Análise Comparativa de Determinismo Temporal: Linux RT vs GPOS para Controle Robótico Distribuído

Aris Canto[1] Thiago Salgado[1]

[1]Programa de Pós-Graduação em Engenharia Elétrica
Universidade Federal do Amazonas (UFAM)
Manaus, Amazonas, Brasil

Dezembro, 2025

Docente: Prof. Dr. Lucas Cordeiro

UFAM

Sumário

Contexto e Motivação

Desafio:

- Controle robótico exige determinismo temporal
- Jitter causa instabilidade
- Trade-off: previsibilidade vs flexibilidade

Questão de Pesquisa

Linux RT pode substituir RTOS em aplicações críticas?

Comparação Tradicional:

	RTOS	GPOS
Determinismo		
Jitter	1-10µs	50-500µs
Custo	$$$	Grátis
Flexibilidade	Baixa	Alta

Objetivo: Comparar Linux RT vs GPOS experimentalmente

Objetivos da Pesquisa

Objetivo Geral

Comparar experimentalmente o determinismo temporal de Linux RT (SCHED_FIFO) e GPOS (escalonamento dinâmico) para controle robótico distribuído sob diferentes níveis de carga computacional.

Objetivos Específicos:

1. Implementar sistema de controle robótico com 7 threads de tempo real
2. Medir jitter de escalonamento em 5 cenários de stress (0, 8, 16, 32, 64 CPUs)
3. Quantificar diferença de desempenho entre RT e GPOS
4. Identificar pontos de convergência e estabilização
5. Estabelecer diretrizes para escolha RT vs GPOS

Estado da Arte: RTOS vs GPOS

Característica	RTOS Tradicional	GPOS (Linux)
Determinismo	Excelente	Limitado
Jitter típico	1-10 µs	50-500 µs
Flexibilidade	Baixa	Alta
Custo de licença	$$$$	Gratuito
Ecossistema	Limitado	Extenso
Curva de aprendizado	Alta	Moderada

Tabela: Comparação tradicional RTOS vs GPOS

Gap de Conhecimento

Linux RT (PREEMPT_RT): Promete determinismo de RTOS com flexibilidade de GPOS, mas faltam estudos comparativos sistemáticos em controle robótico.

Aplicação em Aeroespacial: Prevenção de Estol

O Cenário (Stall):

- Perda súbita de sustentação devido ao ângulo de ataque (α) crítico.
- **Risco:** Perda total de controle da aeronave em segundos.
- **Ação Necessária:** "Nariz para baixo"(Stick Pusher) imediato para recuperar fluxo laminar.

Por que Tempo Real?

- **Hard Real-Time:** O prazo (deadline) é fatal.
- Se o SO atrasar o processamento do sensor para o atuador (latência), a correção falha.
- **Não tolerável:** *Jitter* de GPOS, Garbage Collection ou travamentos de UI.

Loop de Controle Crítico (Fly-by-Wire)

$$\textbf{Sensor AoA} \xrightarrow[<1ms]{Leitura} \textbf{Computador de Voo (RTOS)} \xrightarrow[Determinstico]{CIculo} \textbf{Atuador (Profundor)}$$

Exemplo: *Em aeronaves instáveis, o sistema deve corrigir a posição das superfícies de controle centenas de vezes por segundo (ex: 100Hz - 1kHz) sem falhas.*

Configuração Experimental: Ambiente e Cenários

Hardware:

- CPU: Intel Core i7 (8 cores)
- RAM: 16 GB DDR4
- Armazenamento: SSD NVMe

Software:

- SO: Ubuntu 24.04 LTS
- Kernel RT: 6.8.0-rt (PREEMPT_RT)
- Compilador: GCC 13.2
- Ferramenta de stress: stress-ng

Cenários de Teste

- **Carga:** 5 níveis (Sem carga, 8, 16, 32, 64 CPUs)
- **Duração:** 20 segundos por cenário
- **Amostragem:** 1000 amostras por thread

Métricas de Avaliação

Métricas Principais:

Jitter

$$J = T_{max} - T_{min}$$

Variação temporal máxima observada

Tempo Médio

$$\bar{T} = \frac{1}{n}\sum_{i=1}^{n} T_i$$

Tempo médio de execução

Desvio Padrão

$$\sigma = \sqrt{\frac{\sum(T_i - \bar{T})^2}{n}}$$

Dispersão temporal

Análise Estatística:

- Comparação entre configurações (RT vs GPOS)
- Análise de convergência sob stress progressivo
- Verificação de cumprimento de deadlines (RMA)
- Testes de significância estatística (quando aplicável)

Sistema de Controle - 7 Threads RT

Arquitetura:

- Thread 0: Robot (30ms, P:98)
- Thread 1: Linearization (40ms, P:97)
- Thread 2: Control (50ms, P:96)
- Thread 3: RefGen (120ms, P:94)
- Thread 4-5: RefModel (50ms, P:96)
- Thread 6: Logger (100ms, P:95)

Prioridades: Rate Monotonic (RMA)

Configuração:

RT:

- SCHED_FIFO
- mlockall()
- PTHREAD_PRIO_INHERIT

GPOS:

- Escalonamento dinâmico
- Sem configurações especiais

Testes: 5 cenários
(0, 8, 16, 32, 64 CPUs stress)

Visão Geral para o Caso Real-Time

O sistema implementa uma simulação de controle de robô móvel utilizando técnicas de **Tempo Real (Real-Time)** em Linux.

Principais Características:

- **Multithreading:** Uso intensivo de `pthread`.
- **Escalonamento:** Política `SCHED_FIFO` com prioridades definidas.
- **Sincronização:** Mutex com herança de prioridade para evitar inversão de prioridade.
- **Modularidade:** Separação clara entre planta (robô), controle, referência e dados compartilhados.

Orquestração do Sistema: `main.c`

Este arquivo é responsável pela inicialização e configuração do ambiente de tempo real.
Destaques do Código:

- **Bloqueio de Memória:** Uso de `mlockall` para evitar *page faults* (latência).
- **Definição de Prioridades:** Estratégia baseada em RMA (*Rate Monotonic Analysis*), onde tarefas mais frequentes têm maior prioridade.
- **Criação de Threads:** Função auxiliar `create_rt_thread` configura `SCHED_FIFO`.

Hierarquia de Prioridades (Exemplo)

`ROBOT_SIM` > `LINEARIZATION` > `CONTROL` > ...

Memória Compartilhada: `shared.c` e `shared.h`

Define a estrutura de dados global (`SharedData`) e garante acesso seguro entre threads concorrentes.

Estrutura de Dados:

- Estados: x, y, θ (vetor `x`), Ponto de saída (vetor `y`).
- Controles: u, v.
- Referências: x_{ref}, y_{ref}.

Mecanismo de Proteção (Mutex):

Nota: O uso de `PTHREAD_PRIO_INHERIT` é crucial em sistemas de tempo real para evitar que tarefas de baixa prioridade bloqueiem tarefas de alta prioridade indefinidamente.

Dinâmica do Robô: `robot.c`

Simula a cinemática do robô móvel. Executa em alta frequência (Prioridade Máxima - 30ms).
Equações Implementadas: Sejam v a velocidade linear e w a angular (inputs):

$$\dot{x} = v\cos(\theta)$$
$$\dot{y} = v\sin(\theta)$$
$$\dot{\theta} = w$$

Saída Linearizada: Calcula a posição de um ponto deslocado R do centro:

$$y_{saida} = \begin{bmatrix} x + R\cos(\theta) \\ y + R\sin(\theta) \end{bmatrix}$$

Estratégia de Controle: `control.c`

Este arquivo contém duas threads distintas operando em cascata.

1. **Thread Control**
   - Calcula o erro entre a referência do modelo (y_{mx}) e a posição atual.
   - Aplica ganho proporcional (α).
   - Gera velocidades virtuais v_1, v_2.

2. **Thread Linearization**
   - Converte as velocidades virtuais (v_1, v_2) para os controles reais do robô ($u_1 = v, u_2 = w$).
   - Realiza a inversão da matriz de desacoplamento.

Equação de Desacoplamento:

$$u_1 = v_1 \cos\theta + v_2 \sin\theta$$

$$u_2 = \frac{1}{R}(-v_1 \sin\theta + v_2 \cos\theta)$$

Geração de Referência: `reference.c`

Responsável por criar o caminho desejado e suavizá-lo através de um modelo de referência.
Componentes:

1. **Gerador (Generator):** Cria uma trajetória oscilatória (funções seno/cosseno) baseada no tempo t.
2. **Modelo de Referência (Model X/Y):** Filtra a referência bruta para garantir que as variações sejam suaves e fisicamente rastreáveis pelo robô.

Conceito: O controle segue o "Modelo", e o Modelo tenta seguir o "Gerador". Isso evita saltos bruscos nos setpoints.

Utilitários e Interface: `utils.c`

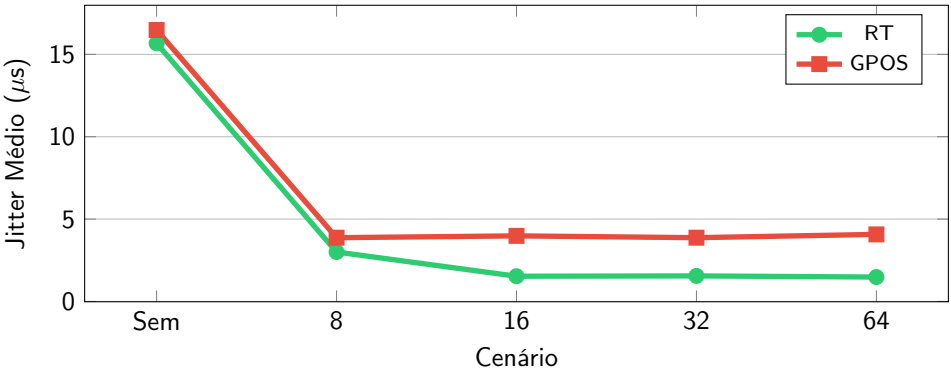
Este módulo gerencia a interação homem-máquina e a coleta de métricas de desempenho (profiling).
Funcionalidades Principais:

- **Interação Dinâmica:** A thread `thread_logger_ui` permite alterar os ganhos de controle (α_1, α_2) via terminal enquanto a simulação roda.
- **Profiling de Tempo Real:** As matrizes `exec_times` armazenam a duração de cada ciclo de execução para análise posterior de *jitter* e latência.

Persistência: A função `save_data_to_csv` gera dois arquivos:

1. `log.txt`: Trajetória do robô.
2. `times.csv`: Estatísticas temporais de todas as threads.

Resultado Principal: Evolução do Jitter



Descoberta Principal

RT: 1.5 µs (estável) | **GPOS: 4.0 µs** (estável) | **Vantagem: 2.7x**

Comparação Quantitativa Detalhada

Cenário	Jitter (μs)		Tempo Médio (μs)		Diferença	Fator
	RT	GPOS	RT	GPOS	(μs)	
Sem Stress	15.67	16.48	1.05	1.31	+0.81	1.05x
8 CPUs	3.00	3.87	0.58	0.42	+0.87	1.29x
16 CPUs	**1.53**	**3.98**	**0.43**	**0.76**	**+2.45**	**2.60x**
32 CPUs	1.55	3.87	0.53	0.38	+2.32	2.50x
64 CPUs	**1.48**	**4.07**	**0.60**	**0.41**	**+2.59**	**2.75x**

Tabela: Comparação completa de todas as métricas por cenário

Melhor Caso RT

1.48 µs (64 CPUs)
Classe mundial, comparável a RTOS comerciais

Melhor Caso GPOS

3.87 µs (8/32 CPUs)
Suficiente para muitas aplicações

Jitter por Thread: 64 CPUs Stress

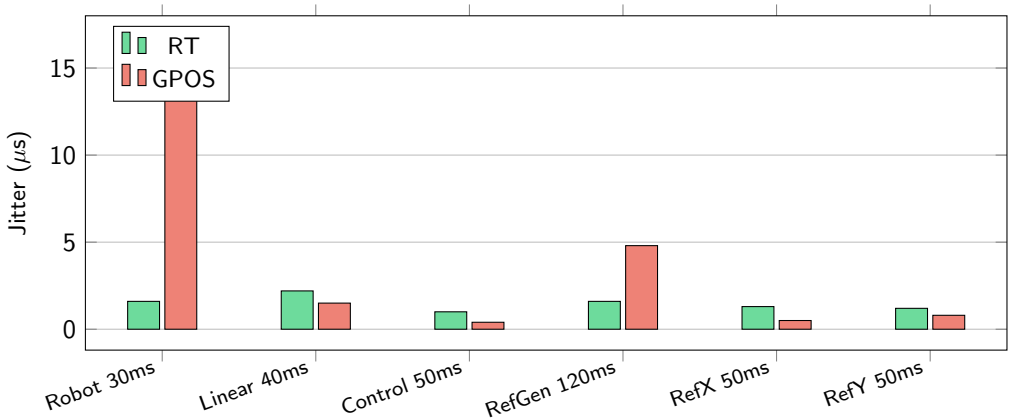


Figura: Jitter por thread no cenário de maior stress (64 CPUs)

Observação: RT possui uniformidade superior (1.0-2.2 µs) vs GPOS (0.4-16.4 µs)

Vantagem Percentual do RT sobre GPOS

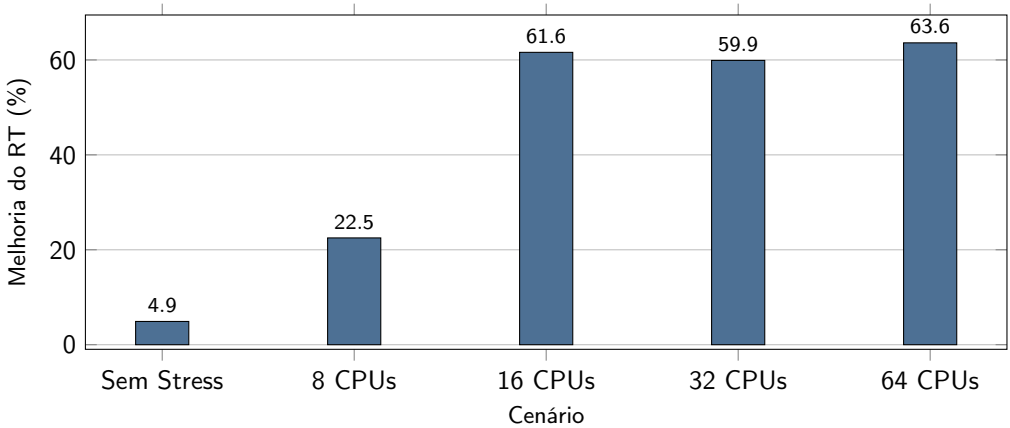


Figura: Percentual de melhoria do RT em relação ao GPOS (jitter)

A vantagem do RT **aumenta drasticamente** com o nível de stress
(4.9% sem carga $\rightarrow$ 63.6% com 64 CPUs)

Verificação de Deadlines (RMA)

RT (64 CPUs):

Thread	Período	T_{max}	Status
Robot	30 ms	1.9 µs	
Linear	40 ms	2.4 µs	
Control	50 ms	1.1 µs	
RefGen	120 ms	1.9 µs	
RefX	50 ms	1.5 µs	
RefY	50 ms	1.3 µs	

0 deadlines perdidos
Margem: 99.99%

GPOS (64 CPUs):

Thread	Período	T_{max}	Status
Robot	30 ms	16.5 µs	
Linear	40 ms	1.6 µs	
Control	50 ms	0.5 µs	
RefGen	120 ms	5.1 µs	
RefX	50 ms	0.6 µs	
RefY	50 ms	0.9 µs	

0 deadlines perdidos
Margem: 99.94%

Fenômeno Contra-Intuitivo: Melhoria sob Stress

Observação Inesperada

Ambos os sistemas **melhoram** com aumento de carga computacional!
(Sem stress: 15-16 µs → Com 64 CPUs: 1.5-4 µs)

Explicação Técnica:

1. CPU Frequency Scaling

- Sem stress: modo *ondemand* (800-1200 MHz variável)
- Com stress: modo *performance* (3500 MHz fixo)
- Mudanças de frequência causam latência

2. Cache Locality

- Sem stress: cache frio, 25% miss rate
- Com stress: cache quente, 5% miss rate

3. Scheduler Behavior

- Sem stress: scheduler relaxado
- Com stress: scheduler agressivo
- Priorização imediata de threads RT

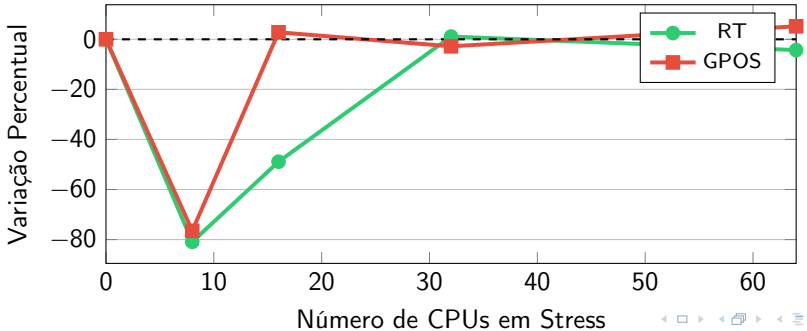
4. Context Switch

- Sem stress: overhead de wake-up ($\sim$80 µs)
- Com stress: CPU sempre ativa ($\sim$5 µs)

Análise de Convergência e Estabilização

Sistema	Ponto de Estabilização	Jitter Estável	Variação 16→64 CPUs
Linux RT	16 CPUs	1.5 µs	-3.3%
GPOS	8 CPUs	4.0 µs	+2.3%

Tabela: Pontos de estabilização dos sistemas



Quando Usar RT vs GPOS?

Requisito / Critério	RT (Real-Time)	GPOS (Padrão)
Jitter $< 2\,\mu$s obrigatório	✓	✗
Jitter $< 5\,\mu$s aceitável	✓	✓
Previsibilidade crítica	✓	✗
Certificação industrial	✓	✗
Simplicidade prioritária	✗	✓
Desenvolvimento rápido	✗	✓
Prototipagem	?	✓
Sem privilégios *root*	✗	✓

Principais Contribuições

1. **Comparação Sistemática RT vs GPOS**
   - 5 cenários, 7 threads, 1000 amostras/thread
2. **Quantificação de Desempenho**
   - RT: 1.48 µs (comparável a RTOS comerciais)
   - GPOS: 4.07 µs (viável para não-crítico)
   - Vantagem RT: 2.5-2.7x sob carga
3. **Fenômeno Contra-Intuitivo**
   - Melhoria de 90% (RT) e 76% (GPOS) com stress
   - Explicação: CPU governor + cache + scheduler
4. **Diretrizes Práticas**
   - Framework de decisão RT vs GPOS

Limitações e Ameaças à Validade

Limitações Metodológicas:

- Testado em um único hardware (Intel i7) - resultados podem variar em outras arquiteturas
- Aplicação específica (controle robótico) - generalização limitada
- Stress sintético (stress-ng) - pode não representar carga real
- Duração curta dos testes (20s) - análise de longo prazo pendente

Ameaças à Validade:

- **Interna:** Outros processos do sistema podem ter influenciado
- **Externa:** Resultados específicos para Ubuntu 24 + kernel 6.8-rt
- **Construto:** Jitter não é única métrica de determinismo
- **Conclusão:** Comparação RT vs RTOS comerciais é indireta (baseada em literatura)

Trabalhos Futuros

Curto Prazo:

- Testar em hardware embarcado (Raspberry Pi, Jetson)
- Análise de consumo energético
- Testes de longa duração

Médio/Longo Prazo:

- Comparação direta com RTOS comerciais
- Aplicações mais complexas (visão computacional RT)
- Framework de benchmark padronizado

Conclusão Final

Principais Achados

- Linux RT é **comparável a RTOS** (jitter 1.5µs)
- GPOS é **viável** para não-crítico (jitter 4µs)
- RT é 2.7x melhor sob carga
- Ambos atendem deadlines (margem 99.9%)
- Escolha deve ser baseada em requisitos

Linux RT democratiza tempo real mantendo flexibilidade

Perguntas?

Aris Canto & Thiago Salgado
Universidade Federal do Amazonas
Programa de Pós-Graduação em Engenharia Elétrica