Do Impacto das *Syscalls* Linux em Tarefas de Tempo Real

Benhur Tessele[1], **Rômulo Silva de Oliveira**[1], **Cristian Koliver**[2]

[1]Centro Tecnológico (CTC) – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88040-900 – Florianópolis – SC– Brasil

[2]Departmento de Automação e Sistemas (DAS) – UFSC

[3]Departamento Informática e Estatística (INE) – UFSC

benhurzi@gmail.com, {cristian.koliver,romulo.deoliveira}@ufsc.br

Abstract. *Despite the increasing use of Linux as the operating system used in embedded real-time applications, many kernel services have different performance features compared to other standard POSIX operating systems. Such particularities can greatly influence the performance of real-time applications. In this paper, we describe the results achieved by using the Perf tool related to some typical situations of real-time applications. Our study shows that the solution or mitigation of bottlenecks requires a deeper knowledge of the functioning of the Linux kernel.*

Resumo. *Apesar do uso crescente do Linux como o sistema operacional utilizado em aplicações embarcadas de tempo real, muitos serviços do kernel possuem características diferenciadas de funcionamento em relação a outros sistemas operacionais padrão POSIX e que podem influenciar sobremaneira o desempenho das aplicações de tempo-real. Neste artigo, descrevemos os resultados obtidos utilizando a ferramenta Perf para algumas situações típicas que ocorrem em aplicações de tempo real. Nosso estudo mostra que a solução ou mitigação de gargalos identificados exige do projetista, em boa parte dos casos, um conhecimento mais profundo do funcionamento do kernel do Linux.*

1. Introdução

Cada vez mais, o Linux[1] vem sendo utilizado como sistema operacional (SO) para sistemas embarcados mais sofisticados. Mesmo não sendo um SO originalmente concebido para aplicações de tempo real (*real-time application* ou RTA) críticas, algumas versões de seu kernel direcionadas para RT têm se tornado a plataforma preferida para execução de aplicações com restrições temporais críticas e não críticas, como por exemplo, em sistemas embarcados de veículos aéreos não tripulados (UAVs, de *unmanned aerial vehicle*). Motivos para tal incluem: gratuidade e abertura do código; portabilidade e a escalabilidade, permitindo que as soluções computacionais de mercado evoluam, migrem para outras plataformas, e possuam mais funcionalidades sem retrabalho no SO; existência de uma comunidade que apoia o Linux presente no mundo todo, o que facilita a obtenção de informações para desenvolvimento e suporte de soluções e aplicações.

[1]https://www.kernel.org/

Por outro lado, muitos serviços do kernel do Linux possuem características diferenciadas de funcionamento em relação a outros SOs padrão POSIX. Essas características, muitas vezes parcamente documentadas, podem influenciar sobremaneira o desempenho das RTAs, impactando, inclusive, no atendimento de suas restrições temporais. Neste artigo, descrevemos os resultados obtidos utilizando a ferramenta Perf para algumas situações típicas em programas voltados para RTAs: acesso à memória secundária, exclusão mútua a seções críticas e alocação dinâmica de memória. Nosso estudo mostra que a solução ou mitigação dos gargalos identificados exige do projetista, em boa parte dos casos, um conhecimento profundo do funcionamento do kernel do Linux.

A principal contribuição deste trabalho é mostrar como os gargalos nas execuções das *syscalls* do Linux afetam o desempenho de aplicações, descrevendo precisamente como essa influência ocorre, e como o uso dessas *syscalls* pode comprometer o cumprimento de requisitos temporais de RTAs críticas. Outra contribuição do nosso trabalho é mostrar algumas das potencialidades e limitações da ferramenta Perf enquanto ferramenta para medições de desempenho de aplicações para Linux.

O texto está assim estruturado: na Seção 2, descrevemos alguns trabalhos cujo escopo inclui avaliação do impacto de *syscalls* no desempenho de tarefas de tempo real; na Seção 3 superficialmente descrevemos a ferramenta Perf; na Seção 4, descrevemos o ambiente de testes, as medições realizadas (e como elas foram feitas via Perf) e os resultados obtidos; finalmente, nas Seção 5 discutimos os resultados e apresentamos algumas propostas de trabalhos futuros.

2. Trabalhos Relacionados

Dada a disseminação do uso de versões do Linux como o SO de RTAs embarcadas, há alguns trabalhos na literatura recente cujo escopo é a realização de testes para verificação da adequação e das limitações desse SO nesse contexto. Geralmente, os trabalhos têm como foco uma RTA específica e as medições são realizadas sobre essa aplicação. Também, a interferência é medida de maneira global, utilizando comandos disponíveis em qualquer SO Unix-like, como *time*, *date*, *prof* ou *gprof*, que apresentam baixa resolução e granularidade alta (apenas no nível de tarefa ou sub-rotina). Apenas a título de exemplos, descrevemos alguns desses trabalhos a seguir.

[Nabil and Ben Saoud 2011] descrevem uma avaliação comparativa (*benchmarking*) realizada por eles de vários kernels Linux aprimorados para o suporte de RTAs. Os parâmetros de comparação são latência, vazão e pior tempo de execução. Ele não realiza, entretanto, um estudo no nível de *syscalls*.

[Kadar et al. 2019] analisam a adequação do uso de *desktops* como hardware para execução de sistemas de detecção de intrusão de host (HIDS). Os autores propõem uma metodologia para avaliar o impacto da implantação de instrumentação de *syscalls* nesse contexto fornecendo medições de sobrecarga, com pior, melhor e tempo de execução médio de algumas *syscalls* Linux para esse tipo específico de aplicação.

Em [Reghenzani et al. 2017], é proposto um modelo para interferências intra e inter kernel e é realizada uma análise quantitativa da sobrecarga induzida pelo SO. Nos experimentos, os autores identificaram que uma tarefa comum pode aumentar em mais de 50% o pior tempo e o tempo médio de execução de uma tarefa de tempo real crítica

apenas pelas *syscalls* Linux que ela realiza. Os autores não discriminam nem detalham como cada *syscall* contribui nesse aumento.

Em [Reghenzani et al. 2019] encontramos uma compilação das abordagens recentes para a construção de SOs de tempo real baseados no Linux, com foco no PREEMPT_RT. Os autores resumem os resultados de vários trabalhos nos quais são realizadas medidas de latência de escalonamento, entrada e saída e comunicação inter-processos.

3. Perf

A ferramenta Perf[2] está entre as mais populares para análise, depuração e monitoramento de tarefas executadas sobre sistemas Linux-like devido à sua capacidade de fornecer resultados das aplicações executando tanto no modo usuário bem como no modo supervisor, sem a necessidade de outras ferramentas externa, e abstraindo diferenças de hardware [Eranian et al. 2015]. Outra vantagem do Perf é o fato dele fazer parte do código fonte do Linux (introduzido a partir da versão 2.6.31 do kernel), o que faz com que a ferramenta seja constantemente aprimorada e evoluída. Os eventos mensuráveis pelo Perf são provenientes de diversas fontes, estando presentes em grande parte dos componentes do kernel e do hardware. Eles podem ser classificados nos seguintes grupos:

- eventos de hardware: disponibilizados pela Unidade de Monitoramento de Performance (PMU), um componente de depuração e monitoramento presente nos processadores atuais. A PMU possui registradores que contam a ocorrência de diversos eventos dentro do processador, como, por exemplo, número de acesso à memória, número de instruções executadas, número de ciclos gastos, quantidade de ausências de cache (*cache misses*) e predição de desvios realizadas. Esses contadores são denominados Contadores de Monitoramento de Performance (PMCs) ou Contadores de Instrumentação de Performance (PICs). Os eventos da PMU são específicos de cada arquitetura, o que significa que o número de eventos disponíveis varia de acordo com o modelo e o fabricante dos processadores. Por exemplo, enquanto as arquiteturas ARMv7 e ARMv8 registram cerca de trinta eventos de hardware, as arquiteturas x86 e x64 registram mais de duzentos;
- eventos de software: monitorados por meio de contadores que o kernel utiliza para informar a ocorrência de eventos como temporizadores internos da CPU; ausências de página (*page faults*), chaveamentos de contexto, migrações de CPU e outros.

Além do Perf, outras ferramentas, permitem a coleta de dados quando da execução de tarefas em ambientes Unix-like. Por exemplo, a ferramenta OProfile permite contar quantos ciclos um programa precisou para executar, através da leitura dos contadores da CPU, ou até mesmo analisar a pilha de chamadas de funções (*call stack*) [Levon 2004]. Ela também permite analisar o *assembly* do código compilado. Entretanto, existem algumas limitações associadas a esta ferramenta, como, por exemplo sua usabilidade. Ademais, o Perf possui recursos sem análogos no OProfile[3].

Outra ferramenta bastante popular é o PAPI. Ela apresenta um método muito eficiente de ler contadores da CPU e calcular os índices de performance de aplicações

[2] https://perf.wiki.kernel.org/

[3] https://developer.ibm.com/tutorials/migrate-from-oprofile-to-perf/

[Terpstra et al. 2010]. Segundo [Weaver 2013], a ferramenta possui um baixo *overhead* associado às leituras dos registradores que armazenam as informações sobre a CPU. Entretanto, diferente de outras ferramentas, o PAPI não fornece uma interface com o usuário pela linha de comando. Suas funcionalidades estão disponíveis por meio de uma biblioteca, codificada na linguagem C. Assim, o uso do PAPI é intrusivo, pois exige que o código fonte da aplicação invoque funções de sua biblioteca para coleta de dados de execução. Outra desvantagem do PAPI decorre do fato dos eventos disponíveis para observação se limitarem a eventos relacionados à CPU.

4. Estudos de Caso

Os estudos de caso foram executados sobre um microcomputador Raspberry Pi modelo 3 que utiliza o *chip* BCM2837 da Broadcom, que, por sua vez, integra um processador ARM Cortex-A53 *Quad-Core* de 1.2 GHz com arquitetura ARMv8. O *clock* da CPU foi configurado no modo *powersave*, resultando numa frequência de 600 MHz. O SO usado foi o Raspbian, uma distribuição baseada do Linux Debian da própria fundação Raspberry Pi, com a versão do kernel 4.14.

4.1. Sobrecarga de Acesso à Memória Não Volátil

Nossos experimentos iniciais visaram obter dados relativos aos tempos de execução de uma tarefa de tempo real como um todo e, posteriormente, de seus procedimentos individualmente. Para tal, desenvolvemos uma aplicação que utiliza diferentes recursos do sistema, como rede, manipulação de arquivos e processamento de dados. Ela possui uma sub-rotina periódica denominada *netio_task* que realiza os seguintes passos: (1) comunicação com um servidor através do protocolo TCP/IP; (2) criptografia dos dados recebidos do servidor; (3) escrita dos dados criptografados em um arquivo. Embora a aplicação represente apenas um exemplo, existem várias soluções no mercado que possuem comportamento semelhante, como, por exemplo, *gateways* que realizam leitura periódica de sensores e escrevem em arquivos as informações lidas com a finalidade de manter o histórico de leituras. O objetivo deste estudo de caso é realizar a análise dos tempos de resposta da sub-rotina periódica descrita acima, verificar se seus requisitos temporais estão sendo cumpridos, e, em caso contrário, apontar eventuais problemas e possíveis correções para que os requisitos temporais sejam alcançados. A aplicação foi implementada em C, e utiliza as funções da biblioteca GNU *sys/sockets.h* para realizar a comunicação TCP. O processo de criptografia dos dados utiliza o Padrão de Criptografia Avançada (AES), com chave simétrica de 128 *bits*, sendo seu algoritmo disponível na biblioteca *openssl*. A sub-rotina *netio_task* é invocada por uma *thread* (*netio_thread*) a cada 100 ms. Essa *thread* também inicia a conexão com o servidor usada para a comunicação de *netio_task*. Por questões de espaço, neste artigo limitaremos nossa descrição aos resultados obtidos em relação à escrita de dados.

Com o intuito de encontrar as partes do código que demandam maior tempo de execução, coletamos dados temporais em cada passo da execução da função chamando

```
perf probe -x netio -L netio_task
```

e da adição de pontos de prova para calcular os tempos de execução das funções *send*, *recv*, *encrypt_now*, *fopen* e *fprintf*.

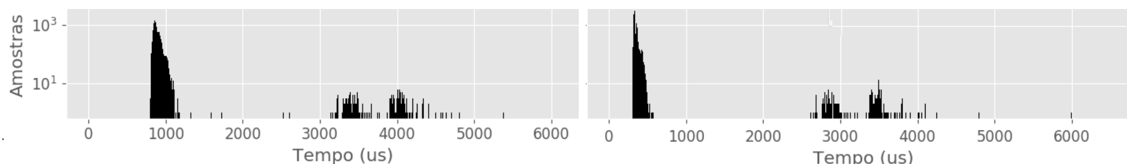

Figura 1. Histograma dos tempos de resposta das funções: *netio_task* (esq.); *fopen* e *fprintf* juntas (dir.)

O histograma da Figura 1 (à esquerda) mostra a concentração de amostras por tempo de execução para a função *netio_task*. Em síntese, das 10.000 amostras, 9.412 obtiveram tempo de resposta inferior a 1.000 μs e o pior tempo de execução – 6.546 μs — foi alcançado por apenas uma amostra. Comportamento bastante semelhante ocorreu nas amostragens realizadas da entrada da função *fopen* até a saída da função *fprintf* (Figura 1, à direita). Conforme o esperado, o tratamento das *syscalls* para manipulação de arquivos são o motivo dos picos nos tempos de resposta de *netio_task*. Visando mitigar o problema da sobrecarga do acesso à memória não volátil, modificamos nossa aplicação para que ela realizasse menos acessos a essa memória às custas de uma maior carga de dados a cada operação de escrita. Para isso, alteramos *netio_task* para que ela armazenasse os dados criptografados em um *buffer*. A cada dez execuções, ela sinaliza uma *thread* que escreve os dados do *buffer* no meio de armazenamento não volátil invocando a função *safe_write*. Com essa modificação, o pior tempo de execução de *netio_task* caiu para 1.048 μs. A Figura 2 mostra o histograma das 1.000 execuções da função *safe_write*. O pior tempo de execução subiu foi de 51.411 μs. Esses resultados reforçam o forte impacto das operações de abertura e escrita em arquivos no tempo de execução.

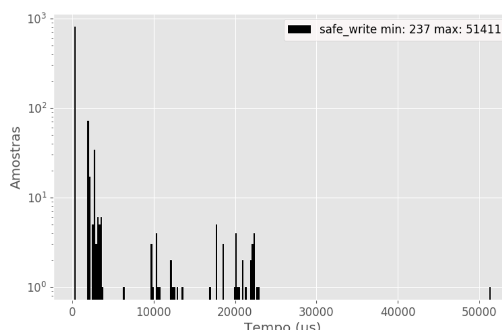

Figura 2. Histograma dos tempos de resposta da função *safe_write* referente a 1.000 execuções.

Realizamos várias execuções que apresentaram resultados bastante díspares, mostrando que a manipulação de arquivos possui tendência de sofrer grande variação de tempo de resposta para operações de escrita. Para descobrir as causas dessa variação, analisamos os eventos do kernel por meio do comando *perf list tracepoint* que ocorrem durante a execução de *safe_write*. Os eventos escolhidos para serem monitorados foram: (1) *sched_switch*: evento que informa a troca de processos realizada pelo escalonador do Linux; (2) *jbd2_start_commit*: evento que sinaliza o início da escrita no *journal*; (3) *block_rq_issue*: evento que informa o envio de uma requisição de escrita para o *driver* de bloco do Linux; (4) *mmc_request_start*: evento que informa o início da escrita no barramento do cartão SD. Nossas medições mostraram que a ocorrência dos eventos (1) e (4)

coincidia com o aumento no tempo de execução de *safe_write*.

Por fim, realizamos testes com o intuito de aumentar a tolerância a falhas em caso de *crash* forçando a escrita dos dados em memória não volátil. Conforme [Bovet and Cesati 2005], capitulo 15, o *driver* de bloco do sistema, responsável por realizar as transições de dados da memória principal para o disco (ou cartão SD, nesse caso), possui caches que são alocadas dinamicamente na memória pelo kernel. Isso implica que as operações em arquivo realizadas por *safe_write* não são imediatamente escritas em disco, mas ficam armazenadas nas caches. Periodicamente, o kernel escreve os dados da cache para no disco (daí as execuções atípicas detectadas nos testes realizados com *safe_write*). Tal estratégia contribui para melhorar o desempenho geral do sistema, pois um cenário onde toda operação de escrita é imediatamente realizada na memória não volátil resulta numa degradação do sistema. Para quantificarmos essa degradação, utilizamos a *syscall sync*, que força a persistência dos dados da cache para a memória não volátil. Essa *syscall* é realizada após cada operação de escrita em *safe_write*. Os resultados (Figura 3) mostraram o forte impacto nos tempos de execução de *safe_write* quando da não utilização da cache de disco. Portanto, do ponto de vista do desempenho de aplicações que manipulam arquivos, existe uma grande vantagem em manter o comportamento normal de caches.

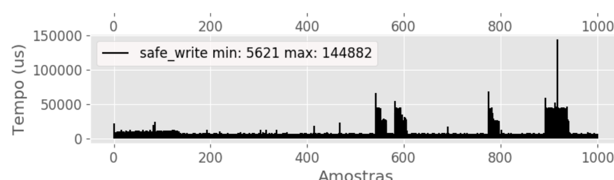

Figura 3. Histograma dos tempos de resposta da função *safe_write* para escrita forçada

4.2. Sobrecarga de Operações de Sincronização

O Linux não possui *syscalls* para operações de bloqueio e desbloqueio de variáveis *mutex*. Em vez disso, ele fornece *syscalls* que podem ser usadas para implementar um mutex no espaço do usuário. São elas: (1) *futex_wait*: suspende a execução da *thread* que a chamou; (2) *futex_wake*: acorda uma ou n *threads* que estão suspensas no kernel. A função *futex_wait* recebe dois argumentos: o endereço de uma variável inteira definida pelo usuário e o valor esperado dessa variável. Se os valores não corresponderem, a chamada retornará imediatamente, assim evitando a ativação de chamadas perdidas. A API GNU POSIX *pthread* disponibiliza as operações *lock* e *unlock* para bloqueio e desbloqueio de variáveis mutex implementadas usando as funções futex. Visando diminuir o número de *syscalls*, a função de *lock* realiza uma operação atômica no espaço de usuário para alocar o *mutex* sem interferência do kernel, caso não seja contestada. Por outro lado, caso a operação seja contestada (outra *thread* já alocou o *mutex*), a função faz uma *syscall* para disparar escalonador para suspender imediatamente a execução da *thread* contestada que permanece suspensa até que o *mutex* seja liberado quando ela volta para o estado pronto (*ready*).

Nesse estudo de caso, utilizamos um algoritmo para o cálculo de π por meio da regra da decomposição obtida via integração, no qual é definido um intervalo que repre-

senta o número de iterações para realizar o cálculo, e a cada iteração, calcula-se o ponto médio da função e acumula-se a soma, conforme a Eq. 1:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx. \quad (1)$$

Realizamos duas implementações desse algoritmo nas quais o cálculo da integral é dividida em N intervalos calculados por N *threads* trabalhadoras; o número de termos (precisão) da série é n. Na primeira implementação (Algoritmo 1), cada *thread* i ($i = 0, 1, 2, ...N-1$) acumula o seu intervalo calculado na variável compartilhada pi. Assim, essa operação representa uma seção crítica. Na segunda implementação (Algoritmo 2), cada *thread* i acumula o seu intervalo calculado na i-ésima posição do vetor pi. Ao final, a *thread* mestre soma as posições do vetor e divide o resultado por uma constante.

Algoritmo 1: Cálculo aproximado de π com contenção

1 $lim \leftarrow (i+1) \times n/N$
2 $k \leftarrow n/N \times i + 1$
3 **repita**
4 $\quad mx.lock$
5 $\quad pi \leftarrow pi + 4.0/(1.0 + x^2)$
6 $\quad mx.unlock$
7 $\quad k \leftarrow k + 1$
8 **até** $k < lim$;

Algoritmo 2: Cálculo aproximado de π sem contenção

1 $lim \leftarrow (i+1) \times n/N$
2 $k \leftarrow n/N \times i + 1$
3 **repita**
4 $\quad pi[i] \leftarrow pi[i] + 4.0/(1.0 + x^2)$
5 $\quad k \leftarrow k + 1$
6 **até** $k < lim$;

Utilizamos a estratégia de cálculo de π da Eq. 1 porque ela resulta em uma seção crítica enxuta e rapidamente computada, o que torna a disputa pelo *mutex* mais intensa, facilitando a análise de desempenho das funções de exclusão mútua. No primeiro cenário de teste, utilizamos $n = 4.000$ e $N = 4$, resultando em 1.000 iterações por *thread* e os recursos de *perf probe* para instrumentar a entrada e a saída da seção crítica. O relatório gerado pelo Perf mostrou detalhadamente quando e quais *threads* adquiriram e liberaram o mutex, permitindo, assim, verificar os tempos de espera para ingresso na seção crítica.

A Figura 4 mostra os histogramas dos tempos de resposta da função *pthread_mutex_lock* para cada *thread*. O tempo máximo – 19.765 μs – foi obtido pela *thread* t_3. Por meio dos eventos gerados por *perf record* sabemos que isso ocorreu porque t_3 tentou alocar o *mutex* e falhou, pois foi t_1 que retornou da função com sucesso. A partir desse instante, t_3 teve sua execução suspensa e as demais *threads* disputaram o mutex 758 vezes, fazendo com que t_3 esperasse 19.765 μs até alocar o *mutex* e retomar sua execução. Essas resultados corroboram com a observação de [Drepper 2008]: a função *futex_wake* não seleciona as *threads* aguardando um mutex usando a política FIFO e tampouco garante respeito às prioridades.

Visando analisar o impacto pela disputa de mutexes, realizamos um teste no qual π foi calculado 1.000 vezes usando os Algoritmos 1 e 2, ambos com $N = 4$ e $n = 4.000$. A Figura 5 apresenta os resultados das 1.000 execuções do Algoritmo 1 e a Figura 6 os resultados para o Algoritmo 2. A diferença dos resultados em termos de tempos de execução de cada *thread* é gritante: enquanto aquelas que usavam a função com mutex

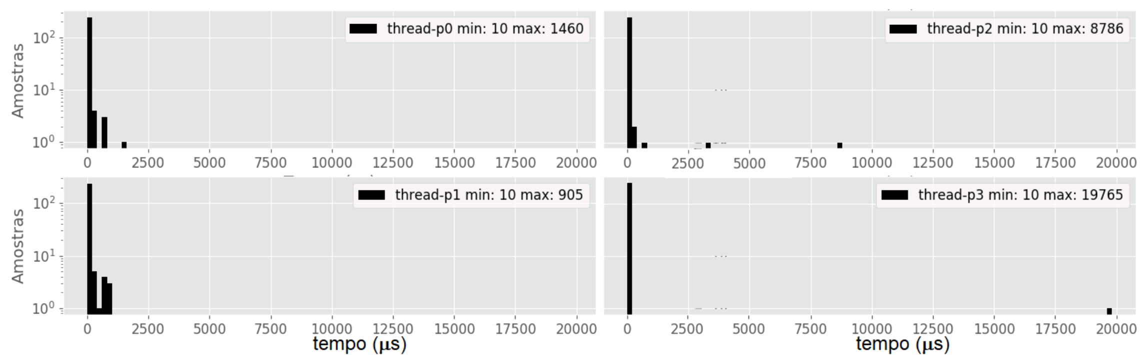

Figura 4. Histogramas das *threads* t_0, t_1, t_2, e t_03 referentes aos tempos de resposta da função *pthread_mutex_lock*, com eixo Y em escala logarítmica.

tiveram, na maioria das vezes, tempos de execução distribuídos entre 1.000 μs e 3.000 μs, as *threads* que executaram a função sem mutex permaneceram distribuídas entre 200 μs e 600 μs. Isso ocorreu porque o uso de mutexes gera significativamente mais trocas de contexto e *syscalls*. Para obter uma métrica precisa, utilizamos o comando

```
perf stat -e sched:switch -e syscalls:sys_enter_futex
```

para contar o número de trocas de contexto e *syscalls futex*. Os resultados do comando mostraram que as 1.000 execuções do Algoritmo 1 geraram cerca de 45 vezes mais trocas de contexto do que nas execuções do Algoritmo 2, além de gerar 2.000 *syscalls futex* . Portanto, essa diferença, juntamente com os resultados observadoe mostrados graficamente nos histogramas das Figuras 6 e 5, reforçam o real impacto de um cenário *multi-thread* com disputa intensa por um *mutex*.

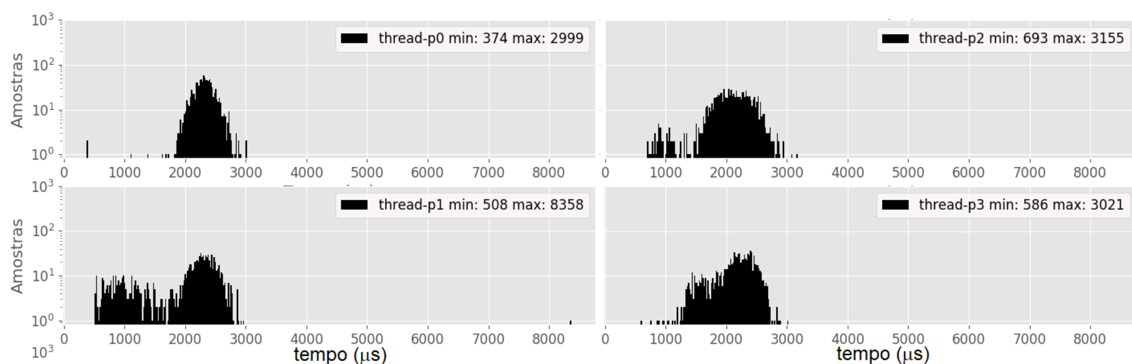

Figura 5. Histograma de 1.000 execuções do Algoritmo 1 com 4 *threads* realizando 1.000 iterações cada, com eixo Y em escala logarítmica.

4.3. Avaliação de Desempenho de Alocação Dinâmica de Memória

Embora os desenvolvedores de RTAs evitem o uso de alocação dinâmica de memória por temerem que o pior tempo de execução das sub-rotinas de alocação não sejam limitados ou tenham um limite excessivamente alto [Puaut 2002], algumas aplicações exigem essa estratégia. Por exemplo, aplicações que coletam e processam imagens em tempo real não podem reservar espaço para armazenamento e processamento de quadros de maneira estática.

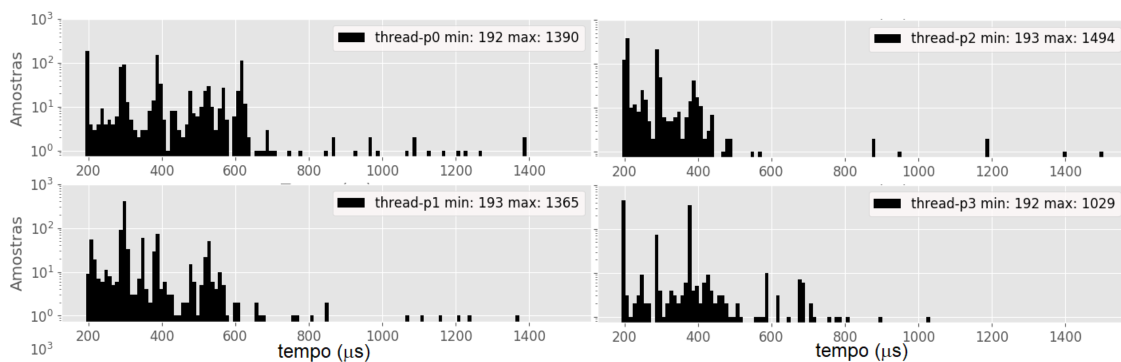

Figura 6. Histograma de 1.000 execuções do Algoritmo 1 realizando 1.000 iterações cada, com eixo Y em escala logarítmica.

Para realizar as tarefas de alocação dinâmica de memória física, o Linux utiliza, principalmente, dois algoritmos: o *buddy allocator* e o *slab allocator*. O primeiro é invocado para alocar páginas físicas. Ele determina a primeira potência de 2 cujo valor satisfaça à requisição de páginas feita pelo kernel. Embora muito eficiente, pois em poucas iterações as requisições são atendidas, o uso do *buddy allocator* recai na fragmentação interna. Por exemplo, se o kernel requisita 65 páginas, o algoritmo retornará 128, o que resulta em 63 páginas desperdiçadas. Para resolver a fragmentação interna, o Linux utiliza o *slab allocator*, que aproveita os pedaços de memória que sobraram oriundos da requisição do *buddy allocator*. O *slab allocator* tem um papel importante no gerenciamento de memória, pois o kernel constantemente instancia e remove estruturas e objetos internos de diferentes subsistemas, criando caches exclusivas para estas estruturas que são alocadas e gerenciados pelo *slab allocator*. A técnica de utilização de caches ajuda consideravelmente na performance do sistema, porque permite que o kernel reutilize as caches quando instancia novas estruturas e objetos do mesmo tipo.

Nosso teste inicial visou identificar quais *syscalls* são ativadas quando da chamada da função *malloc* com alocações de 1 KB. Para isso, observamos os eventos relacionados às *syscalls kmalloc*, *cache_alloc* e *page_alloc* bem como a outros 3 eventos: entrada das funções *main* (*main_entry*), *malloc* (*start_alloc*) e *free* (*end_alloc*)). O relatório gerado pelo Perf mostrou que, após a entrada da função *main*, ocorre a sequência de chamadas *kmalloc*, *cache_alloc* e *page_alloc*, que estão relacionadas às estruturas que o kernel cria para gerenciar a memória do processo. O relatório mostrou, também, que o processo invocou a *syscall brk* para expandir seu *heap*, acarretando na alocação de mais estruturas internas e páginas físicas que comportasse a requisição de 1 KB feita por *malloc*. Por fim, o relatório mostrou que a alocação de 1 KB não resultou em alocação de página física, apenas numa *syscall kmalloc* para alocação de 64 bytes para uso interno do kernel.

A seguir, repetimos o teste para alocação de 1 MB. O relatório gerado pelo Perf mostrou que a sequência após a entrada da função *main* é semelhante àquela do teste anterior, o que era esperado devido ao comportamento de todo processo no Linux. Por outro lado, a requisição de 1 MB fez com que *malloc* invocasse a *syscall mmap*, utilizada para requisições de grande quantidade de memória. Esta, por sua vez, resultou em 251 chamadas à *page_alloc*, que faz as alocações de páginas físicas de 4 KB.

O próximo teste teve como objetivo validar a técnica *copy-on-write* de alocação

física que o Linux utiliza. Para isso, nosso programa alocava memória mas não fazia nenhuma referência posterior à área alocada. O relatório do Perf mostrou a ocorrência dos mesmos eventos relacionado à alocação de 1 MB do teste anterior. Entretanto, não ocorreram as 251 alocações de páginas físicas.

Realizamos, também, testes de alocações aleatórias para analisar o comportamento temporal diante de um cenário com muitas requisições de alocação com variadas quantidades de memória. Isso permite verificar se a fragmentação de memória consequente dessas alocações pode ser um problema no que diz respeito aos tempos de alocação. Para tal, executamos um programa composto dos seguintes passos: (1) alocação de memória para conter N nós; onde N é recebido por parâmetro; (2) realização da troca de posições dos nós forma aleatória, garantindo acesso dinâmico aos nós da lista; (3) criação de uma lista encadeada na memória alocada; (4) percorrimento da lista e, posteriormente, liberação da memória alocada.

O programa foi executado 1.000 vezes, com N aleatoriamente variando entre 40 bytes e 4 MB de memória. Assim, seriam requisitadas, no máximo, 1.024 páginas. O resultado do teste é mostrado na Figura 7. Como esperado, existe uma correlação entre o número de bytes solicitados e o número de páginas físicas alocadas (primeiro e segundo gráfico); pelos gráficos, percebemos, também, a correlação entre o número de páginas alocadas e o tempo necessário para tal (terceiro gráfico). O valor máximo observado foi de, aproximadamente, 200 ms para a requisição de 4 MB. Logo, o tempo máximo observado está relacionado ao valor máximo de memória alocada, e as demais amostras de tempo possuem um comportamento proporcional às respectivas amostras de memória alocadas. Então, em um cenário de intensas requisições de memória com quantidades aleatórias, o kernel do Linux se comporta de forma eficiente, e o programa não sofre nenhum impacto temporal ao longo do tempo devido à fragmentação ocorrida.

5. Conclusão

Neste trabalho, mostramos os resultados obtidos por meio da ferramenta Perf em relação ao desempenho de *syscalls* do Linux quando da escrita de dados em memória não volátil, operações de exclusão mútua e alocação dinâmica de memória.

De maneira pontual e conforme esperado, a escrita de arquivos é o processo que mais impacta no tempo de execução de tarefas, mesmo quando feita sobre um meio do tipo *solid-state storage*. Por outro lado, propusemos uma solução simples visando mitigar esse problema pela realização de menos *syscalls* para escrita mas com uma maior quantidade de dados. Essa investigação permitiu, também, que verificássemos que o kernel do Linux implementa caches de disco para diminuir o número e acessos à memória não volátil.

Na sincronização de tarefas, nossos testes mostraram que as *threads* podem sofrer um grande impacto temporal durante as disputas pelo mutex. Esse impacto está associado ao comportamento da interface *futex* somado ao controle de sincronismo implementado nas funções da API. De fato, as funções relacionadas à *syscall futex* são otimizadas para operar em situações de não contenção, ou seja, em casos onde não há disputa por um *mutex*. Por conseguinte, uma tarefa pode fazer o acesso a uma mesma seção crítica diversas vezes mesmo que hajam outras tarefas aguardando no mutex dessa seção. Assim, projetistas de RTAs devem utilizar mutexes com muita parcimônia, procurando, quando possível, utilizar outras estratégias que diminuam o número de seções críticas das aplicações, prin-

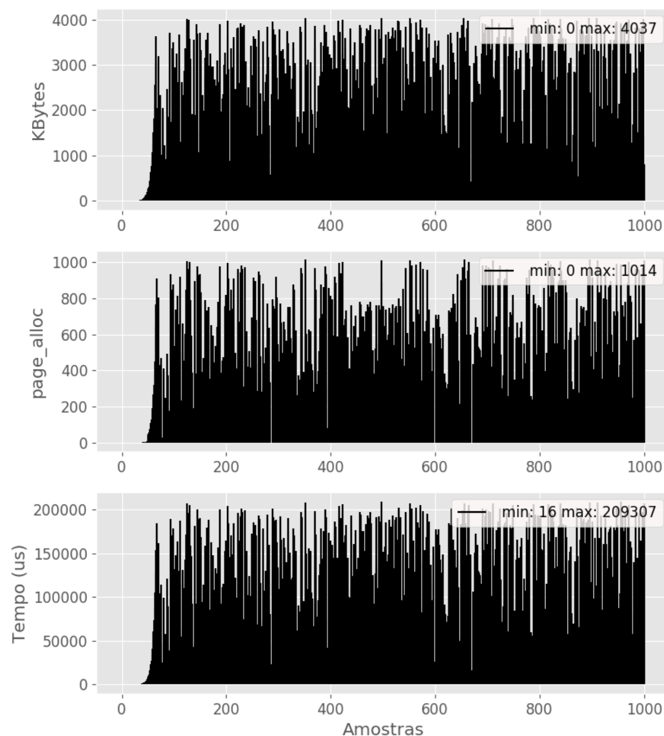

Figura 7. Gráfico com três janelas de amostras: quantidade de bytes requisitados; número de páginas alocadas; tempo consumido na execução do programa, respectivamente.

cipalmente quando as tarefas têm prioridades diferentes.

Finalmente, com relação à alocação dinâmica de memória, nossos testes mostraram que o kernel do Linux implementa um gerenciamento bastante eficiente, com técnicas de otimização que resultam em economia no uso de memória e diminuição de ausências de páginas. Mesmo em um cenário de intensas alocações de memória com valores aleatórios, as aplicações não sofrem grande impacto temporal em razão das *syscalls*.

Nossos estudos envolveram outros testes e produziram mais resultados que não foram aqui descritos por questões de espaço. De maneira geral, os resultados mostraram um não determinismo em relação ao tratamento das *syscalls* do Linux, com uma grande variação no pior tempo de execução das tarefas, o que pode comprometer o uso do Linux em ambientes híbridos, nos quais tarefas de tempo real convivem com tarefas que não são de tempo real.

Com relação ao Perf, nossos experimentos mostraram que essa ferramenta possui recursos valiosos para as depurações, observações de eventos no nível do kernel e determinação de tempos de execução com uma granularidade bastante alta. Por outro lado, nossos estudos também apontaram o *overhead* causado pela ferramenta quando de seu uso para a coleta dinâmica de eventos muito frequentes. Os resultados obtidos mostraram que os *tracepoints* dinâmicos do Perf são ineficazes para mensurar tempos de resposta de tarefas inferiores a 10 μs, na arquitetura e no processador usado em nossos experimentos. Uma análise detalhada da sobrecarga introduzida pela ferramenta pode ser

encontrada em [Weaver 2013].

Trabalhos futuros consistem em análises de custo de *syscalls* relacionadas a outros eventos, como acesso a interfaces de rede e periféricos.

Referências

Bovet, D. P. and Cesati, M. (2005). *Understanding the Linux Kernel*. Oreilly & Associates Inc.

Drepper, U. (2008). Futex are Tricky. Disponível em `http://www.ic.unicamp.br/~islene/2s2013-mo806/futex/futex-are-tricky.pdf`. Acesso em 15/06/2020.

Eranian, S., Gouriou, E., Moseley, T., and de Bruijn, W. (2015). *Linux kernel profiling with perf*. Disponível em `https://perf.wiki.kernel.org/index.php/Tutorial`. Acesso em 14/06/2020.

Kadar, M., Tverdyshev, S., and Fohler, G. (2019). System calls instrumentation for intrusion detection in embedded mixed-criticality systems. In Asplund, M. and Paulitsch, M., editors, *4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems, CERTS@ECRTS 2019, July 9, 2019, Stuttgart, Germany*, volume 73 of *OASICS*, pages 2:1–2:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Levon, J. (2004). *OProfile manual*. Disponível em `http://oprofile.sourceforge.net/doc/index.html`. Acesso em 14/06/2020.

Nabil, L. and Ben Saoud, S. (2011). Impact of the linux real-time enhancements on the system performances for multi-core intel architectures. *International Journal of Computer Applications*, 17.

Puaut, I. (2002). Real-time performance of dynamic memory allocation algorithms. In *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, pages 41–49.

Reghenzani, F., Massari, G., and Fornaciari, W. (2017). Mixed time-criticality process interferences characterization on a multicore linux system. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 427–434.

Reghenzani, F., Massari, G., and Fornaciari, W. (2019). The real-time linux kernel: A survey on preempt_rt. *ACM Comput. Surv.*, 52(1).

Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010). Collecting performance data with papi-c. In Müller, M. S., Resch, M. M., Schulz, A., and Nagel, W. E., editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg. Springer.

Weaver, V. M. (2013). Linux perf_event Features and Overhead. In *FastPath Workshop*, Austin, TX.