do *RTOS* gerencia tarefas essenciais, como escalonamento, sincronização e gerenciamento de memória, garantindo que os recursos sejam alocados de forma eficiente. No espaço do usuário, as aplicações interagem com o *kernel* para acessar os serviços do sistema operacional e executar suas funcionalidades específicas. Para que as aplicações na camada do usuário possam interagir com os serviços do *kernel*, os *RTOS*s disponibilizam um conjunto de *Application Programming Interface* / Interface de Programação

da Aplicação (*API*)s, que permitem criar e gerenciar tarefas, sincronizar processos, controlar a temporização e acessar outros serviços essenciais do sistema (Mazzi *et al.*, 2021).

Figura 1 – Arquitetura típica de um *RTOS*



Fonte: (Luna; Islam, 2021, p. 5)

Diferente dos *General-Purpose Operating System* / Sistema Operacional de Propósito Geral (*GPOS*), que são projetados para otimizar o uso dos recursos do sistema e gerenciar múltiplas aplicações simultaneamente, os *RTOS*s priorizam previsibilidade e controle temporal. Para isso, utilizam algoritmos de escalonamento determinísticos, que asseguram que tarefas críticas sejam concluídas sem atrasos imprevisíveis. Já os *GPOS*, como *Linux* e *Windows*, adotam estratégias que favorecem a responsividade e a alocação dinâmica de recursos, o que pode resultar em tempos de resposta variáveis (Atmadja *et al.*, 2014).

2.1.1 Escalonamento

O escalonamento em *RTOS*s é o processo de alocar os recursos da CPU entre as tarefas em execução. Essa função é desempenhada pelo escalonador, que decide quais tarefas devem ser executadas, e pelo dispatcher, responsável por transferir o controle do processador

para a tarefa selecionada (Ismael *et al.*, 2021). A lógica utilizada para essa decisão é definida pelos algoritmos de escalonamento, que organizam a execução das tarefas de acordo com critérios como prioridade e restrições temporais, garantindo que os requisitos de tempo real sejam atendidos e os recursos do sistema sejam gerenciados de maneira eficiente (Abdelsamea *et al.*, 2016).

Esse escalonamento pode ser preemptivo ou não preemptivo. No preemptivo, uma tarefa pode ser interrompida para que outra mais prioritária seja executada, reduzindo tempos de espera. No não preemptivo, a tarefa em execução só libera o processador quando termina, o que simplifica o gerenciamento, mas pode causar atrasos (Abdelhamid; Zamel, 2023). A prioridade das tarefas pode ser estática, onde cada tarefa mantém sempre o mesmo nível, ou dinâmica, onde a prioridade pode mudar durante a execução para otimizar o uso da CPU (Abdelhamid; Zamel, 2023).

2.1.2 Sincronização de tarefas

Quando múltiplas tarefas tentam acessar um mesmo recurso ao mesmo tempo, surgem problemas que podem impactar o tempo de resposta e a previsibilidade do sistema. Se uma tarefa estiver utilizando um recurso enquanto outra aguarda, pode haver atrasos na execução e perda de eficiência (Ungurean; Gaitan, 2018). Em sistemas com múltiplos núcleos, isso é ainda mais crítico, pois diferentes núcleos podem tentar acessar simultaneamente a mesma região de memória, exigindo um controle eficiente para evitar bloqueios (Xing *et al.*, 2024).

Para lidar com esses desafios, os *RTOS*s oferecem mecanismos de sincronização que coordenam a interação entre tarefas e controlam o acesso a recursos compartilhados. Entre os principais métodos, destacam-se semáforos, mutexes, filas de mensagens, eventos e mailboxes (Ungurean; Gaitan, 2018). No contexto *multi-core*, são utilizados mecanismos de bloqueio, conhecidos como locks, sendo o Giant Lock, que aplica um único bloqueio global para todo o sistema, e o Fine-Grained Lock, que utiliza múltiplos bloqueios individuais para cada recurso (Xing *et al.*, 2024).

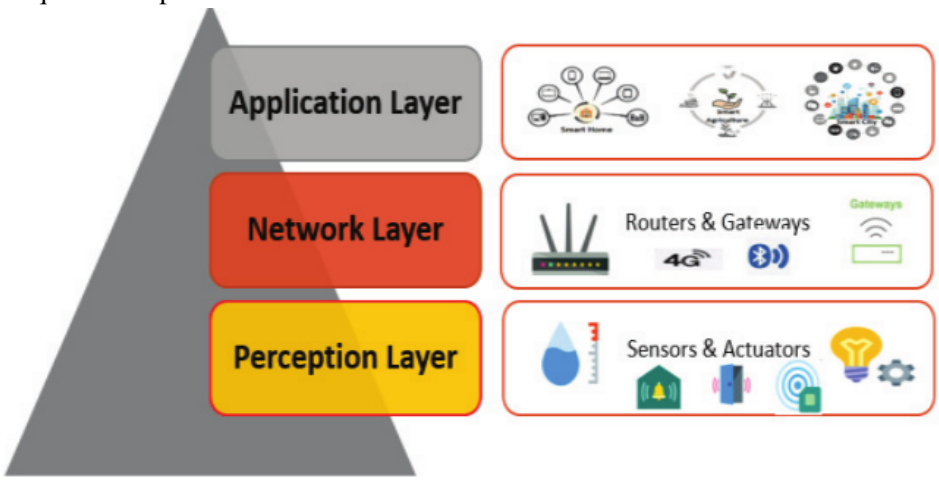
2.2 *IoT*

A *IoT* é um conceito que conecta dispositivos físicos à internet, permitindo a troca e o processamento de informações sem necessidade de intervenção humana. Sua evolução foi

impulsionada pelo avanço das tecnologias sem fio e protocolos de comunicação, não se limitando apenas à conectividade de dispositivos, mas também envolvendo a coleta, análise e utilização de dados para melhorar processos e criar sistemas inteligentes em diversos setores (Paul; Saraswathi, 2017).

A *IoT* é estruturada em diferentes camadas, sendo a arquitetura de três camadas uma das mais comuns, como ilustrado na Figura 2. A camada de percepção é a base do sistema, responsável por coletar informações do ambiente por meio de sensores e atuadores. Os dados captados são enviados à camada de rede, que gerencia a comunicação e a transmissão das informações entre dispositivos, utilizando tecnologias como *Wi-Fi*, redes *LPWAN* e *5G*. Por fim, a camada de aplicação processa os dados e fornece serviços inteligentes aos usuários, como automação residencial, monitoramento de saúde e controle industrial (Choudhary *et al.*, 2021).

Figura 2 – Arquitetura típica da *IoT*



Fonte: (Choudhary et al., 2021, p. 1)

Os dispositivos utilizados em *IoT* podem ser classificados em três categorias principais, conforme suas capacidades computacionais e restrições de *hardware* (Ojo *et al.*, 2018):

– ***Low-end***: operam sob restrições severas de *hardware*, possuindo capacidade de processamento, memória e armazenamento extremamente limitados. Devido a essas limitações, esses dispositivos geralmente não conseguem executar sistemas operacionais complexos, como *Linux* ou *Windows IoT Core*, e dependem de *firmware* especializado ou sistemas operacionais minimalistas para seu funcionamento. Eles são amplamente utilizados em aplicações onde a eficiência energética é essencial, como redes de sensores sem fio ou dispositivos vestíveis (Hahm *et al.*, 2016).

  Para padronizar a categorização desses dispositivos, foi estabelecida uma classificação

que os divide em três subclasses, de acordo com a capacidade de memória RAM e armazenamento *Flash* (Bormann *et al.*, 2014). Essa classificação é apresentada no quadro 1.

Quadro 1 – Classificação de dispositivos *IoT low-end*

Classe	Memória RAM	Memória *Flash*
C0	Muito abaixo de 10 KiB	Muito abaixo de 100 KiB
C1	Aproximadamente 10 KiB	Aproximadamente 100 KiB
C2	Aproximadamente 50 KiB	Aproximadamente 250 KiB

Fonte: elaborado pelo autor.

- ***Middle-end***: possuem um equilíbrio entre desempenho e eficiência energética, situando-se entre os dispositivos *low-end* e os *high-end*. Esses dispositivos possuem processadores de 32 bits, memória RAM variando de centenas de KB a poucos MB e maior flexibilidade na comunicação, integrando tecnologias como *Wi-Fi*, *Bluetooth*, *Zigbee* e *LTE-M*. Além disso, apresentam um número maior de interfaces de *hardware*, como *GPIOs*, *SPI*, *I2C* e *UART*, possibilitando uma conectividade mais abrangente com sensores e atuadores.
- ***High-end***: possuem capacidade computacional significativamente superior, permitindo a execução de sistemas operacionais complexos. Esses dispositivos contam com processadores *multi-core* de alto desempenho, memória RAM que pode ultrapassar centenas de MB e suporte a tecnologias avançadas de comunicação e segurança. Sua capacidade de processamento possibilita a execução de aplicações complexas, como análise de dados em tempo real, inteligência artificial embarcada e controle avançado de sistemas industriais. São amplamente utilizados em *gateways IoT*, veículos autônomos, automação residencial sofisticada e infraestrutura de cidades inteligentes, onde a necessidade de alto poder computacional e conectividade robusta é essencial.

As aplicações da *IoT* têm crescido significativamente, abrangendo setores como saúde, automação industrial, cidades inteligentes e agricultura de precisão (Choudhary *et al.*, 2021). Grande parte dessas aplicações utiliza dispositivos *IoT low-end* (Baccelli *et al.*, 2018), muitos dos quais exigem operação em tempo real. Nesses casos, o uso de sistemas operacionais otimizados é fundamental para garantir previsibilidade na execução de tarefas e o gerenciamento eficiente de recursos (Silva *et al.*, 2019). Para atender a essas exigências, esses dispositivos frequentemente adotam um *RTOS*, garantindo confiabilidade mesmo em ambientes com restrições computacionais.

Um *RTOS* para essas aplicações deve ser eficiente em energia, integrar *stacks* de

rede leves (como IEEE 802.15.4, *BLE* e *6LoWPAN*) e garantir modularidade para economizar recursos. Ferramentas como gerenciamento de tarefas, escalonamento e abstração de *hardware* são essenciais para atender demandas de tempo real. Além disso, é crucial oferecer suporte à segurança com bibliotecas de criptografia para proteger dados e privacidade em redes distribuídas (Hahm *et al.*, 2016). Entre os principais *RTOS*s utilizados em dispositivos *IoT* estão FreeRTOS, NuttX, Zephyr, Contiki-NG, Mbed OS, TinyOS e RIOT, amplamente utilizados devido à flexibilidade e ao suporte a requisitos críticos de sistemas conectados (Kanwal *et al.*, 2023).

2.3 Multiprocessamento e sistemas *multi-core*

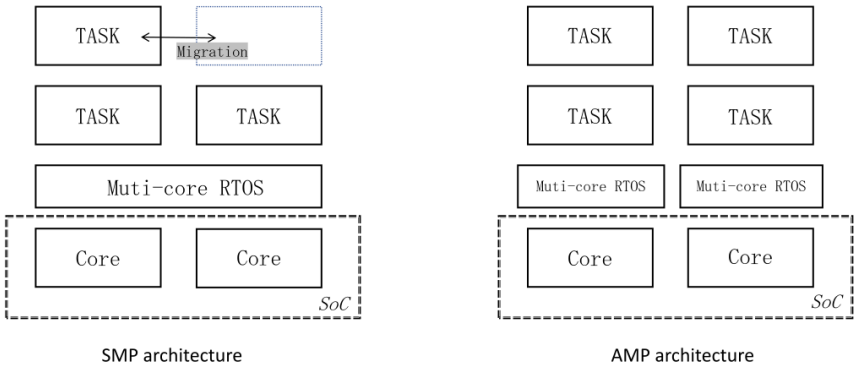
O aumento da demanda por capacidade de processamento impulsionou a evolução dos sistemas computacionais, levando ao desenvolvimento de arquiteturas mais eficientes para lidar com cargas de trabalho complexas. Com a necessidade de maior desempenho e melhor aproveitamento dos recursos computacionais, surgiram os sistemas multiprocessados, que se destacam por permitir a execução simultânea de tarefas, otimizando a utilização do *hardware* disponível. O multiprocessamento é uma abordagem que possibilita o uso de múltiplos processadores ou núcleos para executar processos em paralelo, reduzindo o tempo de execução das tarefas e aumentando a eficiência do sistema (Bueno, 2007).

O multiprocessamento pode ser classificado em duas principais abordagens: *SMP* e *AMP*. No *SMP*, todos os processadores ou núcleos são idênticos, compartilham uma única memória principal e executam tarefas de forma equilibrada (Hanafi *et al.*, 2020). Por outro lado, no *AMP*, os processadores podem ter diferentes características, desempenhando funções específicas dentro do sistema. Normalmente, um processador *master* gerencia a execução das tarefas e delega processos a processadores *slave*, balanceando tarefas entre núcleos de alto desempenho e núcleos de baixo consumo energético (Martos; Garrido, 2017).

As técnicas de multiprocessamento impulsionaram o surgimento de sistemas *multi-core*, que integram múltiplos núcleos em um único chip, oferecendo maior desempenho e eficiência energética. Esses sistemas permitem a execução simultânea de múltiplas tarefas, otimizando o uso de recursos computacionais e reduzindo o consumo de energia, especialmente em aplicações modernas que demandam alto desempenho. No entanto, a crescente complexidade dessas arquiteturas exige soluções avançadas de gerenciamento, como os *RTOS*s *multi-core*, que desempenham um papel crucial ao lidar com o escalonamento de tarefas, a comunicação entre núcleos e o balanceamento de carga. A Figura 3 ilustra como as arquiteturas *SMP* e *AMP* são

aplicadas no contexto de *RTOS multi-core*. No *SMP*, um único *RTOS* gerencia todos os núcleos, permitindo a migração dinâmica de tarefas para balancear a carga de trabalho. Já no *AMP*, cada núcleo opera com um *RTOS* independente, simplificando o gerenciamento, mas limitando a flexibilidade do sistema (Xing *et al.*, 2024).

Figura 3 – Arquiteturas *SMP* e *AMP* em *RTOS multi-core*



Fonte: (Xing et al., 2024, p. 3)

3 TRABALHOS RELACIONADOS

Este capítulo apresenta uma discussão detalhada dos trabalhos relacionados que fundamentam este estudo. Os artigos analisados oferecem diferentes perspectivas sobre *benchmarking* e desempenho de *RTOS*s abordando aplicações em dispositivos IoT e plataformas *multicore*. A seguir, é apresentada uma análise de três trabalhos principais que contribuem para a compreensão do tema e servem como base para a comparação com o escopo deste trabalho.

3.1 Operating Systems for Internet of Things Low-End Devices: Analysis and *benchmarking*

O trabalho de (Silva *et al.*, 2019) busca compreender as características e desafios dos sistemas operacionais em dispositivos *IoT low-end*. O objetivo é realizar uma análise comparativa detalhada entre os principais *RTOS*s utilizados nesse contexto, avaliando suas capacidades e limitações em diferentes cenários de uso para orientar escolhas em aplicações *IoT*.

Inicialmente, o artigo conduz uma análise do ecossistema *IoT*, identificando métricas cruciais como consumo de energia, *footprint* de memória e suporte a operações em tempo real. Além disso, identifica os *RTOS*s mais utilizados nesse contexto, e seleciona três (RIOT, Contiki-NG e Zephyr) para conduzir um estudo comparativo com base nas métricas identificadas.

Para avaliar os sistemas, o autor utilizou o *Thread-Metric Benchmark Suite*, que define um total de oito testes que têm como objetivo avaliar os serviços mais comuns de *RTOS*s e seus mecanismos de processamento e interrupção, sendo eles:

- **Processamento Básico:** avalia o desempenho base de execução de um único *thread*, servindo como referência para os demais testes.
- **Escalonamento Cooperativo:** avalia o custo de trocas de contexto em escalonamento cooperativo entre *threads* de mesma prioridade.
- **Escalonamento Preemptivo:** avalia o custo de trocas de contexto baseadas em prioridade, considerando preempção entre *threads*.
- **Processamento de Interrupções:** avalia a latência e o custo do atendimento a interrupções sem ocorrência de preempção.
- **Processamento de Interrupções com Preempção:** avalia o impacto do atendimento a interrupções quando há preempção de *threads* de menor prioridade.
- **Comunicação por Mensagens:** avalia o custo dos mecanismos de comunicação entre

threads por meio de filas de mensagens.

- **Sincronização por Semáforos:** avalia o custo das operações de sincronização baseadas em semáforos.
- **Gerenciamento Dinâmico de Memória:** avalia o custo das operações de alocação e desalocação dinâmica de memória.

Em cada um dos cenários, os sistemas foram avaliados sob diferentes estados da rede: *idle* (sem tráfego), aceitação (pacotes processados pelo nó), rejeição (pacotes descartados) e encaminhados (pacotes retransmitidos para outros nós).

Com essa avaliação, o artigo concluiu que o Contiki-NG apresentou a melhor eficiência energética e *footprint* de memória reduzido, sendo ideal para dispositivos extremamente limitados onde o consumo de energia é prioritário. O RIOT demonstrou bom equilíbrio entre desempenho e previsibilidade em tempo real, tornando-se uma escolha robusta para aplicações críticas. Já o Zephyr, apesar de consumir mais recursos, destacou-se pela flexibilidade e suporte a arquiteturas modernas como RISC-V, sendo adequado para aplicações mais complexas.

Embora a metodologia apresentada avalie métricas relevantes e forneça uma visão abrangente sobre o desempenho dos *RTOS*s, ela enfatiza o impacto da conectividade e a interação dos sistemas operacionais com a pilha de rede, se aprofundando pouco nas características de tempo real de cada sistema, além de se limitar a abordagem *single-core*.

3.2 *Performance study of real-time operating systems for Internet of Things devices*

O trabalho de (Belleza; Pignaton, 2018) analisa os desafios enfrentados por dispositivos *IoT* de baixa capacidade e busca avaliar o desempenho *RTOS*s amplamente utilizados nesse contexto. A finalidade é entender como esses *RTOS*s lidam com propriedades de tempo real essenciais, como primitivas de sincronização, resposta a eventos externos e desempenho geral. Oferecendo subsídios para a escolha de soluções otimizadas em aplicações *IoT*.

O estudo inclui um levantamento detalhado dos *RTOS*s disponíveis no mercado, avaliando critérios como popularidade, licenciamento de código aberto e suporte direto à plataforma de testes utilizada. Selecionou-se, a partir dessa análise, cinco *RTOS*s para avaliação: FreeRTOS, RIOT, Zephyr, µC/OS-II e µC/OS-III. Que foram analisados quanto a tempos de troca de contexto, manipulação de semáforos, transferência de mensagens, alocação de memória e ativação de tarefas a partir de interrupções.

Para realizar a avaliação, os autores configuraram uma plataforma NXP/Freescale

FRDM-K64F, escolhida por seu suporte direto a todos os *RTOS*s selecionados. Os *benchmarks* utilizados foram projetados para simular cenários práticos, como sincronização de tarefas e resposta a eventos externos, permitindo medir aspectos específicos, como tempos de troca de contexto e manipulação de semáforos. As métricas também incluíram testes para alocação de memória e transferência de mensagens, tudo com o objetivo de avaliar o comportamento dos sistemas sob condições reais de uso, como cargas intensas ou eventos inesperados em dispositivos *IoT*.

Os resultados indicam que o RIOT apresentou consistência em diversos cenários, enquanto o FreeRTOS demonstrou flexibilidade e bom desempenho em sincronização de tarefas. Por outro lado, o Zephyr apresentou maior latência em algumas operações devido à sua arquitetura mais complexa. Os *RTOS*s µC/OS-II e µC/OS-III destacaram-se pela previsibilidade e certificações de segurança, tornando-os mais adequados para aplicações críticas.

Embora o artigo apresente um estudo rico sobre *RTOS*s voltados ao *IoT* e avalie métricas fundamentais para o desempenho em tempo real, essa avaliação não é feita de forma tão aprofundada, limitando-se a executar testes apenas de primitivas simples como captura e liberação de semáforos. Além disso, ele não inclui o impacto de uma arquitetura *multicore* em seu estudo.

3.3 *A Multi-core RTOS Benchmark Methodology To Assess System Services Under Contentions*

O trabalho de (Xing *et al.*, 2024) propõe uma metodologia de *benchmarking* para *RTOS*s em plataformas *multicore*, com o objetivo de superar as limitações de ferramentas existentes que não avaliam adequadamente os impactos das contenções de recursos e os diferentes designs de *locks* em sistemas *multicore*. O foco é fornecer *insights* sobre como os serviços de um *RTOS* são influenciados por tais fatores, abordando questões como escalabilidade, previsibilidade e eficiência em contextos complexos.

Inicialmente o autor fez uma análise Aprofundada do *Kernel* do *TOPPERS/FMP* , agrupando suas *APIs* conforme a sequência e o tipo de recurso interno acessado, conforme mostra o quadro 2. *APIs* que compartilham o mesmo fluxo de acesso apresentam comportamento equivalente em termos de contenção e locking, permitindo que um único representante seja utilizado no benchmark.

A metodologia inclui dois componentes principais: o *"stress workload benchmark"*

Quadro 2 – Padrões de acesso às APIs do kernel TOPPERS/FMP e APIs representativas

Padrão	**Sequência de acesso aos recursos**	**API de exemplo**
Pattern 1	Objeto	`ref_sem`, `pol_sem`, `ref_flg`
Pattern 2	Objeto → tarefa (própria)	`wai_sem`, `twai_sem`, `wai_flg`
Pattern 3	Objeto → tarefa (outra)	`sig_sem`, `snd_mbx`, `psnd_dtq`
Pattern 4	Objeto → tarefa (própria ou outra)	`snd_dtq`, `rcv_dtq`, `tsnd_pdq`
Pattern 5	Objeto → múltiplas tarefas	`ini_sem`, `set_flg`, `ini_dtq`
Pattern 6	Tarefa	`get_pri`, `slp_tsk`, `wup_tsk`
Pattern 7	Tarefa → tarefa	`mig_tsk`, `mact_tsk`
Pattern 8	Tarefa → objeto	`chg_pri`, `rel_wai`

Fonte: (Xing *et al.*, 2024)

e o *"service throughput benchmark"*. O primeiro visa avaliar a latência e o tempo de execução de serviços do *RTOS* sob diferentes níveis de contenção, enquanto o segundo mede escalabilidade e equidade na distribuição de recursos entre núcleos. Os *benchmarks* foram implementados em uma plataforma experimental composta pelo kernel TOPPERS/FMP e a placa Xilinx ZCU-104, escolhida por sua arquitetura de 4 núcleos e suporte a diferentes designs de *locks*.

Para implementar os *benchmarks*, foram criados cenários que simulam diferentes padrões de acesso a recursos, utilizando combinações de tarefas de avaliação e tarefas geradoras de estresse. O tempo de execução de cada *API* foi medido, assim como eventos de *hardware* relacionados à contagem de acesso a memória, barramentos e cache L2. Esses dados foram analisados para identificar como os diferentes designs de *locks* - como *giant lock* e *fine-grained lock* – afetam o desempenho em condições variadas de carga e paralelismo.

Os resultados mostraram que os *fine-grained locks* apresentam vantagens significativas em termos de paralelismo e escalabilidade, especialmente em cenários com altos níveis de contenção. No entanto, também revelaram limitações, como maior sobrecarga em operações que envolvem múltiplos recursos, o que pode impactar a previsibilidade em aplicações críticas. Por outro lado, os *giant locks* demonstraram melhor desempenho em casos de migração de tarefas entre núcleos devido ao menor custo de manutenção de *locks*.

O artigo propõe uma metodologia bastante eficaz na análise de um *RTOS* em um sistema *multicore*, porém ele não traz uma comparação direta entre diferentes *RTOS*s, e mantém seu foco em dispositivos *high-end*, não abordando os impactos do *multicore* em sistemas com maior limitação de recursos.

3.4 Análise Comparativa

Os trabalhos relacionados discutidos apresentam diferentes abordagens para a avaliação de *RTOS*s, abordando desde dispositivos *IoT low-end* até plataformas *multicore* com foco em contenções e escalabilidade. Essa diversidade oferece uma base sólida para compreensão das limitações e potencialidades no desenvolvimento e *benchmarking* de *RTOS*s.

A Figura 4 apresenta uma comparação geral do que foi abordado nos trabalhos, destacando as diferenças nos tipos de comparação, metodologias de *benchmarking* e métricas utilizadas

Figura 4 – Comparação entre os trabalhos relacionados.

Trabalho	Tipo de Comparação	Metodologia de Benchmarking		Métricas Avaliadas			
		Primitivas Isoladas	Simulação de Cenário Real	Tempo Real	Uso de recursos do sistema	Multicore	Conectividade
(Silva, et al., 2019)	Entre RTOSs	✓	✓	✓	✓		✓
(Belleza; Pignaton, 2018)	Entre RTOSs	✓		✓			
(Xing et al., 2024)	Entre Locks	✓		✓		✓	
Este trabalho	Entre RTOSs	✓	✓	✓	✓	✓	

Fonte: Elaborado pelo autor.

4 METODOLOGIA

Este capítulo descreve a metodologia adotada para a condução dos experimentos deste trabalho. As etapas foram organizadas para garantir reprodutibilidade e comparabilidade dos resultados, desde a definição da plataforma de *hardware* e dos *RTOS*s até a execução dos testes e a análise dos dados coletados.

Quadro 3 – Etapas da metodologia adotada neste trabalho.

Etapa	Descrição
Revisão da literatura e definições preliminares	Definição da plataforma de *hardware* e dos *RTOS*s avaliados.
Cenários e métricas	Definição dos cenários de teste e das métricas utilizadas em cada um.
Configuração experimental	Configuração da placa, coleta via UART e sincronização das tarefas.
Execução dos experimentos	Execução dos cenários em *single-core* e *multi-core*.
Coleta e análise dos dados	Cálculo das métricas, normalizações e organização dos resultados.

Fonte: elaborado pelo autor.

4.1 Revisão da literatura e definições preliminares

A revisão da literatura foi conduzida para definir os principais elementos do estudo, incluindo os *RTOS*s a serem avaliados e a plataforma de *hardware* utilizada.

4.1.1 Plataforma de hardware

Para a escolha da plataforma de *hardware* foram considerados, principalmente, a presença de múltiplos núcleos e especificações compatíveis com dispositivos *low-end*. Nesse contexto, optou-se pela placa *Raspberry Pi Pico*, baseada no microcontrolador RP2040, que integra dois núcleos ARM Cortex-M0+ operando de forma simétrica.

A ausência de memória cache contribue para um ambiente previsível, pois reduz a variabilidade de tempo associada a *hits/misses*, enquanto a *SRAM* compartilhada permite que variáveis globais e estruturas do *RTOS* permanecem visíveis a ambos, permitindo observar com mais clareza o custo de seções críticas e mecanismos de sincronização. Além disso, o microcontrolador disponibiliza temporizadores e controladores de interrupção, Recursos

Figura 5 – Plataforma de desenvolvimento: *Raspberry Pi Pico*.

Fonte: raspberrypi.com/products/raspberry-pi-pico/.

importantes, que foram utilizados e avaliados nos casos de teste. O baixo custo e a ampla disponibilidade da placa também facilitaram a execução e repetição dos testes.

4.1.2 RTOSs

Para a seleção dos *RTOS*s, foi realizada uma revisão da literatura considerando apenas projetos de código aberto. Adicionalmente, foi exigido suporte à arquitetura *SMP* na plataforma de *hardware* adotada. Também foram considerados: documentação técnica, adoção comercial (especialmente em aplicações de *IoT*) e comunidade ativa. Assim, os *RTOS*s escolhidos foram *FreeRTOS* e *NuttX*.

4.1.2.1 FreeRTOS

O *FreeRTOS* é amplamente utilizado em sistemas embarcados e aplicações de *IoT*, destacando-se por simplicidade e baixo consumo de recursos. Possui documentação consolidada e ampla adoção, o que facilita configuração, instrumentação e reprodução dos testes.

4.1.2.2 NuttX

O *NuttX* é um *RTOS* de código aberto com arquitetura mais completa e suporte a uma interface *POSIX-like*, além de amplo suporte a plataformas embarcadas. Também apresenta suporte à *SMP* no RP2040, permitindo avaliar a execução *multi-core* em um sistema com serviços mais abrangentes.

4.2 Cenários de teste e métricas associadas

Os cenários e métricas foram definidos de forma integrada, de modo que cada cenário explore uma dimensão do comportamento dos *RTOS*s em execução *single-core* e *multi-core*.

O primeiro cenário é baseado no *Thread-Metric Benchmark Suite*, com foco em cargas compostas. Nesse caso, a métrica principal é o *throughput* bruto, definido como a quantidade de iterações executadas em uma janela temporal fixa. Como métrica complementar, foi monitorado o consumo de memória durante a execução.

O segundo cenário avalia escalabilidade com primitivas isoladas, agrupadas por padrões de acesso a *API*s descritos nos trabalhos relacionados. A métrica utilizada é o *throughput* normalizado, no qual o desempenho em *single-core* é a referência (100%).

O terceiro cenário avalia contenção de recursos, medindo como o *throughput* se degrada quando múltiplas tarefas acessam o mesmo recurso compartilhado. Nesse caso, utiliza-se novamente *throughput* normalizado, adotando como referência a execução sem contenção induzida.

4.3 Configuração experimental

Os experimentos foram executados na *Raspberry Pi Pico*, com coleta de resultados via UART0. A frequência de operação do microcontrolador foi mantida em 125 MHz ao longo de todos os testes. No *NuttX*, a UART também foi utilizada para interação com o *NuttShell*, que é um recurso nativo desse sistema, sendo útil para facilitar a execução dos testes.

Para garantir comparabilidade entre execuções e evitar vieses causados por inicialização assíncrona, o experimento foi estruturado em duas etapas: (i) sincronização do instante de início e (ii) definição de uma janela fixa de execução. Essas duas funções são implementadas, respectivamente, por um mecanismo de barreira e por uma tarefa de controle temporal.

O mecanismo de barreira, mostrado na Listagem 1, funciona como um ponto de encontro: cada tarefa participante (incluindo a tarefa de temporização) chama `barrierwait()` antes de iniciar seu laço principal. A variável `g_barrier_count` contabiliza quantas tarefas já alcançaram esse ponto; quando o total esperado é atingido, a flag `g_barrier_go` é liberada e todas as tarefas prosseguem em conjunto. Enquanto a barreira não é liberada, as tarefas permanecem em espera, cedendo processador com `taskYIELD()`.

A janela de execução é controlada por uma tarefa dedicada, denominada *TimerCon-*

```
void barrierwait(void) {
    taskENTER_CRITICAL();
    g_barrier_count++;
    if (g_barrier_count >= (TASKS_PER_CORE*NUM_OF_CORES+1))
        {
        g_barrier_go = 1;
    }
    taskEXIT_CRITICAL();

    while (!g_barrier_go) {
        taskYIELD();
    }
}
```

Código-fonte 1 – Mecanismo de barreira (`barrierwait`).

trolTask, apresentada na Listagem 2. Essa tarefa também participa da barreira para assegurar que a contagem de tempo só se inicie após todas as tarefas estarem prontas. Em seguida, ela aguarda o intervalo predefinido (`EXECUTION_LOOP_SECONDS`) e, ao final, sinaliza o encerramento do experimento por meio de uma variável global (`run`). As demais tarefas monitoram essa flag e interrompem seus laços quando `run` torna-se `false`, finalizando a execução de forma consistente entre os diferentes sistemas avaliados.

```
void TimerControlTask(void *pvParameters) {
    barrierwait();
    vTaskDelay(pdMS_TO_TICKS(EXECUTION_LOOP_SECONDS * 1000)
       );
    run = false;
    vTaskDelete(NULL);
}
```

Código-fonte 2 – Tarefa de controle temporal (*TimerControlTask*).

4.4 Execução dos experimentos

Esta seção descreve a execução dos três cenários de teste. Em todos os casos, foram consideradas execuções em modo *single-core* e *multi-core*, mantendo a mesma janela temporal e os mesmos mecanismos de sincronização.

4.4.1 Primeiro cenário

Este cenário foi importante para trazer uma avaliação integrada dos serviços de cada *kernel*. Para este experimento, foi desenvolvido um algoritmo para representar cada um dos oito padrões de execução do *Thread-Metric Benchmark Suite*. Cada padrão foi executado em ambos os *RTOS*s, em *single-core* e *multi-core*. Os padrões foram executados em laço por uma janela fixa de 30 s; a cada iteração completa, um contador era incrementado para cálculo do *throughput*. Para reduzir interferências, os recursos utilizados pelas tarefas foram mantidos locais por tarefa/núcleo sempre que possível, evitando contenção explícita neste cenário.

4.4.1.1 Padrão T1 — Processamento básico

É criada uma tarefa por núcleo, que executa operações aritméticas/lógicas em loop e incrementa o contador por iteração. O diagrama da Figura 6 descreve o fluxo de uma tarefa nesse padrão.

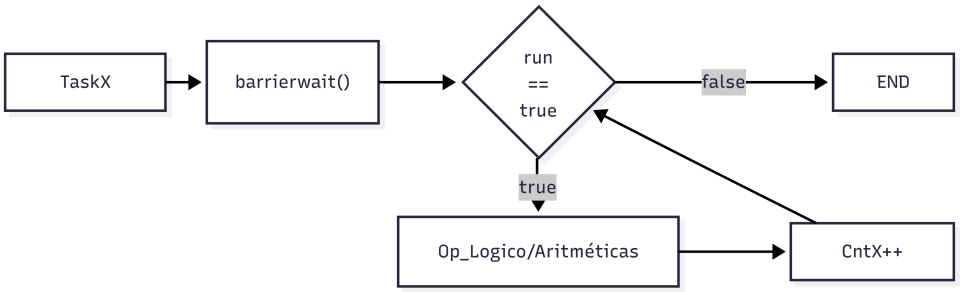


Figura 6 – Diagrama do padrão T1 — Processamento básico.

4.4.1.2 Padrão T2 — Escalonamento cooperativo

São criadas cinco tarefas por núcleo, todas com mesma prioridade. Cada tarefa incrementa um contador e, em seguida, executa *yield*, liberando o processador para a próxima tarefa. Esse processo ocorre de forma cíclica até que o processador retorne à primeira tarefa, repetindo-se continuamente. O diagrama da Figura 7 descreve o fluxo de uma tarefa nesse padrão.

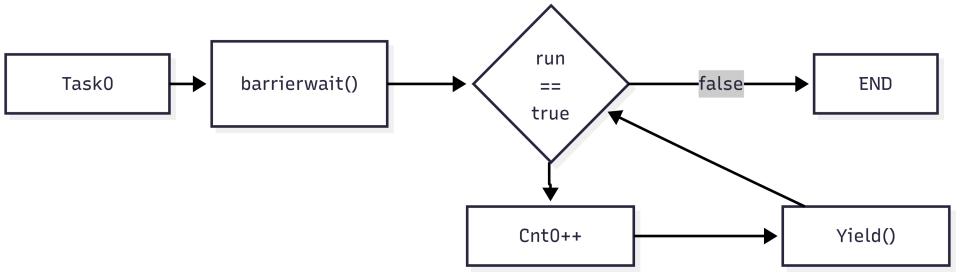


Figura 7 – Diagrama do padrão T2 — Escalonamento cooperativo.

4.4.1.3 Padrão T3 — Escalonamento preemptivo

São criadas cinco tarefas por núcleo, todas com prioridades diferentes. As tarefas iniciam entrando em bloqueio por meio de *ulTaskNotifyTake*, aguardando uma notificação. Após receberem a notificação, incrementam um contador e sinalizam a próxima tarefa de maior prioridade por meio de *xTaskNotifyGive*. Como a tarefa seguinte possui prioridade superior, ocorre preempção. Nesse cenário, a tarefa de *timer* é utilizada também para liberar a notificação da primeira tarefa nesse padrão. O diagrama da Figura 8 descreve o fluxo de execução das tarefas e da tarefa de *timer* nesse padrão.

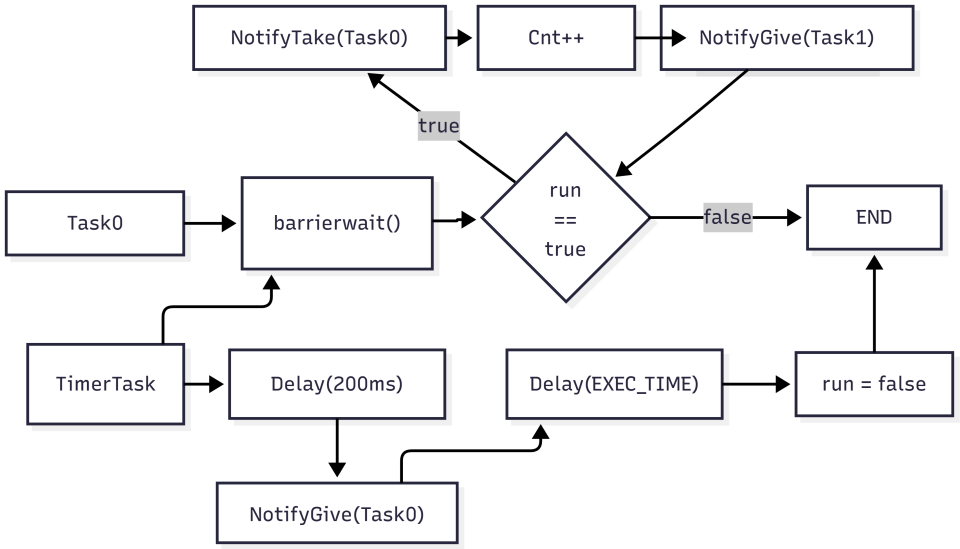


Figura 8 – Diagrama do padrão T3 — Escalonamento preemptivo.

4.4.1.4 Padrão T4 — Processamento por interrupções

É criada uma tarefa e instanciada uma rotina de interrupção por núcleo, juntamente com um semáforo associado a cada um. Em cada tarefa, um semáforo binário é criado e definido inicialmente como indisponível. A rotina de interrupção é configurada para liberar esse semáforo

por meio de uma interrupção (*xSemaphoreGiveFromISR*). No laço principal, a interrupção é explicitamente definida como pendente (*irq_set_pending*), e a tarefa tenta capturar o semáforo, permanecendo bloqueada até que o tratamento da interrupção ocorra. Após a liberação do semáforo, o contador é incrementado. O diagrama da Figura 9 descreve o fluxo desse padrão.

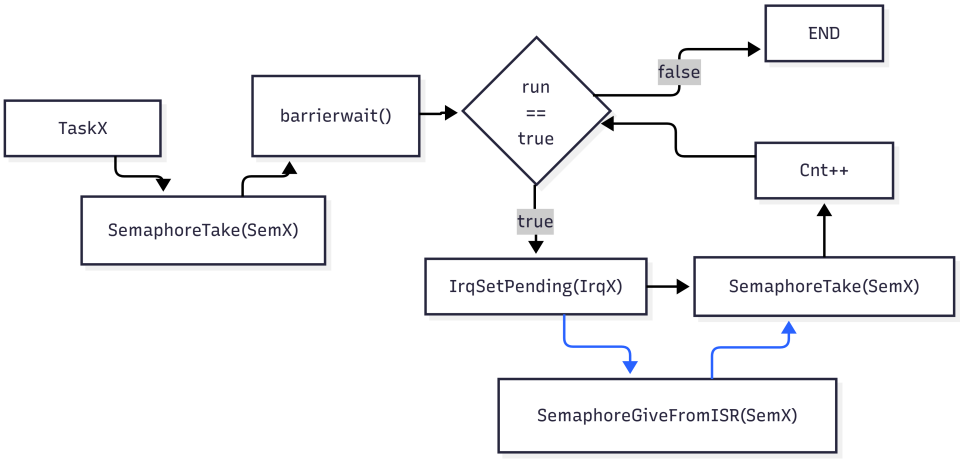


Figura 9 – Diagrama do padrão T4 — Processamento por interrupções.

4.4.1.5 Padrão T5 — Preempção induzida por interrupção

São criadas duas tarefas e uma rotina de interrupção por núcleo. A tarefa 0 possui maior prioridade e é responsável por incrementar um contador e, em seguida, se suspender (*vTaskSuspend*). A rotina de interrupção é configurada para retomar essa tarefa por meio de *xTaskResumeFromISR*. A tarefa 1, de menor prioridade, é responsável por definir a interrupção como pendente. Como a tarefa 0 possui prioridade superior à tarefa 1, sua retomada a partir da interrupção provoca preempção. O diagrama da Figura 10 ilustra o fluxo desse padrão.

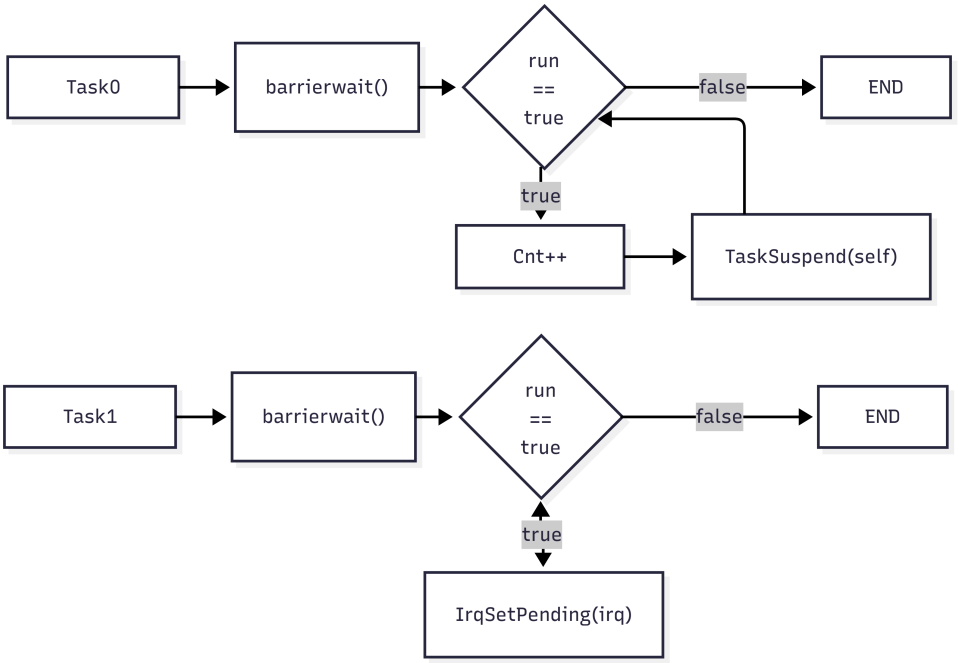


Figura 10 – Diagrama do padrão T5 — Preempção induzida por interrupção.

4.4.1.6 Padrão T6 — Processamento de mensagens

É criada uma tarefa por núcleo, que executa operações de envio e recebimento de mensagens por meio de filas (*queue send/receive*) em loop, com incremento do contador a cada iteração. O diagrama da Figura 11 descreve o fluxo de uma tarefa nesse padrão.

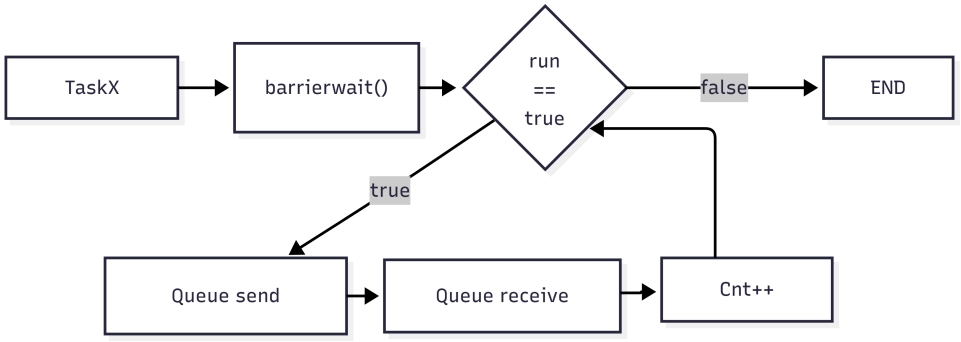


Figura 11 – Diagrama do padrão T6 — Processamento de mensagens.

4.4.1.7 Padrão T7 — Sincronização

É criada uma tarefa por núcleo, que executa operações de aquisição e liberação de mecanismos de sincronização (*semaphore take/give*) em loop, com incremento do contador por iteração. O diagrama da Figura 12 descreve o fluxo de uma tarefa nesse padrão.

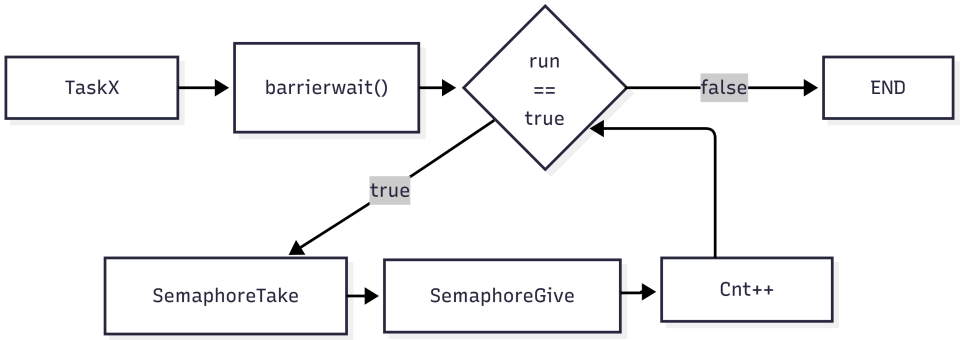


Figura 12 – Diagrama do padrão T7 — Sincronização.

4.4.1.8 Padrão T8 — Alocação dinâmica de memória

É criada uma tarefa por núcleo, que executa operações de alocação e liberação de memória (*malloc/free*) em loop, com incremento do contador a cada iteração. O diagrama da Figura 13 descreve o fluxo de uma tarefa nesse padrão.

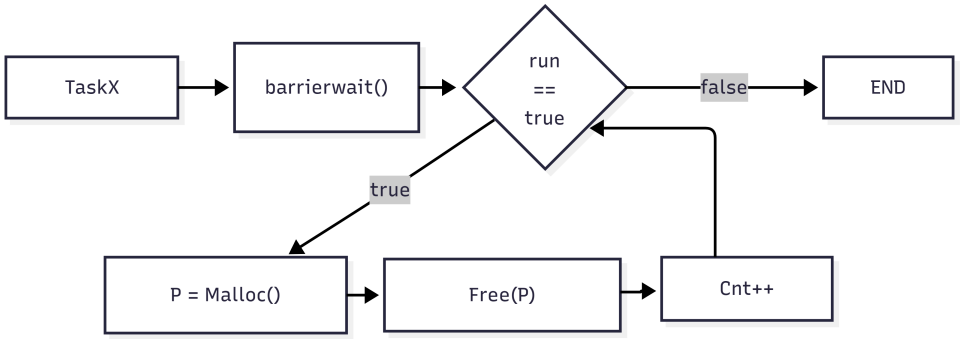


Figura 13 – Diagrama do padrão T8 — Alocação dinâmica de memória.

4.4.2 Segundo Cenário

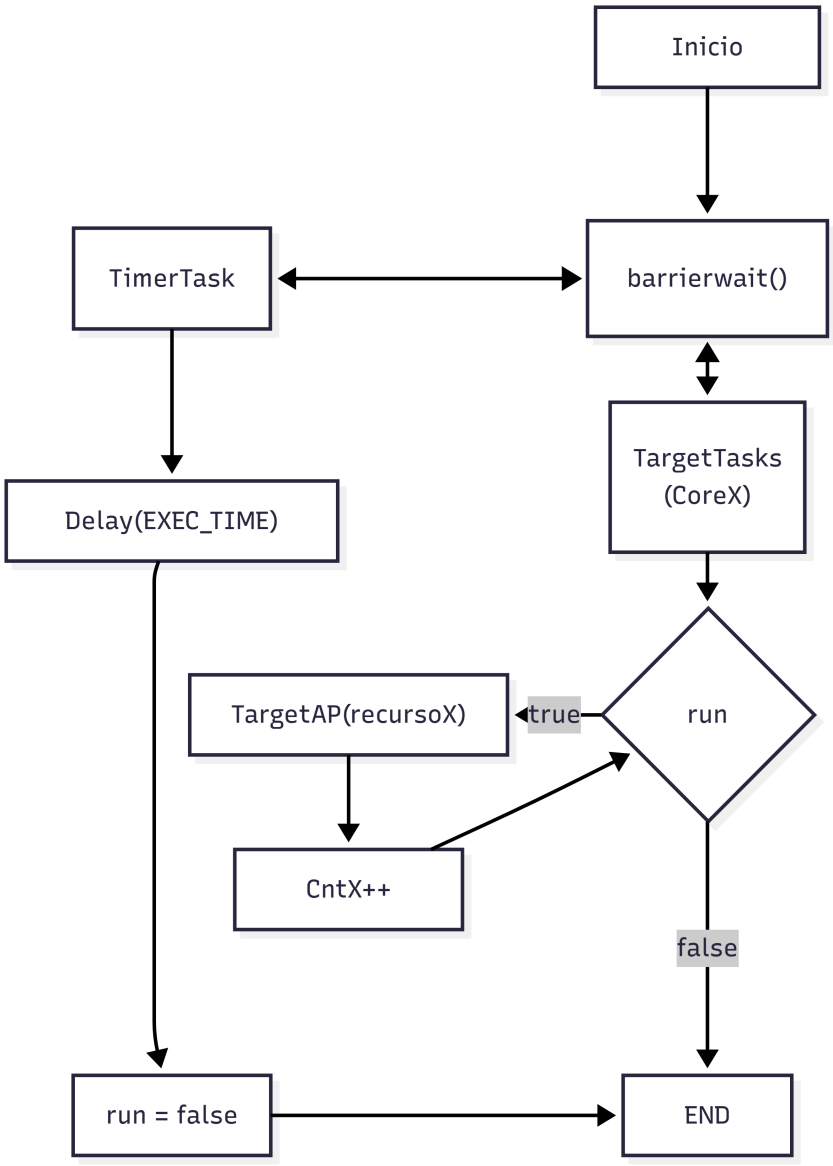
Este Cenário trouxe uma avaliação de escalabilidade de primitivas de *kernel*. Os padrões de acesso a APIs de *RTOS*s adotados neste trabalho foram baseados na taxonomia proposta por (Xing *et al.*, 2024), que define oito padrões de interação entre tarefas e recursos do kernel (Tabela **??**). Como o escopo desta avaliação concentra-se em primitivas comuns e diretamente comparáveis entre *FreeRTOS* e *NuttX*, foram instanciados apenas os seis primeiros padrões dessa taxonomia (Padrões 1–6). A Tabela **??** apresenta a correspondência entre cada padrão e as primitivas representativas selecionadas em cada sistema.

O experimento de escalabilidade consiste em executar essas primitivas sob uma carga paralela controlada, conforme o fluxo da Figura 14. Em cada configuração, é criada uma tarefa de benchmark por núcleo ativo. Cada tarefa executa, em laço, a primitiva associada ao padrão

em teste e incrementa um contador local a cada iteração. Ao final de uma janela fixa de tempo, o valor do contador é coletado via UART e utilizado para calcular o *throughput* (operações por segundo).

Para isolar o custo da chamada de API e reduzir interferências externas, as execuções foram realizadas sem compartilhamento de recursos entre núcleos: cada tarefa opera sobre instâncias independentes dos objetos do kernel (por exemplo, semáforos, filas ou estruturas equivalentes), evitando contenção e bloqueios por disputa.

Figura 14 – Fluxo do experimento de escalabilidade.



Fonte: Elaborado pelo autor.

Quadro 4 – Primitivas representativas utilizadas no experimento de escalabilidade.

Padrão	FreeRTOS (primitivas)	NuttX (primitivas)
1	`uxSemaphoreGetCount()`	`sem_getvalue()`
2–3	`xSemaphoreTake()`, `xSemaphoreGive()`	`sem_wait()`, `sem_post()`
4	`xQueueSend()`, `xQueueReceive()`	`mq_send()`, `mq_receive()`
5	`xEventGroupSetBits()`, `xEventGroupClearBits()`	`pthread_cond_signal()`, `pthread_cond_wait()`
6	`uxTaskPriorityGet()`	`pthread_getschedparam()`

Fonte: elaborado pelo autor.

4.4.3 Terceiro cenário

Este cenário trouxe uma Avaliação de como cada *kernel* lida com contenção de recursos. Aqui, foi avaliado o impacto da contenção de um mesmo recurso sobre o desempenho dos *RTOS*s em execução *multi-core*. O teste utiliza apenas o padrão de aquisição e liberação de semáforo (padrão 2–3), com duas tarefas: a *target task*, que executa continuamente *take/give* no semáforo, e a *stress task*, que acessa o mesmo semáforo por uma fração controlada do tempo. Esse controle é feito através da geração de um número pseudo aleatório (que pode ir de zero a cem) a cada iteração, que é salvo na variável *R*. Esse número é comparado com a Variável *Stress*, que representa a frequncia de Contenção desejada.

A intensidade de contenção foi parametrizada variando de 0% a 70%, em incrementos de 10%. O resultado é reportado como *throughput* normalizado, considerando a execução com *Stress*=0% como referência (100%).

4.5 Coleta e análise dos dados

A coleta de dados foi realizada a partir da execução dos testes na *Raspberry Pi Pico*. Ao final de cada execução, os valores medidos eram enviados ao computador hospedeiro pela UART0 e registrados para posterior processamento.

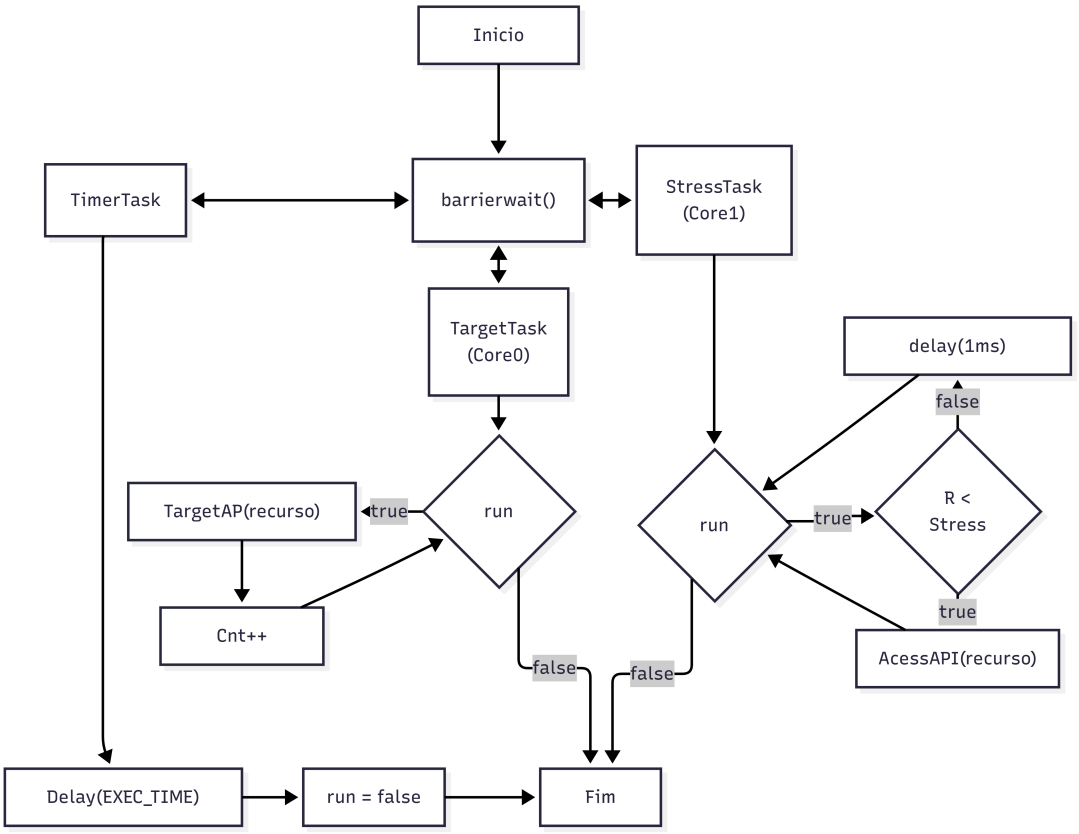
Em todos os cenários, o *throughput* foi obtido a partir do número total de iterações executadas durante a janela temporal do experimento:

$$\mathit{throughput} = \frac{N_{\text{iter}}}{T},$$

em que N_{iter} é o contador de iterações e T é a duração do teste.

No cenário baseado no *Thread-Metric*, os resultados foram analisados por meio do *throughput* bruto e de métricas de memória monitoradas durante a execução. Nos cenários

Figura 15 – Fluxo do cenário de *workload* para avaliação sob contenção.



Fonte: Elaborado pelo autor.

de escalabilidade e de contenção, a análise foi conduzida por *throughput* normalizado. No cenário de escalabilidade, o caso *single-core* foi definido como 100%; no cenário de contenção, a execução com *stressFrequency*=0% foi definida como 100%. Assim:

$$throughput_{norm}(\%) = \frac{throughput_{medido}}{throughput_{base}} \times 100.$$

Por fim, os dados foram organizados por cenário, por *RTOS* e por configuração de execução, e utilizados para gerar tabelas e gráficos comparativos apresentados no capítulo de resultados.

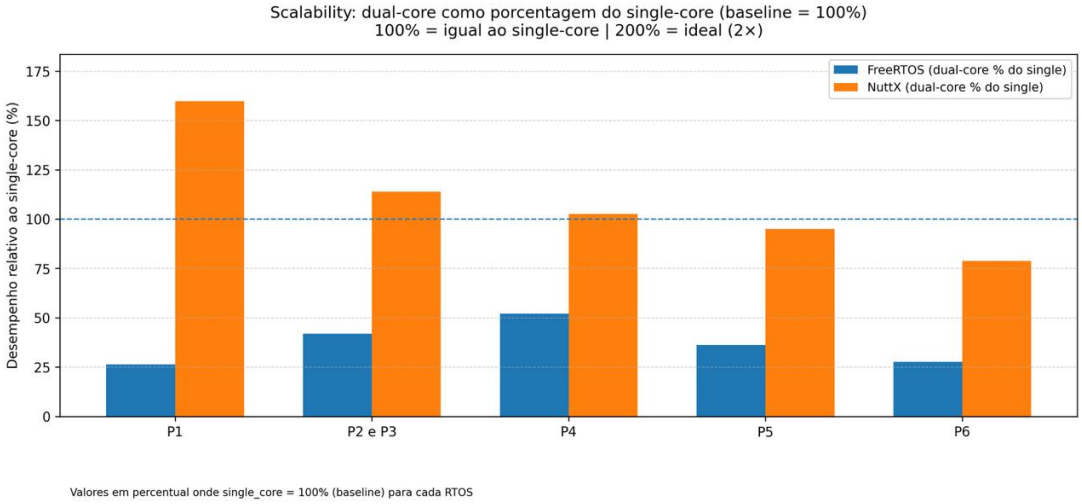
5 RESULTADOS

Este capítulo apresenta os resultados obtidos nos três cenários experimentais definidos neste trabalho. Para facilitar a interpretação, a análise é organizada em ordem crescente de complexidade: inicialmente, são discutidos os resultados do teste de escalabilidade com primitivas isoladas (Seção 5.1), em seguida o cenário de contenção de recursos (*workload*) (Seção 5.2) e, por fim, o conjunto integrado T1–T8 (Seção 5.3), utilizando os achados anteriores para explicar os comportamentos observados.

5.1 Escalabilidade com primitivas isoladas

A Figura 16 apresenta os resultados do cenário de escalabilidade, no qual o *throughput* em *multi-core* é expresso como percentual do caso *single-core* (100%). Como as primitivas foram exercitadas de forma isolada e com recursos separados por núcleo/tarefa, este cenário evidencia principalmente o custo estrutural de sincronização interna do *kernel* ao habilitar *SMP*, sem introduzir contenção deliberada de um mesmo recurso.

Figura 16 – Resultado do cenário de escalabilidade (normalizado em relação ao *single-core*).



Fonte: Elaborado pelo autor.

Os resultados mostram uma degradação acentuada no *FreeRTOS* quando executado em *multi-core*, mesmo em padrões em que não há compartilhamento explícito do recurso entre núcleos. Esse comportamento está diretamente relacionado ao mecanismo de seções críticas do *FreeRTOS*. Em *single-core*, grande parte das *API*s do sistema utiliza `taskENTER_CRITICAL()` para proteger estruturas internas, o que implica custo associado à desativação de interrupções

e à entrada/saída da região protegida. Entretanto, em *SMP*, a proteção vai além: a entrada em seção crítica passa a exigir sincronização entre núcleos por meio de *spinlocks*. Assim, mesmo que dois núcleos estejam operando sobre recursos distintos, eles não conseguem progredir simultaneamente caso tentem entrar em seção crítica no mesmo instante; um núcleo necessariamente aguarda o outro liberar o *lock*. Na prática, isso tende a serializar trechos relevantes das chamadas de sistema, explicando a queda de desempenho observada nos padrões avaliados em *multi-core*.

No *NuttX*, os resultados indicam maior capacidade de preservar desempenho em *SMP*, o que é consistente com o uso de um caminho rápido (*fast path*) em diversas *API*s de sincronização, especialmente semáforos. De forma simplificada, a implementação tenta concluir a operação por meio de instruções atômicas sem entrar imediatamente em seção crítica, e apenas recorre a um caminho lento (*slow path*) quando a condição de sucesso não é atendida. Esse comportamento é ilustrado pelo fluxo típico de `nxsem_wait()`: inicialmente, o código tenta adquirir o semáforo por comparação-e-troca atômica (*compare-and-swap*), retornando sucesso quando o contador indica disponibilidade. Caso o contador não permita aquisição (por exemplo, valor menor que 1) ou a operação atômica falhe por concorrência, a execução segue para `nxsem_wait_slow()`, que entra em seção crítica (`enter_critical_section()`), atualiza estruturas do escalonador, manipula filas de espera e pode bloquear a tarefa até que o semáforo se torne disponível.

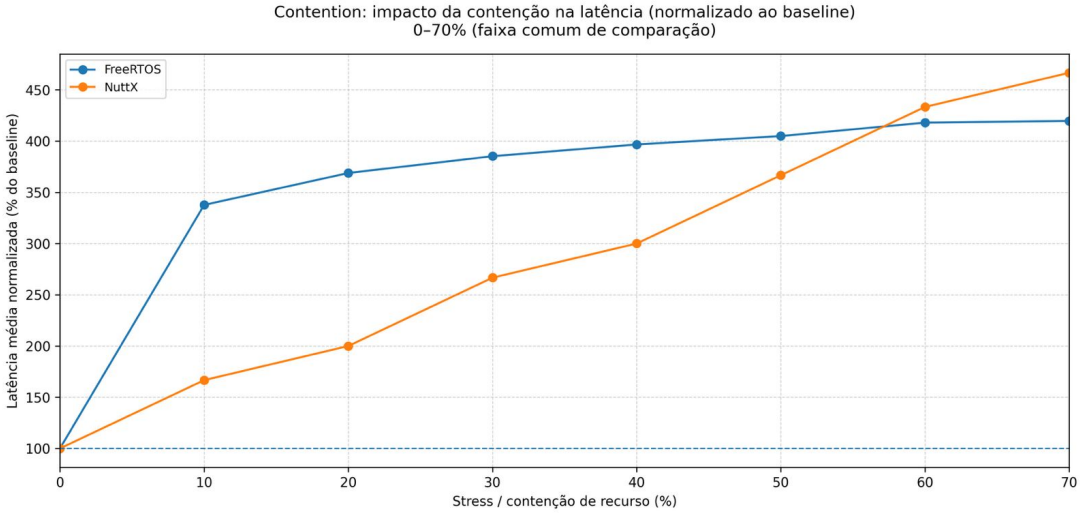
Essa estratégia tem uma implicação direta para este cenário: como os testes de escalabilidade foram construídos para evitar contenção real do recurso (recursos separados por núcleo), o *NuttX* tende a permanecer majoritariamente no *fast path* e, portanto, reduz a frequência de entrada em seções críticas e de manipulação de filas de bloqueio. Em contrapartida, em cenários nos quais a contenção é alta, a probabilidade de queda para o *slow path* aumenta, e o custo adicional associado ao caminho completo (incluindo seção crítica, alterações no escalonador e possível bloqueio) pode se tornar dominante. Essa relação é explorada diretamente no cenário de contenção apresentado na Seção 5.2, no qual o recurso é deliberadamente compartilhado e a taxa de colisão é controlada.

5.2 Cenário de contenção de recursos (*workload*)

A Figura 17 mostra o comportamento dos *RTOS*s sob contenção controlada, variando *stressFrequency* de 0% a 70%. Nesse cenário, a *target task* executa continuamente aquisição/liberação de semáforo, enquanto a *stress task* acessa o mesmo semáforo por uma fração do tempo,

induzindo contenção real do recurso.

Figura 17 – Resultado do cenário de contenção de recursos (*workload*).



Fonte: Elaborado pelo autor.

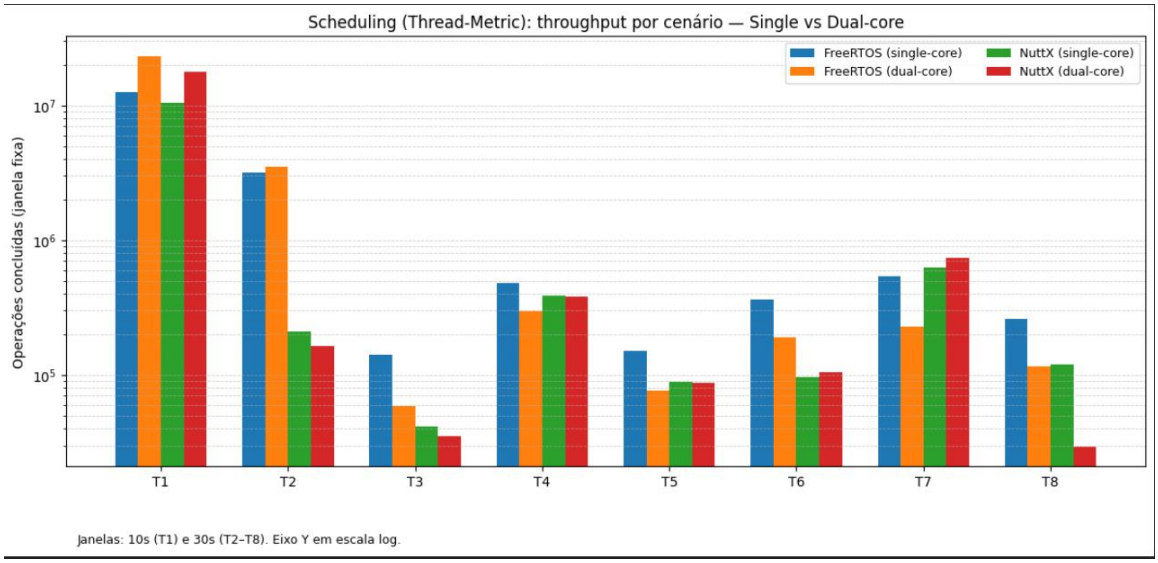
O resultado evidencia dois comportamentos distintos. No *FreeRTOS*, há uma queda abrupta do caso 0% para 10%, indicando alta sensibilidade ao início do paralelismo em regiões críticas: com qualquer nível de acesso concorrente, parte do custo passa a ser dominado pela coordenação de entrada em zona crítica. Entretanto, à medida que *stressFrequency* aumenta, o impacto incremental tende a reduzir, sugerindo que a maior parcela do overhead decorre do mecanismo de bloqueio associado à região crítica em si, e não exclusivamente do tempo de espera pela liberação do semáforo.

No *NuttX*, a degradação tende a ocorrer de forma mais gradual com o aumento de *stressFrequency*. Isso é consistente com a ideia de caminhos *fast*/*slow*: à medida que o semáforo passa a estar ocupado com maior frequência, o sistema é forçado a utilizar com mais recorrência o caminho mais completo (e mais custoso), acumulando overhead progressivamente. Em níveis mais altos de contenção, o custo adicional pode superar o observado no *FreeRTOS*, fazendo com que o *NuttX* apresente resultados comparativamente piores quando a contenção passa a dominar o comportamento do sistema.

5.3 Avaliação integrada de serviços do *kernel* (T1–T8)

Após caracterizar o comportamento estrutural das primitivas em *SMP* (escalabilidade) e sob contenção real (workload), esta seção discute os resultados do conjunto integrado T1–T8. A Figura 18 apresenta uma visão consolidada em escala logarítmica.

Figura 18 – Resultados consolidados do conjunto T1–T8 em escala logarítmica.



Fonte: Elaborado pelo autor.

Em T1, os resultados foram extremamente próximos entre os *RTOS*s, o que é esperado por se tratar de uma carga majoritariamente computacional, com baixa dependência de serviços do *kernel*. Essa proximidade serve como um indicativo de consistência do ambiente experimental e reforça a confiabilidade das comparações nos demais padrões, nos quais o custo do *kernel* passa a ser dominante.

Os padrões T2–T8 reforçam, de forma progressiva, os comportamentos já observados nos cenários de escalabilidade e de contenção. Em linhas gerais, quando o teste é dominado por troca de contexto e operações simples do *kernel*, o *FreeRTOS* tende a se beneficiar por ter um núcleo menor; por outro lado, quando a execução em *multi-core* passa a exigir entradas frequentes em seções críticas, o custo de sincronização em *SMP* no *FreeRTOS* se torna um fator limitante, enquanto o *NuttX* tende a preservar desempenho em situações com baixa contenção por meio de caminhos rápidos (*fast path*).

No T2, a diferença acentuada em favor do *FreeRTOS* é coerente com esse panorama. O padrão é essencialmente um laço com chamadas contínuas de `taskYIELD()`, isto é, mede o custo de alternância cooperativa com mínima lógica adicional. Nesse caso, o *FreeRTOS* apresenta uma troca de execução significativamente mais leve, ao passo que o *NuttX* carrega overhead maior por envolver uma estrutura de *kernel* mais abrangente.

No T3, a vantagem do *FreeRTOS* diminui em relação ao T2. Embora ainda trate de escalonamento, esse padrão aumenta a incidência de sincronizações internas e, principalmente em *multi-core*, reforça o efeito discutido na Seção 5.1: o uso frequente de seções críticas em

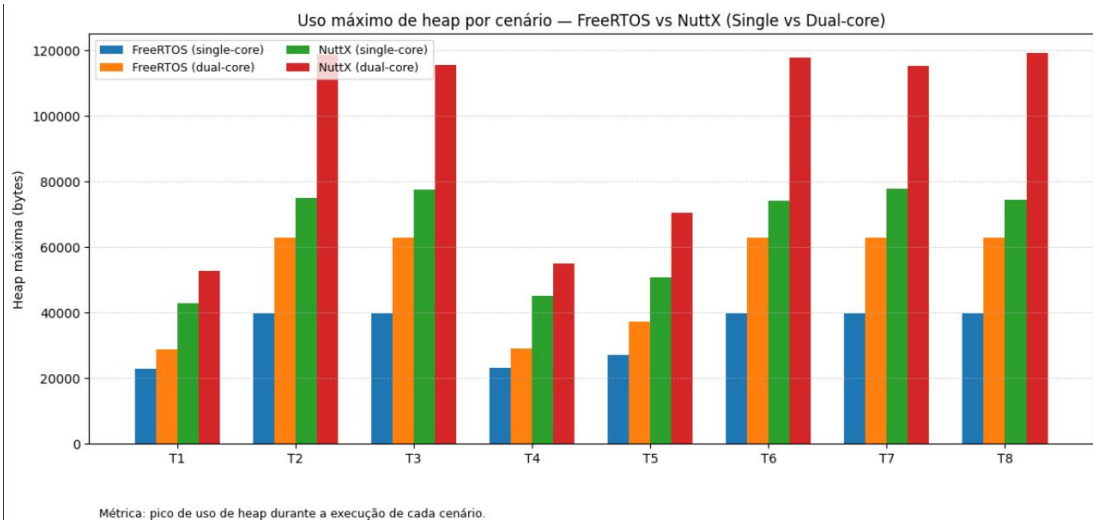
SMP tende a serializar partes do caminho de execução, reduzindo o ganho relativo ao habilitar o segundo núcleo.

Quanto ao desempenho em interrupções (T4 e T5), o FreeRTOS demonstrou uma pequena vantagem em *single-core*, porém perde em *multi-core*. Os padrões T6 e T7 referem-se a cenários de primitivas isoladas que já foram avaliadas anteriormente.

Vale destacar a queda no desempenho em *multi-core* em ambos os *RTOS*s no T8, com uma queda ainda maior no *NuttX*, consequência da implementação de *Malloc* e *Free*, que sempre bloqueia o spinlock em uma mesma heap, trazendo um cenário para ambas com 100% de contenção.

Além do *throughput*, foi analisado o uso máximo de *heap* durante a execução dos padrões (Figura 19). Observa-se que o *NuttX* apresenta picos de *heap* superiores aos do *FreeRTOS*, o que é consistente com a execução em um ambiente mais completo, mantendo componentes do sistema ativos (como o *Nuttshell* e serviços associados) e exigindo mais estruturas internas para gerenciamento de tarefas, comunicação e sincronização. Em *multi-core*, o pico tende a aumentar em ambos os *RTOS*s devido ao maior número de entidades ativas e estados mantidos simultaneamente durante a janela do experimento.

Figura 19 – Uso máximo de *heap* observado durante a execução do conjunto T1–T8.



Fonte: Elaborado pelo autor.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho comparou os *RTOS*s *FreeRTOS* e *NuttX* na *Raspberry Pi Pico* (RP2040), considerando execução *single-core* e *multi-core* sob *SMP*. A avaliação foi conduzida em cenários complementares, indo desde primitivas simples do *kernel* até cargas mais integradas e um cenário dedicado à contenção de recursos, o que permitiu observar diferenças de desempenho e, principalmente, relacioná-las a decisões internas de sincronização e organização do *kernel*.

De forma geral, os resultados indicaram comportamento semelhante em cargas computacionais e diferenças relevantes quando o desempenho passa a ser dominado por escalonamento e sincronização, especialmente em *multi-core*. No *FreeRTOS*, como a maioria das chamadas entra em seções críticas, a proteção por *spinlocks* em *SMP* tende a introduzir serialização entre núcleos, reduzindo a escalabilidade mesmo quando não há contenção deliberada do recurso. No *NuttX*, a presença de caminhos rápidos baseados em operações atômicas favorece o desempenho em baixa contenção e explica a melhor preservação de *throughput* em diversos casos, enquanto o aumento de contenção força a execução a recorrer com maior frequência a caminhos lentos, reduzindo gradualmente essa vantagem. Como continuidade, a análise pode ser estendida para execução em *AMP* e para um conjunto maior de plataformas e *RTOS*s, de modo a verificar a generalização dos resultados e ampliar a discussão sobre como decisões de *kernel* e características de *hardware* influenciam a escalabilidade e o impacto da contenção.

REFERÊNCIAS

ABDELHAMID, A. M. H.; ZAMEL, A. A. Implementing and measuring the performance of pb, rr and pbrr scheduling algorithms on atmega32a using freertos. In: **2023 5th Novel Intelligent and Leading Emerging Sciences Conference (NILES)**. [*S. l.*: *s. n.*], 2023. p. 18–22.

ABDELSAMEA, M. H. A.; ZORKANY, M.; ABDELKADER, N. Real time operating systems for the internet of things, vision, architecture and research directions. In: **2016 World Symposium on Computer Applications Research (WSCAR)**. [*S. l.*: *s. n.*], 2016. p. 72–77.

AMARA, F. Z.; HEMAM, M.; DJEZZAR, M.; MAIMOR, M. Semantic web and internet of things: Challenges, applications and perspectives. **Journal of ICT Standardization**, v. 10, n. 2, p. 261–291, 2022.

ATMADJA, W.; CHRISTIAN, B.; KRISTOFEL, L. Real time operating system on embedded linux with ultrasonic sensor for mobile robot. In: **2014 International Conference on Industrial Automation, Information and Communications Technology**. [*S. l.*: *s. n.*], 2014. p. 22–25.

BACCELLI, E.; DOERR, J.; KIKUCHI, S.; PADILLA, F. A.; SCHLEISER, K.; THOMAS, I. Scripting over-the-air: Towards containers on low-end devices in the internet of things. In: **2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)**. [*S. l.*: *s. n.*], 2018. p. 504–507.

BELLEZA, R. R.; PIGNATON, E. de F. Performance study of real-time operating systems for internet of things devices. **IET Software**, Wiley Online Library, v. 12, n. 3, p. 176–182, 2018.

BORMANN, C.; ERSUE, M.; KERäNEN, A. **Terminology for Constrained-Node Networks**. RFC Editor, 2014. RFC 7228. (Request for Comments, 7228). Disponível em: https://www.rfc-editor.org/info/rfc7228. Acesso em: 07 mai. 2025.

BUENO, M. A. F. **Análise e implementação de suporte a SMP (multiprocessamento simétrico) para o sistema operacional eCos com aplicação em robótica móvel**. Dissertação (Dissertação de Mestrado) – Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, São Carlos, SP, Brasil, 2007. Disponível em: https://www.teses.usp.br/teses/disponiveis/55/55134/tde-21062007-150831/publico/DissertacaoMaikonBueno.pdf. Acesso em: 04 fev. 2025.

CHOUDHARY, V.; TANWAR, S.; RANA, A. Demystifying security and applications of internet of things. In: **2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)**. [*S. l.*: *s. n.*], 2021. p. 1–5.

HAHM, O.; BACCELLI, E.; PETERSEN, H.; TSIFTES, N. Operating systems for low-end devices in the internet of things: A survey. **IEEE Internet of Things Journal**, v. 3, n. 5, p. 720–734, 2016.

HAMBARDE, P.; VARMA, R.; JHA, S. The survey of real time operating system: Rtos. In: **2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies**. [*S. l.*: *s. n.*], 2014. p. 34–39.

HANAFI, M. E.; ABOZIED, M. A. H.; ELHALWAGY, Y. Z.; ELFAROUK, A. O. Real time symmetric multiprocessing software design for onboard computer of guided missiles. In: **2020 12th International Conference on Electrical Engineering (ICEENG)**. [*S. l.*: *s. n.*], 2020. p. 350–355.

ISMAEL, G. A.; SALIH, A. A.; AL-ZEBARI, A.; OMAR, N.; MERCEEDI, K. J.; AHMED, A. J.; SALIM, N. O. M.; HASAN, S. S.; KAK, S. F.; IBRAHIM, I. M.; AL. et. Scheduling algorithms implementation for real time operating systems: A review. **Asian Journal of Research in Computer Science**, v. 11, n. 4, p. 35–51, Sep. 2021.

KANWAL, M.; WAJEEHA, M.; KHAN, N. Iot devices operating systems unveiled: An analysis and comparison of operating system for internet of things. **International Journal of Multidisciplinary Sciences and Engineering**, v. 14, n. 2, p. 1–7, 2023.

LUNA, R.; ISLAM, S. Security and reliability of safety-critical rtos. **SN Computer Science**, v. 2, 09 2021.

MARTOS, P. I.; GARRIDO, A. Software patterns for asymmetric multiprocessing devices on embedded systems: a performance assessment. In: **2017 Eight Argentine Symposium and Conference on Embedded Systems (CASE)**. [*S. l.*: *s. n.*], 2017. p. 1–6.

MAZZI, Y.; GAGA, A.; ERRAHIMI, F. Benchmarking and comparison of two open-source rtoss for embedded systems based on arm cortex-m4 mcu. **Indian Journal of Science and Technology**, v. 14, p. 1261–1273, 04 2021.

NIKOLOV, N.; NAKOV, O.; GOTSEVA, D. Operating systems for iot devices. In: **2021 56th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)**. [*S. l.*: *s. n.*], 2021. p. 41–44.

OJO, M. O.; GIORDANO, S.; PROCISSI, G.; SEITANIDIS, I. N. A review of low-end, middle-end, and high-end iot devices. **IEEE Access**, v. 6, p. 70528–70554, 2018.

OLIVEIRA, D.; CHEN, W.; PINTO, S.; MANCUSO, R. Investigating and mitigating contention on low-end multi-core microcontrollers. In: **Proceedings of Cyber-Physical Systems and Internet of Things Week 2023**. New York, NY, USA: Association for Computing Machinery, 2023. (CPS-IoT Week '23), p. 221–226. ISBN 9798400700491.

PAUL, P. V.; SARASWATHI, R. The internet of things — a comprehensive survey. In: **2017 International Conference on Computation of Power, Energy Information and Commuincation (ICCPEIC)**. [*S. l.*: *s. n.*], 2017. p. 421–426.

REDDY, B. S. N.; VENKAT, B. S.; NAIDU, G. N.; NISHA, K. S. A comprehensive review on functional analysis of real-time operating systems. In: **2023 3rd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)**. [*S. l.*: *s. n.*], 2023. p. 1098–1102.

SILVA, M.; CERDEIRA, D.; PINTO, S.; GOMES, T. Operating systems for internet of things low-end devices: Analysis and benchmarking. **IEEE Internet of Things Journal**, v. 6, n. 6, p. 10375–10383, 2019.

UNGUREAN, I.; GAITAN, N. C. Performance analysis of tasks synchronization for real time operating systems. In: **2018 International Conference on Development and Application Systems (DAS)**. [*S. l.*: *s. n.*], 2018. p. 63–66.

XING, Y.; LI, Y.; TAKADA, H. A multi-core rtos benchmark methodology to assess system services under contentions. **Journal of Information Processing**, Information Processing Society of Japan, v. 32, p. 829–843, 2024.