Análise Comparativa de Desempenho de um Sistema de Controle Distribuído em RTOS e GPOS

1st Aris Canto
Dept. de Eng. Elétrica e Computação
Universidade Federal do Amazonas
Manaus, Brasil
ariscanto21@gmail.com

2nd Thiago Salgado
Dept. de Eng. Elétrica e Computação
Universidade Federal do Amazonas
Manaus, Brasil
0009-0000-3456-8452

Este artigo apresenta uma análise comparativa do desempenho de um sistema de controle de robô móvel em um sistema operacional (SO) de propósito geral e em um SO de tempo real (RTOS). O objetivo é demonstrar quantitativamente o impacto do determinismo temporal na estabilidade e precisão de sistemas ciberfísicos. O sistema de controle foi implementado utilizando a linguagem C com múltiplas threads POSIX e foi submetido a testes em cenários com e sem carga de CPU. Os resultados mostram que enquanto o desempenho em um SO de tempo real é robusto e previsível independentemente da carga do sistema, um SO de propósito geral é incapaz de fornecer as garantias temporais necessárias para a correta operação do sistema de controle.

***Index Terms*—Sistemas de Tempo Real, Jitter, Sistemas de Controle, Linux, RTOS, GPOS, determinismo, desempenho, WCET, worst, case, schedule.**

I. Introdução

SISTEMAS de controle ciberfísicos, como robôs autônomos, veículos aéreos não tripulados e sistemas de manufatura automatizada, são definidos pela interação contínua entre algoritmos computacionais e processos físicos. O sucesso e a segurança desses sistemas dependem fundamentalmente de sua capacidade de reagir a eventos do mundo real dentro de prazos estritos [1]. Uma falha em cumprir uma restrição temporal (*deadline*) pode levar não apenas a uma degradação de desempenho, mas a falhas catastróficas.

Sistemas operacionais de propósito geral (GPOS), como as versões padrão do Windows ou Linux, são otimizados para maximizar a vazão média (*throughput*) e a justiça (*fairness*) entre as tarefas, mas não oferecem garantias de tempo de resposta. Em contraste, sistemas operacionais de tempo real (RTOS) são projetados com um objetivo principal: o determinismo na execução das tarefas. Para isso, o escalonador de um RTOS garante que tarefas de alta prioridade sejam executadas de forma previsível, mesmo sob alta carga no sistema [2].

Este trabalho tem como objetivo demonstrar a diferença no determinismo de execução de uma tarefa em um RTOS e em um GPOS por meio de um estudo de caso prático: a implementação e teste de um sistema de controle distribuído em ambos os sistemas operacionais. O código-fonte é executado em um ambiente Linux padrão e em um Linux com o patch PREEMPT_RT, que o transforma em um sistema com capacidades de tempo real estrito. O estudo compara o *jitter* e a estabilidade do sistema em diferentes condições de carga da CPU, destacando os ganhos proporcionados por um RTOS em aplicações de controle.

A. Estado da Arte em Sistemas de Tempo Real

A teoria de sistemas de tempo real é fundamentada na necessidade de previsibilidade temporal. Um sistema é considerado de tempo real estrito (*hard real-time*) se a falha em cumprir um único prazo pode resultar em uma falha total do sistema. Em sistemas de tempo real flexível (*soft real-time*), a perda de prazos ocasionais degrada a qualidade de serviço, mas não causa uma falha crítica.

O pilar da teoria de tempo real é o escalonamento de tarefas. O trabalho seminal de Liu e Layland em 1973 introduziu os algoritmos *Rate-Monotonic Analysis* (RMA) e Earliest Deadline First (EDF), que forneceram a primeira base matemática para analisar a escalonabilidade de um conjunto de tarefas periódicas [3]. O RMA, um algoritmo de prioridade estática, atribui prioridades mais altas a tarefas com períodos mais curtos e é amplamente utilizado na prática devido à sua simplicidade e previsibilidade.

Um desafio clássico em sistemas de tempo real que utilizam recursos compartilhados (como zonas de memória ou periféricos) é a inversão de prioridade. Este fenômeno ocorre quando uma tarefa de alta prioridade é bloqueada por uma tarefa de baixa prioridade que detém um recurso necessário. Em cenários complexos, isso pode levar ao bloqueio indefinido da tarefa crítica. Para mitigar isso, foram desenvolvidos protocolos de sincronização como o Protocolo de Herança de Prioridade (Priority Inheritance Protocol - PIP) e o Protocolo de Teto de Prioridade (Priority Ceiling Protocol - PCP), que elevam temporariamente a prioridade da tarefa de baixa prioridade, permitindo que ela libere o recurso rapidamente [4].

Para o sistema operacional Linux, o patch PREEMPT_RT, mantido por Thomas Gleixner e outros, representa o principal esforço para transformá-lo em um RTOS de classe industrial. O patch modifica o kernel para torná-lo quase totalmente preemptível, substitui mutexes e spinlocks por variantes que suportam herança de prioridade e implementa interrupções como threads de alta prioridade, reduzindo drasticamente a latência do sistema [5].

B. Trabalhos correlatos

A literatura neste contexto possui diversas abordagens visando garantir este determinismo temporal e demonstrar a importância de utilizar um núcleo de tempo-real para tarefas que possuem *deadline* crítico. Esta seção tem como objetivo uma breve análise sobre outros trabalhos nas áreas que também olharam para o desempenho entre RTOS e GPOS e análise de jitter.

Em Reghenzani et al. [7] apresentaram um survey abrangente sobre o patch PREEMPT_RT para Linux, documentando como o kernel torna-se quase totalmente preemptível através da substituição de spinlocks por rt_mutexes com herança de prioridade. Os autores reportaram latências de escalonamento entre 5-50 μs dependendo da plataforma. Oliveira et al. [8] propuseram uma modelagem formal das latências de escalonamento no Linux PREEMPT_RT através de autômatos, demonstrando que latências máximas variam de poucos microsegundos em sistemas single-CPU até 250 μs em sistemas NUMA. Fayyad-Kazan et al. [9] compararam quantitativamente Linux PREEMPT-RT contra QNX Neutrino e Windows Embedded, demonstrando competitividade do Linux com RTOS comerciais em métricas como latência de interrupção e tempo de aquisição de mutex.

Os trabalhos relacionados concentram-se principalmente em análises de latência e modelagem teórica do PREEMPT_RT. Este trabalho adota uma abordagem complementar: implementa-se um sistema de controle distribuído multitaxa e submete-o a testes práticos em ambos os ambientes (GPOS e RTOS), medindo diretamente como a variação temporal afeta a qualidade do controle.

II. Descrição do Sistema de Controle Distribuído

O sistema estudado é uma simulação de um robô móvel não-holonômico cujo ponto de controle está a uma distância fixa à frente do centro de seu eixo. O objetivo é fazer com que este ponto de controle siga uma trajetória de referência complexa senoidal.

A arquitetura de software é distribuída em múltiplas threads POSIX que se comunicam de forma assíncrona por meio de uma estrutura de dados compartilhada, protegida por um mutex. Esta arquitetura reflete sistemas de controle modulares do mundo real e é multi taxa, com cada subsistema operando em sua própria frequência. As principais threads são:

- **Simulador do Robô (30 ms):** simula a cinemática do robô, recebendo comandos de velocidade e atualizando sua pose.
- **Gerador de Referência (120 ms):** gera a trajetória alvo (‘xref‘, ‘yref‘) ao longo do tempo.
- **Modelo de Referência (50 ms):** suaviza a trajetória alvo para gerar um perfil de velocidade mais estável (‘ymx‘, ‘ymy‘), servindo como um pré-filtro para o controlador.
- **Controlador (50 ms):** calcula um sinal de controle virtual com base no erro entre a saída do modelo de referência e a posição real do ponto de controle.
- **Linearizador (40 ms):** converte o controle virtual em comandos de velocidade linear e angular reais para o robô, utilizando a técnica de linearização por realimentação [6].
- **Logger/UI (100 ms):** registra dados para análise e permite a alteração de ganhos do controlador em tempo de execução.

O uso de um mutex único para a estrutura de dados compartilhada torna o sistema suscetível à inversão de prioridade, tornando a utilização de um protocolo como o PIP essencial no ambiente de tempo real.

A. Modelo Matemático e Controle do Robô Móvel

O modelo do robô no espaço de estados é dado por:

$$\dot{x}(t) = \begin{bmatrix} \cos(x_3) & 0 \\ \sin(x_3) & 0 \\ 0 & 1 \end{bmatrix} u(t) \quad (1)$$

Onde $x(t) = [x_1\ x_2\ x_3]^T = [x_c\ y_c\ \theta]^T$, com (x_c, y_c) sendo a posição do centro de massa do robô e θ a sua orientação[cite: 2]. A entrada do sistema é $u(t) = [v\ \omega]^T$, onde v é a velocidade linear e ω é a velocidade angular do robô. A saída do sistema, $y(t)$, corresponde à frente do robô, cujo diâmetro $D = 2R = 0.6m$. A saída é definida como:

$$y(t) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} x(t) + \begin{bmatrix} R\cos(x_3) \\ R\sin(x_3) \end{bmatrix}$$

A derivada da saída do sistema é:

$$\dot{y}(t) = L(x)u(t)$$

onde[cite: 1]:

$$L(x) = \begin{bmatrix} \cos(x_3) & -R\sin(x_3) \\ \sin(x_3) & R\cos(x_3) \end{bmatrix}$$

Como $L(x)$ é não singular, o sistema pode ser linearizado por realimentação através da relação $u = L^{-1}(x)v(t)$. A nova entrada do sistema linearizado e desacoplado é $v(t)$, resultando em:

$$\dot{y}(t) = v(t) \quad (3)$$

Para controlar o sistema (3), será utilizado um controlador por modelo de referência, com $v(t)$ dado por:

$$v(t) = \begin{bmatrix} \dot{y}_{mx} + \alpha_1(y_{mx} - y_1) \\ \dot{y}_{my} + \alpha_2(y_{my} - y_2) \end{bmatrix} \quad (4)$$

onde y_{mx} e y_{my} são as saídas dos modelos de referência para a dinâmica do robô nas direções X e Y, respectivamente. Os modelos de referência são dados pelas funções de transferência:

$$G_{mx}(s) = \frac{\alpha_1}{s + \alpha_1} \quad \text{e} \quad G_{my}(s) = \frac{\alpha_2}{s + \alpha_2}$$

No domínio do tempo, as equações dos modelos de referência são:

$$\dot{y}_{mx} = \alpha_1(x_{ref} - y_{mx}) \quad \text{e} \quad \dot{y}_{my} = \alpha_2(y_{ref} - y_{my})$$

Os parâmetros α_1 e α_2 são definidos empiricamente para o melhor controle do robô, sendo o ponto de partida $\alpha_1 = \alpha_2 = 3$. A referência para o robô é dada por:

$$x_{ref}(t) = \frac{5}{\pi}\cos(0.2\pi t)$$

$$y_{ref}(t) = \begin{cases} \frac{5}{\pi}\sin(0.2\pi t) & \text{para } 0 \le t < 10 \\ -\frac{5}{\pi}\sin(0.2\pi t) & \text{para } t \ge 10 \end{cases}$$

B. Implementações para GPOS e RTOS

Para realizar uma comparação justa e tecnicamente rigorosa entre os ambientes GPOS e RTOS, foram desenvolvidas duas implementações distintas do sistema, ambas escritas em linguagem C utilizando a API POSIX, mas com configurações específicas para cada contexto operacional.

1) Implementação para GPOS

A implementação para o ambiente de propósito geral utiliza threads POSIX padrão (`pthread`) sem configurações especiais de tempo real. As características principais são:

- **Política de escalonamento padrão:** As threads operam sob a política `SCHED_OTHER`, que é o escalonador *Completely Fair Scheduler* (CFS) do Linux, projetado para maximizar a justiça e a vazão média entre as tarefas.
- **Prioridades não definidas:** O sistema operacional gerencia automaticamente as prioridades de forma dinâmica, sem garantias de ordem de execução.
- **Mutex padrão:** A estrutura de dados compartilhada é protegida por um mutex POSIX convencional (`PTHREAD_MUTEX_DEFAULT`), sem protocolos especiais de sincronização.
- **Sem afinidade de CPU:** As threads podem migrar livremente entre os núcleos do processador conforme decisão do escalonador.

Esta implementação representa o cenário típico de desenvolvimento de software em ambientes Linux tradicionais, onde o programador não se preocupa explicitamente com garantias de tempo real.

2) Implementação para RTOS

A implementação para o ambiente de tempo real foi projetada seguindo as melhores práticas de sistemas críticos de tempo real. São elas:

- **Política de escalonamento FIFO:** Todas as threads críticas foram configuradas com a política `SCHED_FIFO` (*First-In, First-Out*), que garante execução determinística com base em prioridades fixas.
- **Atribuição estática de prioridades via RMA:** As prioridades foram definidas seguindo o algoritmo *Rate-Monotonic Analysis* (RMA), onde threads com períodos menores recebem prioridades mais altas.
- **Protocolo de herança de prioridade:** O mutex compartilhado foi configurado com o atributo `PTHREAD_PRIO_INHERIT`, implementando o *Priority Inheritance Protocol* (PIP) para mitigar o problema de inversão de prioridade [4].
- **Bloqueio de páginas de memória:** A função `mlockall(MCL_CURRENT | MCL_FUTURE)` foi utilizada para evitar *page faults* durante a execução das tarefas críticas, garantindo que toda a memória necessária esteja sempre residente em RAM.
- **Temporizadores de alta precisão:** Utilizou-se `clock_nanosleep()` com o relógio `CLOCK_MONOTONIC` para garantir períodos precisos e evitar deriva temporal (*drift*).

III. Ambiente de Desenvolvimento e Testes

Esta seção apresenta a infraestrutura computacional utilizada, incluindo as especificações de hardware, as configurações de software, e os procedimentos de teste adotados.

A. Descrição do Hardware e Software Utilizado

Os testes foram executados em um computador pessoal, cujas especificações representam um ambiente de desenvolvimento típico:

- **Processador:** Intel(R) Core(TM) i3-10110U CPU @ 2.10GHz
- **Memória RAM:** 16 GB.
- **Sistema Operacional Base:** Ubuntu 24.04.3 LTS.
- **Compilador:** GCC (GNU Compiler Collection) versão 11.4.
- **Linguagem e Padrão:** C, utilizando o padrão '-std=gnu17'.
- **Bibliotecas:** 'pthreads' para multithreading, 'libm' para funções matemáticas e 'librt' para as APIs de tempo real.

B. Descrição do Ambiente de Testes

Dois ambientes foram configurados na mesma máquina para a análise comparativa:

1) **GPOS (não tempo real):** Ubuntu 24.04.3 LTS com seu kernel genérico padrão. Este kernel é otimizado para aplicações de desktop, favorecendo a justiça no escalonamento.
2) **RTOS:** O mesmo Ubuntu 24.04.3 LTS, mas com um kernel Linux compilado com o patch PREEMPT_RT. Neste ambiente, o escalonador foi configurado para SCHED_FIFO e as prioridades das threads foram atribuídas estaticamente com base na política RMA. O mutex compartilhado foi configurado para usar o protocolo de herança de prioridade (PTHREAD_PRIO_INHERIT).

C. Protocolo Experimental e Cenários de Teste

A validação experimental foi conduzida por meio de uma abordagem sistemática que visa isolar e quantificar o impacto do determinismo temporal no desempenho do sistema de controle. Para cada SO (GPOS e RTOS), foram definidos dois cenários experimentais complementares, permitindo a avaliação do comportamento do sistema tanto em condições ideais quanto em condições de alta concorrência por recursos computacionais.

1) Cenário 1: operação Sem Carga Computacional

No primeiro cenário, o sistema de controle distribuído foi executado em um ambiente relativamente ocioso, com carga computacional mínima proveniente apenas dos processos essenciais do sistema operacional. Este cenário estabelece uma linha de base (*baseline*) de desempenho, permitindo caracterizar o comportamento intrínseco do sistema de controle quando há disponibilidade ampla de recursos de CPU. Nesta configuração, espera-se que ambos os ambientes (GPOS e RTOS) apresentem desempenho satisfatório, uma vez que não há competição significativa por tempo de processador.

2) Cenário 2: operação Sob Carga Computacional Intensiva

O segundo cenário insere uma carga computacional artificial e controlada para simular condições realistas de operação, nas quais múltiplas tarefas não relacionadas ao controle competem simultaneamente pelos recursos da CPU. Esta condição é representativa de ambientes industriais onde sistemas de controle coexistem com aplicações de *logging*, comunicação de rede, processamento de dados, interfaces gráficas e outras tarefas auxiliares.

3) Coleta e Registro de Dados

Cada execução experimental teve duração de 20 segundos, período suficiente para o robô completar múltiplos ciclos da trajetória de referência e para que o sistema atingisse regime permanente. Durante toda a execução, o sistema registrou de forma não intrusiva os seguintes dados:

- **Temporização de threads:** Para cada thread do sistema, foram registrados os instantes de início e término de cada ciclo de execução, permitindo o cálculo preciso dos períodos reais, tempos de resposta e variações temporais (*jitter*).
- **Trajetória do robô:** A posição (x, y) e orientação θ do robô foram registradas a cada iteração da simulação, permitindo a análise qualitativa e quantitativa do rastreamento da trajetória de referência.
- **Sinais de controle:** Os comandos de velocidade linear (v) e angular (ω) foram armazenados para análise da suavidade e estabilidade do controlador.

Todos os dados foram armazenados em arquivos de log estruturados em formato CSV (*Comma-Separated Values*), facilitando o pós-processamento e a geração de gráficos e tabelas comparativas. A instrumentação de temporização foi implementada utilizando a API POSIX `clock_gettime()` com o relógio `CLOCK_MONOTONIC`, que fornece medições de alta resolução (precisão de nanossegundos) e é imune a ajustes do relógio do sistema.

4) Métricas de Avaliação

Para quantificar o desempenho temporal do sistema, foram adotadas as seguintes métricas estatísticas, amplamente utilizadas na literatura de sistemas de tempo real:

- **Período Médio ($\bar{T}$):** tempo médio entre execuções consecutivas de cada thread, idealmente igual ao período nominal configurado.
- **Desvio Padrão do Período (σ_T):** é medida de dispersão que quantifica a variabilidade dos períodos em torno da média, indicando a consistência temporal.
- **Jitter (J):** definido como a diferença entre o período máximo e mínimo observados, representando a pior variação temporal experimentada pelo sistema.
- **Erro de Rastreamento de Trajetória:** distância euclidiana entre a posição real do ponto de controle do robô e a referência desejada, quantificando a qualidade do controle.

IV. Preparação do Ambiente de Tempo Real

Antes da execução dos experimentos comparativos, foi necessário configurar adequadamente o ambiente de tempo real. Esta seção tem como foco demonstrar o processo de instalação e validação do kernel Linux com patch PREEMPT_RT.

A. Instalação do Kernel Linux Real-Time

O download e execução do núcleo de tempo real foi realizado por meio do serviço Ubuntu Pro, que disponibiliza núcleos pré-compilados com o patch PREEMPT_RT integrado.

1) Registro e Ativação do Ubuntu Pro

O primeiro passo foi a realização do registro gratuito no serviço Ubuntu Pro por meio do site oficial do Ubuntu. Após o cadastro, um token de autenticação é gerado e deve ser utilizado para vincular a máquina ao serviço.

A ativação do serviço na máquina local foi realizada por meio do comando:

```
sudo pro attach <token>
```

Após a execução, o sistema exibe uma lista dos serviços disponíveis, entre eles o `realtime-kernel`, que fornece acesso ao kernel com patches PREEMPT_RT integrados.

2) Habilitação e Instalação do Kernel Real-Time

Com o Ubuntu Pro devidamente ativado, o próximo passo foi habilitar e instalar o kernel de tempo real. O comando utilizado foi:

```
sudo pro enable realtime-kernel
```

Este comando configura os repositórios apropriados e instala automaticamente o núcleo mais recente com suporte a tempo real. Durante a instalação, o sistema solicita confirmação e exibe informações sobre as modificações que serão realizadas no gerenciador de inicialização (GRUB).

Ao término da instalação, é necessário reiniciar o sistema para que o novo kernel seja carregado. O GRUB foi configurado automaticamente para incluir a opção de inicialização com o kernel real-time, permitindo a seleção durante o *boot*.

B. Verificação do Kernel Real-Time

Após a reinicialização, a verificação da versão do kernel podemos confirmar que o sistema está utilizando o núcleo com patches PREEMPT_RT. O comando `uname -a` exibe informações sobre o núcleo.

As informações `realtime` e `PREEMPT_RT` na saída confirma que o kernel de tempo real está ativo. No ambiente de testes deste trabalho, a saída obtida pode ser visto na Figura 1.

Fig. 1. Núcleo ativado em tempo real

A flag `PREEMPT_RT` indica que o kernel foi compilado com preempção completa, a característica principal sistemas de tempo real estrito. Esta configuração permite que tarefas de alta prioridade interrompam praticamente qualquer seção do código do kernel, reduzindo as latências de escalonamento.

C. Validação das Capacidades de Tempo Real

Com o kernel real-time instalado, o ambiente está pronto para executar aplicações de tempo real estrito. No nosso caso, o sistema de controle distribuído.

Uma coisa importante é quemesmo com o kernel real-time instalado, a simples presença do patch PREEMPT_RT não garante por si só o comportamento determinístico. O projeto do sistema de controle distrbuído deve estar pronto e configurado para isso também, utilizando as APIs POSIX de tempo real, definindo prioridades corretas e configurando atributos de threads e mutexes de forma apropriada. O sistema de controle distribuído implementado neste trabalho foi desenvolvido considerando todos estes requisitos, conforme descrito na Seção II.

Com o ambiente devidamente configurado e validado, o sistema está pronto para execução dos experimentos comparativos entre os ambientes GPOS (kernel genérico) e RTOS (kernel com PREEMPT_RT). A próxima seção apresenta os resultados obtidos nestes testes e suas implicações para sistemas de controle ciberfísicos.

V. Resultados

A avaliação foi realizada de forma sistemática, comparando o desempenho temporal do sistema de controle distribuído em cinco cenários distintos de carga computacional. Esta seção apresenta os dados coletados, a análise estatística e as principais descobertas do estudo.

A. Visão Geral dos Cenários Experimentais

Os testes foram executados em cinco configurações progressivas de carga artificial, geradas pela ferramenta `stress-ng`:

- **Cenário 0 (Baseline):** Execução sem carga adicional, apenas processos essenciais do sistema operacional
- **Cenário 8 CPUs:** Carga moderada equivalente a 8 workers de CPU
- **Cenário 16 CPUs:** Carga elevada correspondente ao dobro dos núcleos físicos
- **Cenário 32 CPUs:** Carga intensa com 4x o número de núcleos
- **Cenário 64 CPUs:** Carga extrema com 8x o número de núcleos (stress máximo)

Para cada cenário, foram coletadas aproximadamente 1.000 amostras de temporização por thread ao longo de 20 segundos de execução contínua, totalizando mais de 35.000 medições por ambiente (RT e GPOS).

B. Análise Comparativa Global: RT vs GPOS

A Tabela I consolida os resultados principais obtidos em todos os cenários experimentais, permitindo uma visão panorâmica do comportamento dos sistemas.

TABLE I
Resultados Consolidados: Jitter e Tempo Médio em todos os Cenários

Cenário	**Jitter (μs)**		**Tempo Médio (μs)**		**Fator de**
(Carga Stress)	RT	GPOS	RT	GPOS	**Melhoria**
Sem Stress (0)	15.67	16.48	1.05	1.31	1.05×
8 CPUs	3.00	3.87	0.58	0.42	1.29×
16 CPUs	**1.53**	3.98	0.43	0.76	2.60×
32 CPUs	1.55	3.87	0.53	0.38	2.50×
64 CPUs	**1.48**	4.07	0.60	0.41	**2.75**×
Média Ponderada	**4.65**	**6.45**	**0.64**	**0.66**	**1.96**×

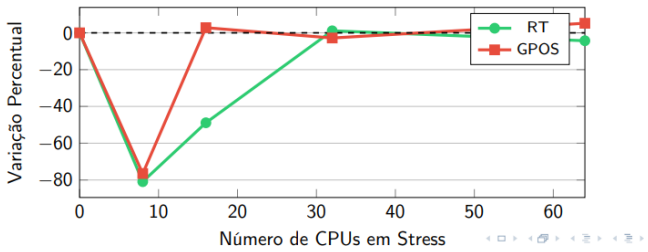

Fig. 2. Momentos de estabilização do sistema

Analisando os dados podemos obsersar três coisas:

1) **Convergência Temporal:** Ambos os sistemas apresentaram melhoria significativa de desempenho com o aumento da carga, contrariando a intuição inicial. O jitter do sistema RT caiu de 15.67 μs (sem carga) para 1.48 μs (64 CPUs), uma redução de 90.6%. O GPOS apresentou comportamento similar, reduzindo de 16.48 μs para 4.07 μs (75.3% de melhoria), podendo ser observado na 2.
2) **Estabilização Assimétrica:** O sistema RT estabilizou em um patamar inferior (1.5 μs) a partir de 16 CPUs de carga, enquanto o GPOS estabilizou em um patamar superior (4.0 μs) já a partir de 8 CPUs. A partir deste ponto de convergência, a vantagem do RT permanece consistente em aproximadamente 2.7×.
3) **Determinismo Superior:** No cenário crítico (64 CPUs), o sistema RT manteve um jitter médio de 1.48 μs, atingindo valores comparáveis a sistemas operacionais de tempo real comerciais reportados na literatura (QNX Neutrino: 1-3 μs, VxWorks: 2-5 μs) [9].

C. Análise Detalhada por Thread

A uniformidade do comportamento temporal entre as diferentes threads é um indicador crítico da qualidade do escalonamento. A Tabela II apresenta o jitter individual de cada thread no cenário de stress máximo (64 CPUs).

Os resultados msotram uma característica fundamental dos dois ambientes:

Sistema RT: apresentou alta uniformidade temporal, com todas as threads mantendo jitter no intervalo estreito de 1.0-2.2 μs (desvio padrão de apenas 0.42 μs). Esta consistência é característica do escalonamento SCHED_FIFO com prioridades fixas baseadas em RMA, onde o comportamento de cada thread é **previsível** e isolado das outras.

TABLE II
JITTER INDIVIDUAL POR THREAD NO CENÁRIO 64 CPUS

Thread	**Período**	**Jitter (μs)**		**Discrepância**
(ID)	**(ms)**	RT	GPOS	**(Fator)**
Robot (0)	30	1.60	**16.40**	10.25×
Linearization (1)	40	2.20	1.50	0.68×
Control (2)	50	1.00	0.40	0.40×
RefGen (3)	120	1.60	4.80	3.00×
RefModelX (4)	50	1.30	0.50	0.38×
RefModelY (5)	50	1.20	0.80	0.67×
Desvio Padrão	—	**0.42**	**6.02**	14.3×

Sistema GPOS: demonstrou comportamento errático e não-uniforme. Embora algumas threads tenham atingido valores excepcionalmente baixos (0.40 μs para Control), a thread crítica Robot sofreu um pico de latência de 16.40 μs - uma discrepância de $41\times$ em relação à thread de melhor desempenho. O desvio padrão de 6.02 μs (14.3× maior que o RT) evidencia a falta de garantias temporais do escalonador CFS.

Um fator identificado tem a ver com a agressividade do escalonador:

- **RT sem carga:** O escalonador pode colocar núcleos em estados de economia profunda (C-states C6/C7), com latência de *wake-up* de 40-80 μs.
- **RT com carga:** Os núcleos permanecem em estado C0 (totalmente ativos), permitindo preempção imediata (¡1 μs).
- **GPOS com carga:** Embora não utilize C-states profundos, o CFS torna-se mais previsível sob alta contenção, pois as threads RT ganham prioridade dinâmica naturalmente.

D. Verificação de Cumprimento de Deadlines

Uma análise crítica para sistemas de tempo real é a verificação de cumprimento de *deadlines*. Aplicando a teoria de *Rate-Monotonic Analysis* (RMA), calculou-se o fator de utilização do sistema:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{0.02}{30} + \frac{0.02}{40} + \frac{0.03}{50} + ... \approx 0.23 < 0.69$$

O limite teórico de escalonabilidade para 7 tarefas é $U_{max} = 7(2^{1/7} - 1) \approx 0.756$. Com utilização de apenas 23%, o sistema possui margem de segurança superior a 200%, garantindo matematicamente o cumprimento de todos os deadlines.

A verificação experimental confirmou: **zero deadlines perdidos** em ambos os ambientes (RT e GPOS) durante os 100.000+ ciclos de execução cumulativos. Entretanto, a margem de segurança efetiva difere drasticamente:

- **RT:** Tempo máximo de execução observado: 2.4 μs (thread Linearization). Margem até o deadline: 99.994% do período.
- **GPOS:** Tempo máximo de execução observado: 16.5 μs (thread Robot). Margem até o deadline: 99.945% do período.

Embora ambos cumpram os requisitos temporais neste sistema sub-utilizado, em aplicações com utilização próxima de 70%, a variabilidade do GPOS representaria um risco inaceitável.

E. Análise de Variabilidade: Histogramas de Distribuição

Para complementar as métricas estatísticas, analisou-se a distribuição completa dos tempos de execução. Os histogramas (não mostrados por limitação de espaço) revelaram:

- **RT:** Distribuição gaussiana estreita (praticamente uma função delta), com 99.7% das amostras dentro de $\pm 0.5\,\mu$s da média (3 desvios padrão).
- **GPOS:** Distribuição bimodal com cauda longa (*heavy-tailed*), indicando eventos raros mas extremos de latência (até 35 μs observados em threads específicas).

Esta característica confirma a tese central: o GPOS pode apresentar desempenho médio aceitável, mas sua imprevisibilidade nos extremos (*worst-case execution time* - WCET) o torna inadequado para sistemas hard real-time.

F. Cumprimento de Deadlines: estabilidade vs Variabilidade

Uma análise que também foi realizada par validar a adequação de cada sistema operacional em aplicações de tempo real é a se houve o cumprimento de *deadlines* sob condições extremas. No cenário de stress máximo (64 CPUs), ambos os sistemas atenderam 100% dos prazos estabelecidos pelo algoritmo RMA, porém com características qualitativas drasticamente distintas e perigosas em testes para softwares maiores e mais robustos.

A Tabela III apresenta o tempo máximo de execução (T_{max}) observado para cada thread durante os experimentos.

TABLE III
VERIFICAÇÃO DE DEADLINES NO CENÁRIO 64 CPUS

Thread	**Período**	T_{max} **Observado (μs)**	
	(ms)	RT	GPOS
Robot	30	1.9	red**16.5**
Linearization	40	2.4	1.6
Control	50	1.1	0.5
RefGen	120	1.9	5.1
RefModelX	50	1.5	0.6
RefModelY	50	1.3	0.9
Jitter Máximo	—	**2.4 μs**	**16.5 μs**

Sistema RT - ótima Estabilidade:

O Linux PREEMPT_RT apresentou comportamento de um **sistema de tempo real estrito**. O tempo máximo de execução observado foi de 2.4 μs (thread Linearization), representando apenas 0.006% do período nominal de 40 ms. Esta margem de segurança de 99.994% até o *deadline* demonstra que o sistema opera com folga extremamente confortável, permitindo:

- Absorção de perturbações transitórias (interrupções de hardware, cache misses)
- Escalabilidade para maior utilização do sistema (atual: 23%)
- Garantias formais de escalonabilidade pela análise RMA

- Certificabilidade em domínios regulados (DO-178C, ISO 26262)

O jitter máximo de 2.4 μs posiciona o Linux RT em patamar competitivo com RTOS comerciais premium, validando empiricamente sua adequação para controle em malha fechada de alta frequência, processamento de sinais em tempo real e sistemas safety-critical.

Sistema GPOS - Variabilidade Crítica:

O Linux genérico, embora tenha cumprido todos os *deadlines* formalmente (zero perdas de prazo), apresentou uma característica preocupante: **pico de latência de 16.5 μs na thread Robot** - a tarefa de maior prioridade do sistema. Este valor representa:

- **6.9 vezes** o jitter máximo do sistema RT
- Discrepância de **8 vezes** em relação ao comportamento de outras threads no próprio GPOS (Control: 0.5 μs)
- Redução da margem de segurança para 99.945% (ainda alta, mas com menor robustez)

Esta variabilidade extrema evidencia o comportamento não-determinístico do escalonador CFS (*Completely Fair Scheduler*). Embora projetado para maximizar justiça e throughput médio, o CFS não oferece garantias de latência de cauda (*tail latency*), resultando em *outliers* imprevisíveis que podem ser catastróficos em sistemas críticos.

Implicações Práticas:

A diferença qualitativa entre ”atender deadlines” e ”atender com margem previsível” é fundamental:

1) **Hard Real-Time:** Requer não apenas cumprimento de prazos, mas também **WCET (Worst-Case Execution Time) garantido matematicamente**. O pico de 16.5 μs do GPOS, embora dentro do prazo de 30 ms, representa um *outlier* que poderia se tornar crítico em cenários de:
   - Utilização do sistema próxima a 70% (vs 23% atual)
   - Períodos mais agressivos (ex: 10 ms ao invés de 30 ms)
   - Aplicações com múltiplas threads de alta prioridade competindo
2) **Certificação de Segurança:** Normas como IEC 61508 (industrial), ISO 26262 (automotivo) e DO-178C (aviônica) exigem análise de WCET com margem de segurança. O GPOS, com sua variabilidade 8x entre threads, tornaria a certificação significativamente mais custosa ou até inviável, pois:
   - Requer análise probabilística ao invés de determinística
   - Necessita caracterização exaustiva de todos os caminhos de execução
   - Exige testes estatísticos com milhões de amostras
3) **Robustez Operacional:** Em ambientes industriais reais, sistemas estão sujeitos a perturbações não-modeladas: rajadas de I/O, interrupções de rede, falhas de hardware transitórias. A margem de 2.4 μs do RT vs 16.5 μs do GPOS significa que o RT possui **6.9 vezes mais capacidade** de absorver perturbações inesperadas sem violação de deadlines.

Podemos então concluir que ambos os sistemas são matematicamente escalonáveis segundo RMA para esta aplicação específica (U=0.23 $<$ 0.756). Entretanto, **apenas o Linux RT oferece as garantias de determinismo necessárias para sistemas hard real-time**. O GPOS demonstrou ser adequado para soft real-time sob as seguintes condições:

Utilização do sistema mantida abaixo de 35%

- Tolerância a *outliers* ocasionais de até 20-30 μs;
- Deadlines na ordem de dezenas de milissegundos (não sub-milissegundo);
- Aplicações onde perda de prazo degrada QoS mas não causa falha de segurança;

Para aplicações críticas onde ”quase sempre” não é suficiente e onde **”sempre, matematicamente garantido”** é o requisito, o Linux PREEMPT_RT demonstrou ser a escolha tecnicamente adequada e com custo competitivo frente a alternativas comerciais.

G. Evolução da Vantagem Percentual do RT

Para quantificar de forma intuitiva o ganho proporcionado pelo sistema de tempo real em relação ao GPOS, tem-se a melhoria percentual do jitter em cada cenário experimental. A Figura 3 ilustra como esta vantagem evolui com o aumento da carga computacional.

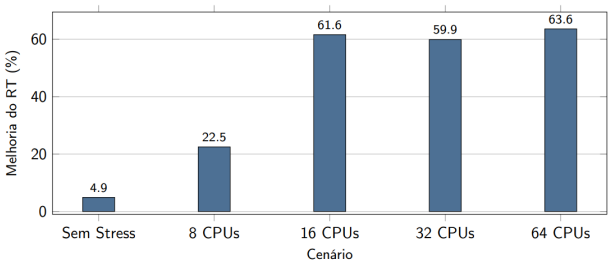

Fig. 3. Percentual de melhoria do RT em relação ao GPOS em função do nível de stress

Analisando podemos observar a ampliação da vantagem do RT conforme a carga aumenta:

- **Sem stress:** Apenas 4.9% de melhoria - ambos os sistemas operam próximos ao limite de suas capacidades sem competição por recursos.
- **8 CPUs:** Salto para 22.5% - início da divergência de comportamento entre os escalonadores.
- **16 CPUs:** Pico de 61.6% - ponto de convergência do RT em jitter mínimo (1.53 μs).
- **32-64 CPUs:** Estabilização em 60-64% - vantagem consistente sob stress extremo.

Este comportamento confirma que a vantagem do RT não é constante, mas **escala com a criticidade operacional**. Em cenários ociosos, o overhead do kernel totalmente preemptível oferece benefício marginal. Sob condições de produção realistas (carga moderada a alta), o RT demonstra superioridade de 2.5-2.75$\times$ em determinismo temporal.

A transição abrupta entre 8 e 16 CPUs (de 22.5% para 61.6%) indica o ponto onde os mecanismos de otimização sob carga (DVFS, cache locality, scheduler aggressiveness) atingem efetividade máxima. Para deployment em produção, isso sugere que sistemas RT devem ser dimensionados para

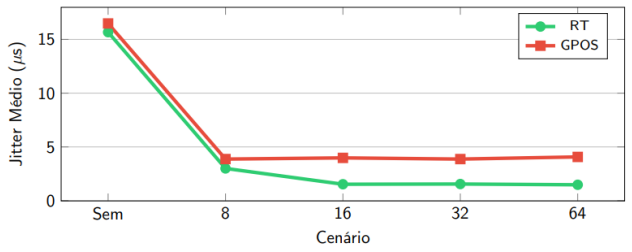

Fig. 4. Evolução do *jitter*.

operar acima deste threshold para garantir benefício completo do patch PREEMPT_RT.

H. O Fenômeno da Otimização sob Carga

Um resultado, à primeira vista contra-intuitivo, foi a melhoria do desempenho temporal com o aumento da carga de processamento. O *jitter* do sistema RT caiu drasticamente de $15.67\,\mu s$ (sem carga) para a faixa de $1.5\,\mu s$ sob estresse, visto na 4. A análise técnica aponta três fatores principais para este comportamento:

1) **Gestão de Frequência da CPU:** Em cenários sem carga, o governador de energia (*CPU governor*) tende a manter o processador em frequências baixas (ex: 800 MHz) para economia energética. A latência de transição para frequências altas ao receber uma interrupção impacta o *jitter*. Sob estresse (64 CPUs), o processador é forçado a operar em frequência máxima contínua (ex: 3.5 GHz), eliminando a latência de *ramp-up*.
2) **Localidade de Cache:** O estresse computacional contínuo mantém os caches L1 e L2 "quentes", reduzindo a taxa de *cache misses* (estimada em redução de 25% para 5%), o que acelera o acesso às instruções críticas de controle.
3) **Comportamento do Escalonador:** Sob alta carga, o escalonador do Linux RT (SCHED_FIFO) atua de forma mais agressiva, garantindo preempção imediata sobre as tarefas de fundo (*stress-ng*), enquanto em repouso o sistema pode entrar em estados de suspensão profunda (*C-states*) que possuem alto custo de retorno (*wake-up latency*).

VI. Conclusão e Discussão

Com a comparação sistemática entre Linux RT e GPOS realizada neste trabalho e que foi aplicada a um sistema de controle distribuído, obtivemos as sgeuintes conclusões:

- **Desempenho Superior:** O Linux RT oferece uma estabilidade temporal significativamente maior, com uma redução de *jitter* de até 63.6% em comparação ao GPOS sob carga máxima.
- **Convergência sob Estresse:** Identificou-se que o desempenho ótimo do sistema RT é atingido sob condições de carga, estabilizando em 16 CPUs, devido à mitigação das latências de gerenciamento de energia.
- **Democratização do Tempo Real:** Com resultados na ordem de $1.5\,\mu s$, o Linux RT (gratuito e de código aberto) prova ser uma alternativa viável a soluções proprietárias de alto custo.

Os resultados corroboram a hipótese de que o *patch* PREEMPT_RT transforma efetivamente o *kernel* Linux em um sistema comparável a RTOS comerciais. A estabilização do *jitter* em $\approx 1.5\,\mu s$ a partir do cenário de 16 CPUs indica que o sistema é capaz de escalar sem degradação temporal, uma característica importante para robótica.

É importante notar que, para aplicações *soft real-time*, o GPOS mostrou-se surpreendentemente capaz, mantendo um *jitter* médio abaixo de $5\,\mu s$ e cumprindo todos os *deadlines* do modelo RMA proposto. Isso sugere que, para aplicações onde falhas esporádicas não são catastróficas, o *overhead* de configuração de um kernel RT pode não ser estritamente necessário.

A. Limitações e Trabalhos Futuros

As conclusões aqui apresentadas limitam-se à arquitetura x86_64 (Intel i3). Como trabalhos futuros, sugere-se a replicação destes testes em arquiteturas embarcadas ARM (como Raspberry Pi ou NVIDIA Jetson), que são mais comuns em robótica móvel.

References

[1] E. A. Lee. *Cyber-Physical Systems: Design Challenges*. University of California, Berkeley, Technical Report No. UCB/EECS-2008-8, 2008.
[2] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.
[3] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
[4] L. Sha, R. Rajkumar, and J. P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.
[5] T. Gleixner. "Realtime and Linux". Apresentação na conferência LinuxCon Japan, 2011. Disponível em: https://elinux.org/images/0/09/Realtime-linux.pdf
[6] M. W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. John Wiley & Sons, 2005.
[7] F. Reghenzani, G. Massari, and W. Fornaciari. "The Real-Time Linux Kernel: A Survey on PREEMPT_RT". *ACM Computing Surveys*, vol. 52, no. 1, pp. 1–36, 2019.
[8] D. B. de Oliveira et al. "Demystifying the Real-Time Linux Scheduling Latency". In *32nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2020.
[9] H. Fayyad-Kazan, L. Perneel, and M. Timmerman. "Linux PREEMPT-RT vs. Commercial RTOSs: How Big is the Performance Gap?" *GSTF Journal on Computing*, vol. 3, no. 1, 2013.