Implementação e Avaliação do Escalonador Agrupado para um Sistema Operacional de Tempo Real

Murilo Ferreira Vitor
Laboratório de Integração Software/Hardware (LISHA)
Universidade Federal de Santa Catarina
Florianópolis, Brasil
Email: murilo@lisha.ufsc.br

Giovani Gracioli
Laboratório de Integração Software/Hardware (LISHA)
Centro de Engenharias da Mobilidade
Universidade Federal de Santa Catarina
Joinville, Brasil
Email: giovani@lisha.ufsc.br

Antônio Augusto Fröhlich
Laboratório de Integração Software/Hardware (LISHA)
Universidade Federal de Santa Catarina
Florianópolis, Brasil
Email: guto@lisha.ufsc.br

***Resumo*—Plataformas *multicore* estão a cada dia difundindo-se mais entre os sistemas embarcados. Logo, fatores relacionados com os sobrecustos dos Sistemas Operacionais de Tempo Real (SOTR), como tempo de troca de contexto e tempo de liberação das tarefas, passam a ser determinantes na escolha de qual algoritmo de escalonamento é preferível. Por essa razão, nesse trabalho é realizada uma comparação empírica entre os escalonadores Global-EDF, Particionado-EDF e *Clustered*-EDF, considerando os sobrecustos de tempo de execução de um SOTR. Uma comparação em termos dos sobrecustos é realizada entre o SOTR com uma modificação do Linux para tempo real. Os resultados mostram que o sobrecusto em tempo de execução do Sistema Operacional (SO) tem um alto impacto na escalonabilidade das tarefas de tempo real.**

***Palavras-chave*-Sistemas operacionais de tempo real, processadores *multicore*, escalonamento de tempo real, *clustered* EDF, particionado EDF, global EDF**

I. Introdução

O desenvolvimento de algoritmos para o escalonamento de processos em plataformas *multicore* vem crescendo a cada ano, principalmente para aplicações de tempo real. As plataformas *multicore* estão sendo cada vez mais usadas em aplicações de tempo real devido às limitações impostas pela estagnação no aperfeiçoamento de processadores *single-core*. As fabricantes de chips chegaram no limite do melhoramento de processadores *single-core* devido ao superaquecimento dos chips.

Diferentemente dos sistemas em geral, os sistemas de tempo real possuem um tempo de resposta, isto é, as tarefas tem um prazo para serem cumpridas. Baseados nisso, diversos algoritmos de escalonamento foram propostos. Os principais e mais conhecidos são o *Rate Monotonic* (RM) e o *Earliest Deadline First* (EDF) [1]. Esses algoritmos apresentam algumas características que visam o cumprimento desses prazos, como por exemplo, a garantia de escalonabilidade através dos testes de escalonabilidade aplicados em um conjunto de tarefas específico. Esses algoritmos são conhecidos e bem entendidos para plataformas *single-core*. Entretanto, o problema de escalonamento em plataformas *multicore*, e consequentemente, o uso desses escalonadores, torna-se um desafio, tanto na parte de implementação devido ao compartilhamento de estruturas internas pelos diversos *cores*, como na parte teórica (testes de escalonabilidade).

Recentemente, alguns trabalhos foram propostos, principalmente avaliando questões de implementação desses algoritmos de escalonamento em plataformas *multicore* [2]–[5]. A maioria desses trabalhos avaliou a implementação utilizando uma extensão de tempo real para o Linux [2]–[4]. Neste caso, o uso de um Sistema Operacional (SO) de propósito geral é uma desvantagem devido à imprevisibilidade e maior sobrecusto em tempo de execução [5].

O *Embedded Parallel Operating System* (EPOS) é o primeiro Sistema Operacional de Tempo Real (SOTR) capaz de suportar os algoritmos de escalonamento global e particionado, que são algoritmos para plataformas *multicore* de tempo real. Dois exemplos dessas categorias suportados pelo EPOS são o *Global Earliest Deadline First* (G-EDF) e o *Partitioned Earliest Deadline First* (P-EDF) [5]. Ambas implementações foram avaliadas e comparadas a uma implementação utilizando uma extensão de tempo real do Linux, comprovando que SOTR tem um grande impacto na taxa de escalonabilidade das tarefas de tempo real [5]. Entretanto, ainda é possível implementar outras classes de algoritmos de escalonamento *multicore* no EPOS, expandindo assim o seu suporte de tempo real.

A fim de expandir o suporte de tempo real no EPOS, este trabalho apresenta o projeto e a implementação de um algoritmo de escalonamento tempo real agrupado (*clustered*) no EPOS. Mais especificamente, o foco da implementação é o *Clustered-EDF* (C-EDF) [3]. O EDF foi escolhido por apresentar algumas vantagens em relação a políticas de escalonamento estáticas (e.g., RM, *deadline monotonic*), como por exemplo, uma maior taxa de utilização do processador [1]. O projeto e implementação é avaliado em termos do sobrecusto em tempo de execução e comparado a uma implementação usando uma extensão de tempo real para o Linux. Além disso, os sobrecustos medidos são adicionados à análise de escalonabilidade do C-EDF e comparados a taxa de escalonabilidade do P-EDF e G-EDF publicadas anteriormente [5]. A avaliação mostra a influência do SOTR na escalonabilidade das tarefas de tempo real, variando-se os escalonadores.

O restante deste artigo é organizado da seguinte maneira. A Seção II apresenta os principais conceitos básicos e o modelo do sistema utilizados neste trabalho. A Seção III apresenta o projeto e implementação do escalonador C-EDF no EPOS, mostrando como ele foi estendido a partir do projeto de escalonamento *multicore* anteriormente proposto no EPOS. A comparação entre as implementações do G-EDF, P-EDF e C-EDF no EPOS e no LITMUS$^{\mathrm{RT}}$ é apresentada na Seção IV. Por fim, a Seção V conclui o artigo.

II. Modelo do Sistema e Conceitos Básicos

Nesse trabalho é considerado um modelo formado por tarefas periódicas, onde um conjunto de tarefas τ é composto de *n* tarefas, $\{T_1, T_2, \ldots, T_n\}$. As *n* tarefas são escalonadas em *m* processadores iguais $\{P_1, P_2, \ldots, P_m\}$. Cada tarefa T_i, onde $i \leq n$ e $i \geq 1$, tem um período p_i e um pior tempo de execução (WCET) e_i. Uma tarefa T_i libera um trabalho a cada unidade de tempo p_i. r_i^j representa o tempo de liberação do *job* j^{th} da T_i, denominada T_i^j. O *deadline* relativo da tarefa T_i é igual ao seu período: $d_i = p_i$. A relação e_i/p_i define a utilização da tarefa T_i, denominada u_i. A soma de todas as utilizações das tarefas definem a utilização total do sistema.

A. Sobrecustos dos SOTR

Os SO possuem fontes de sobrecustos (*overheads*) oriundas da execução de suas atividades [5]. As principais fontes de sobrecustos encontradas em SOTR são descritas a seguir.

Quando uma tarefa é liberada, ocorre o sobrecusto de liberação, que é o tempo necessário para atender a rotina de interrupção responsável pela liberação das tarefas no tempo correto. Sempre que uma decisão de escalonamento é tomada, o sobrecusto de escalonamento ocorre, sendo responsável por selecionar o próximo processo a ser executado e reordenar a fila de processos. O sobrecusto de troca de contexto ocorre quando troca-se a pilha de execução e os registradores de processos. Latência de interrupção entre processadores (IPI) ocorre quando uma tarefa é liberada em um processador diferente da qual a tarefa foi escalonada. Finalmente, o sobrecusto de contagem de *tick* é o tempo necessário para gerenciar o temporizador de interrupções para escalonamento periódico.

Em sistemas de tempo real é importante conhecer todos os sobrecustos introduzidos pelo SO, a fim de garantir o cumprimento dos requisitos de tempo real do sistema.

B. Algoritmos de escalonamento para plataformas multicore de tempo real

Os algoritmos de escalonamento são normalmente classificados em duas categorias: particionada e global [6]. Na abordagem particionada, cada tarefa é atribuída estaticamente para cada processador. As migrações entre os processadores não são permitidas. Para cada processador existe uma fila de processos exclusiva a ele [7]. Na abordagem global, as tarefas podem migrar livremente entre os processadores, podendo assim, serem executadas em qualquer um deles. Isso é possível graças a sua fila de processos, que é única para todos os processadores. Na abordagem agrupada (*clustered*) as tarefas são atribuídas estaticamente para um grupo de processadores, formando assim, agrupamentos de processos, semelhante ao particionado [8]. Em cada agrupamento, as tarefas são livres para migrar entre os processadores, assim como ocorre no global. É possível notar que tanto o particionado, quanto o global, são particularidades do agrupado, onde no particionado, o agrupamento de processadores é formado por apenas um processador, e no global, o número de agrupamentos, se resume a apenas um grupo, formado por todos os processadores.

Os escalonadores *Global*-EDF (G-EDF), *Partitioned*-EDF (P-EDF), e *Clustered*-EDF (C-EDF), são exemplos de cada categoria descrita acima. O G-EDF possui um sobrecusto de tempo de execução razoável para um número moderado de processadores [3]. Além disso, o G-EDF é interessante para aplicações *Soft-Real Time* (SRT), pois mesmo tendo uma utilização total do sistema maior do que *m* (sendo que *m* representa o número de processadores), ele consegue garantir limites para o atraso na execução das tarefas [9]. Por outro lado, o P-EDF tem uma limitação devido ao problema de *bin-packing*, onde tarefas com alta taxa de utilização[1], afetam a heurística de particionamento das tarefas. No C-EDF, todos os processadores que compartilham um nível de *cache* (L2 e L3) formam um *cluster*. O agrupamento diminui o sobrecusto de tempo de execução, quando comparado com o G-EDF, e tem menos problemas de *bin-packing*, quando comparado com o P-EDF [3]. P-EDF é superior ao G-EDF e ao C-EDF para aplicações *Hard-Real Time* (HRT) [5]. O EPOS possui atualmente suporte para o G-EDF e P-EDF. Este trabalho visa implementar e avaliar o C-EDF no EPOS.

C. LITMUS$^{\mathrm{RT}}$

O *Linux Testbed for Multiprocessor Scheduling in Real-Time Systems* (LITMUS$^{\mathrm{RT}}$) é uma extensão do *kernel* do Linux para tempo real, com foco em escalonamento de tempo real para plataformas *multicore*. O *kernel* do Linux foi modificado para suportar o modelo de tarefas esporádicas, além de suportar os escalonadores particionado, agrupado e global. Os escalonadores são baseados principalmente no EDF. O LITMUS$^{\mathrm{RT}}$possui algumas ferramentas, uma delas é o *Feather-Trace*, que é usado para coletar os sobrecustos encontrados no LITMUS$^{\mathrm{RT}}$ [10], auxiliando assim, a análise de tarefas de tempo real em plataformas *multicore*.

III. Projeto e Implementação

O escalonador agrupado foi projetado e implementado no *Embedded Parallel Operating System* (EPOS) [11] [12]. O EPOS é um *framework* de sistemas embarcados, orientado a objetos e baseado em componentes. Componentes independentes do EPOS, implementam serviços tradicionais do SO, como *threads* (tarefas) e semáforos. Nos últimos anos, o EPOS tem sido usado em várias pesquisas acadêmicas e no desenvolvimento de alguns projetos industriais, como redes de sensores sem fio [13] e TV digital [14].

A. Escalonamento

O suporte de escalonamento no EPOS é formado por componentes que realizam todas as funções necessárias para para o escalonamento das tarefas. Tais componentes implementam, por exemplo, os conceitos tradicionais de tarefas periódicas,

[1] Uma tarefa tem uma alta utilização quando u_i é maior que 0.5

escalonadores e sincronização entre tarefas. No EPOS, um componente é uma classe escrita em C++ com interface e comportamento bem definidos.

A classe `Thread` representa uma tarefa aperiódica, definindo o seu fluxo de execução, com seu próprio contexto e pilha. A classe implementa funções tradicionais das *threads*, como as operações de suspender, resumir, dormir, e acordar. Há também a classe `Periodic_Thread` que estende a classe `Thread` para permitir o suporte de tarefas periódicas no EPOS, adicionando mecanismos relacionados à re-execução das tarefas periódicas, como o método `wait_next`, que obriga a tarefa a dormir no tempo referente ao seu período, e também a classe `Alarm` que é responsável por liberar e acordar uma tarefa. O construtor da `Periodic_Thread` cria o `Alarm` relacionado com a tarefa periódica.

A classe `Scheduler` e a classe `Criterion`, mostradas na Figura 1, definem a estrutura que realiza o escalonamento das tarefas. O EPOS promove a reutilização de código, separando a política de escalonamento (representada pela classe `Criterion`) do seu mecanismo (e.g., as implementações de estrutura de dados como listas e pilhas). A ordenação das tarefas é realizada pela estrutura de dados presente na classe `Scheduler`, baseada no critério de escalonamento.

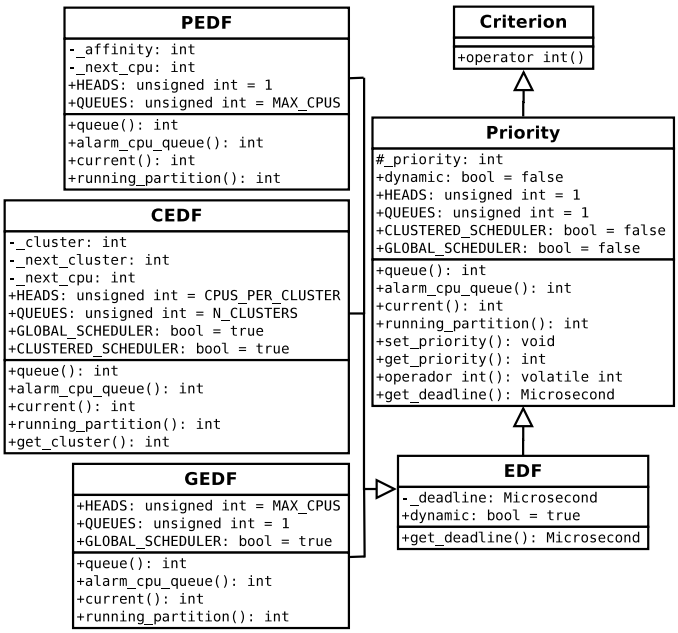

Figura 1. Diagrama de classes da classe `Criterion`

A classe `Trait`[2] define o critério de escalonamento em tempo de compilação (e.g., `typedef Scheduling_Criteria::PEDF Criterion` define o critério de escalonamento como P-EDF). O `Scheduler` consulta as informações fornecidas pela classe `Criterion` para definir o uso adequado das listas e operações. Cada classe `Criterion` define basicamente a prioridade de uma tarefa, que é mais tarde usada pelo escalonador para escolher qual tarefa será executada, além de outras funcionalidades, como por exemplo, definir se o critério de escalonamento é preemptivo ou não. O `Scheduler` recebe um componente como parâmetro (`Thread` nesse caso), e é esse componente que define um `Criterion`, que tem as características como número de filas, preempção, etc. O `Scheduler` usa uma `Scheduling_Queue` para realizar as operações de escalonamento, como mudar, inserir, e remover uma *thread*.

O critério G-EDF é uma extensão do EDF e é suportado pelo EPOS. Uma *flag* chamada `GLOBAL_SCHEDULER` é usada para informar ao escalonador quando o critério é global ou não. Se o critério é global, o escalonador usa uma lista especializada através da classe `Scheduling_Queue`, com apenas uma lista global para o sistema, já que cada tarefa que está sendo escalonada está fora da fila. O P-EDF funciona de maneira diferente, pois existe uma fila por processador, sendo assim, a `Scheduling_Queue` usa uma lista por processador e a tarefa que está sendo executada é a cabeça da lista.

O P-EDF e o G-EDF possuem duas variáveis em suas definições, denominadas `HEADS` e `QUEUES`. Essas variáveis são usadas de modo a especificar que tipo de lista será usada pela `Scheduling_Queue`. O G-EDF, define a variável `QUEUES` com o valor um e a variável `HEADS` com o número máximo de processadores presentes na máquina. Com isso, o `Scheduler` é capaz de escolher uma especialização de lista, definida como `Scheduling_List`, que será usada pela `Scheduling_Queue`, onde será criada apenas uma fila para todo o sistema. Essa especialização da `Scheduling_List` possui mecanismos capazes de identificar em qual processador cada processo está sendo executado, permitindo assim, que o `Scheduler` possa inserir e/ou remover os processos da maneira correta. Já no caso do P-EDF, a variável `HEADS` é definida com o valor um e a variável `QUEUES` com o número máximo de processadores presentes na máquina. Dessa forma, o `Scheduler` é capaz de escolher, qual especialização de lista será usada pela `Scheduling_Queue`. No caso do P-EDF, será criada uma fila para cada processador. O método `Criterion::Current` (veja Figura 1) retorna o processador que está executando a tarefa. Deste modo, o escalonador é capaz de manipular corretamente a inserção e remoção de/para uma fila específica, através da `Scheduling_List`. Este é o mesmo mecanismo usado em outros escalonadores particionados que são suportados pelo EPOS, como o *CPU_Affinity* e o *Partitioned Rate Monotonic* (P-RM). Todas as dependências entre o `Criterion` e o `Scheduler` são resolvidas em tempo de compilação, não gerando assim, sobrecusto em tempo de execução.

B. C-EDF

Com a separação entre escalonador, critério, e tarefa (seção III-A), torna-se simples a inserção de novas políticas de escalonamento. Baseado nisso, este trabalho estende o critério de escalonamento EDF para suportar o escalonador C-EDF.

Na implementação do C-EDF, foram criadas duas variáveis para permitir que as particularidades referentes ao agrupamento das tarefas fossem atendidas. A primeira delas foi a variável `N_CLUSTERS`, pertencente à classe `Traits`, e que representa o número de *clusters* que serão usados na aplicação, associando seu valor à variável `QUEUES`. Baseado nisso, o escalonador pode utilizar a `Scheduling_Queue` da mesma forma que o P-EDF, podendo assim, criar uma

[2]A classe `Trait` é uma classe *template* que associa informação de um componente em tempo de compilação.

fila para cada *cluster*. A segunda variável foi chamada de `CPUS_PER_CLUSTER`, também pertencente à classe `Traits`, e que representa o número de processadores por *cluster*, associando seu valor a variável `HEADS`. Para que o escalonamento dentro de cada *cluster* seja realizado como no escalonamento global, a *flag* `GLOBAL_SCHEDULER` foi definida como *true*. Baseado nos valores das variáveis `HEADS` e `GLOBAL_SCHEDULER`, a classe `Scheduler` é capaz de utilizar a `Scheduling_List` para o escalonamento global, e assim, inserir e/ou remover os processos dentro de cada *cluster*. Um exemplo da interação entre o C-EDF, `Scheduling_Queue`, e `Scheduler` é mostrada na Figura 2. Nota-se as definições das variáveis `QUEUES` (linha 5) e `HEADS` (linha 6), além do método `running_partition` (linha 12) que é usado para identificar qual processador está executando a tarefa. No P-EDF, o método `running_partition` (veja Figura 1), retorna a CPU que está sendo executada, correspondente a fila atual. O G-EDF funciona diferente, o método `running_partition` retorna o valor zero, já que existe apenas uma fila para todo o sistema.

```
 1 // Criterion
 2 class CEDF: public EDF {
 3   public:
 4     static const unsigned int QUEUES = Traits<Thread>::
           N_CLUSTERS;
 5     static const unsigned int HEADS = Traits<Thread>::
           CPUS_PER_CLUSTER;
 6     static const bool GLOBAL_SCHEDULER = true;
 7    ...
 8   public:
 9     CEDF(int p, int c): EDF(p), _cluster(c) {}
10    ...
11    static int running_partition () {
12     return Machine::cpu_id() / HEADS;
13   }
14    ...
15   private:
16     volatile int _cluster;
17    ...
18 };
19 template <typename T, unsigned int Q>
20 class Scheduling_Queue<T, Q> {
21     typedef typename T::Criterion Criterion;
22     typedef Scheduling_List<T, Criterion> Queue;
23   public:
24     typedef typename Queue::Element Element;
25      ...
26     Element * choose() { return _ready[ Criterion ::
           running_partition () ].choose(); }
27      ...
28   private:
29     Queue _ready[Q];
30 };
31 template <typename T>
32 class Scheduler: public Scheduling_Queue<T, T::Criterion::QUEUES
       , T::Criterion::GLOBAL> {
33 typedef Scheduling_Queue<T, T::Criterion::QUEUES, T::Criterion::
       GLOBAL> Base;
34    .....
35   T * choose() { return Base::choose()->object(); }
36    .....
37 };
```

Figura 2. Um exemplo da interação entre as classes `Criterion`, `Trait`, e `Scheduler`

IV. AVALIAÇÃO

Esta seção primeiramente mostra a comparação entre o G-EDF, o P-EDF, e o C-EDF em termos de sobrecusto do SO. O sobrecusto de cada escalonador é medido tanto no EPOS quanto no LITMUS^{RT}[3]. Após, os sobrecustos medidos são usados para avaliar a taxa de escalonabilidade de tarefas dos três escalonadores.

A. Descrição do Experimento

Para medir os sobrecustos associados as atividades do SO, o número de tarefas foi fixado em 5, 15, 25, 50, 75, 100, e 125. Para gerar os períodos das tarefas (em ms), foi usada uma distribuição uniforme entre [3, 33], e para gerar as utilizações das tarefas, foi usada uma distribuição uniforme entre [0,001, 0,1]. A partir dos períodos e utilizações das tarefas, os WCET de cada tarefa (antes de adicionar os sobrecustos do SO) foram definidos.

Com o objetivo de estafar os SO, o mesmo código da função de tarefas é usado no EPOS e no LITMUS^{RT}. Essa função de tarefas soma uma variável local em um laço de 50 repetições para cada tarefa, e partir disso, é medido o sobrecusto de tempo de execução. Cada conjunto de tarefas foi executado por 100 vezes em um processador Intel i7-2600[4] para o EPOS e para o LITMUS^{RT}, e foram extraídos as amostras do WCET para cada sobrecusto dessas execuções.

B. Sobrecusto de tempo de execução

Os sobrecustos de troca de contexto, liberação, contagem de *tick*, e escalonamento foram medidos no EPOS e no LITMUS^{RT}. As Figuras 3-6 mostram os valores obtidos para o EPOS e para o LITMUS^{RT}. O eixo-*x* mostra o número de *threads* e o eixo-*y* mostra o pior tempo de execução em μs. Para medir os sobrecustos, foi usado o *Time-Stamp Counter* (TSC) [5] para o EPOS, e no LITMUS^{RT}, foi usado o *Feather-Trace* [10]. Foram removidos de 2 a 8 outliers nos valores medidos para o LITMUS, principalmente para reduzir os efeitos da imprevisibilidade do Linux nos resultados. Essa abordagem de remoção de outliers é diferente dos trabalhos relacionados [3], onde são removidos 1% dos valores mensurados mais altos. Isso tem um impacto importante na comparação dos resultados, tornando-a mais justa para sistemas HRT.

1) Sobrecusto de troca de contexto: Para o EPOS, configurou-se um caso de teste composto por duas *threads* *a* e *b*. A *thread a* inicia o TSC antes da troca de contexto, e a *thread b* lê a TSC e calcula a diferença, resultando no tempo total da troca de contexto para *thread a*. A troca de contexto é realizada na mesma CPU. Assim, pode-se isolar o exato momento em que uma troca de contexto acontece. Para o LITMUS^{RT}, a medição do sobrecusto de troca de contexto foi realizada usando o *Feather-Trace* e um conjunto de tarefas executando com um número fixo de tarefas, como apresentado anteriormente. A Figura 3 mostra que há uma alta variação do tempo de execução, de 9,4 μs para 29,56 μs no G-EDF, de 2 μs para 26,95 μs para o P-EDF e de 12,3 μs para 37,4 μs para o C-EDF. Esta alta variação pode ser devido a alguns fatores, como por exemplo, os aspectos não determinísticos do Linux [15]. Este resultado reforça a teoria de que o Linux sofre interferência de outras partes do sistema, prejudicando a previsibilidade necessária, principalmente, em sistemas HRT.

[3] Versão 2012.1, kernel 3.0

[4] Processador Intel i7-2600 com 4 Processadores, *Hyper-Threading* (SMT) com 2 por processador (8 processadores lógicos) e cache L3 com 8 MB

2) Sobrecusto de escalonamento: O sobrecusto de escalonamento foi medido no EPOS e no LITMUS^{RT} usando a metodologia descrita na Seção IV-A. Na Figura 4 observa-se que o sobrecusto de escalonamento do EPOS aumenta após 75 *threads*, principalmente para o escalonador G-EDF. A responsável por isso é a lista de escalonamento, que demora mais tempo para realizar as operações de inserção e remoção, já que a há mais *threads* na lista, que é única para todo o sistema. Diferentemente, o P-EDF do EPOS não apresenta uma variação considerável no sobrecusto de escalonamento, isso é explicado pelo fato das *threads* serem distribuídas entre os processadores, e cada processador ter a sua lista de escalonamento. O C-EDF do EPOS tem um comportamento muito semelhante ao do P-EDF do EPOS, onde devido aos agrupamentos, nesse caso formado por dois processadores, tem uma lista de escalonamento maior que a do P-EDF, mas bem inferior à do G-EDF, fazendo com que ele tenha uma variação no sobrecusto de escalonamento próxima à do P-EDF. Para o LITMUS^{RT}, como o número de *threads* aumenta, a imprevisibilidade do Linux também aumenta, por essa razão os sobrecustos de escalonamento do G-EDF e P-EDF são tão altos a partir de 75 *threads*. O C-EDF do LITMUS^{RT} conseguiu reduzir essa imprevisibilidade mantendo-se praticamente constante.

3) Sobrecusto de liberação das threads*:* O sobrecusto de liberação das *threads* foi medido no EPOS e no LITMUS^{RT} usando a metodologia descrita na Seção IV-A. O método `Alarm handler` realiza todas as operações relacionadas ao sobrecusto de liberação das *threads* no EPOS [5]. Por essa razão, os sobrecustos do P-EDF, G-EDF e C-EDF não foram medidos separados, pois as operações de liberação das *threads* são as mesmas para os três escalonadores.

A Figura 5 mostra que no EPOS o pior tempo de execução aumenta de acordo com o número de *threads* no sistema, isso porque, o `Alarm handler` libera mais *threads* em uma mesma manipulação, afetando assim, o sobrecusto. O escalonador G-EDF do LITMUS^{RT} tem um comportamento similar, enquanto que o escalonador P-EDF não apresenta uma variação considerável depois de 50 *threads*. Já o C-EDF do LITMUS^{RT}, é uma mistura entre o G-EDF e P-EDF do LITMUS^{RT}, onde aumenta de acordo com o número de *threads* no sistema, mas não apresenta uma variação considerável depois de 100 *threads*.

4) Sobrecusto de contagem de tick*:* O sobrecusto de contagem de *tick* foi medido no EPOS e no LITMUS^{RT} usando a metodologia descrita na Seção IV-A. Como dito anteriormente, o Linux possui uma imprevisibilidade associada ao aumento no número de *threads*, contribuindo dessa forma com o aumento no sobrecusto de contagem de *tick*. Outra possível causa pode ser relacionada ao mecanismo de tratamento de interrupções, onde alguma outra atividade do sistema pode ter sido executada, influenciando assim, no sobrecusto de contagem de *tick*. Isso justifica o fato de que após 75 *threads* (Figura 6), o G-EDF e o P-EDF passam a ter um aumento considerável no sobrecusto de contagem de *tick*. Já o C-EDF permanece praticamente constante até 50 *threads*, e depois tem um pequeno aumento se comparado com o P-EDF e G-EDF. No EPOS, o sobrecusto de contagem de *tick* é constante, graças ao seu método `Alarm handler`, que acessa o registrador adequado da CPU ID (maiores detalhes são encontrados em [5]), dessa forma o sobrecusto de contagem de *tick* não depende do número de *threads*.

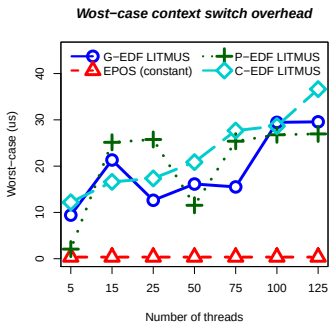

Figura 3. Sobrecusto de troca de contexto para o pior tempo de execução.

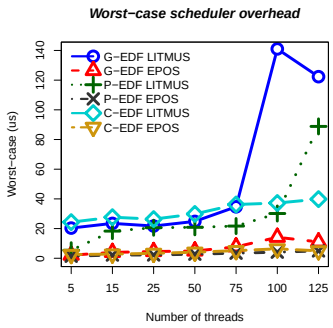

Figura 4. Sobrecusto de escalonamento para o pior tempo de execução.

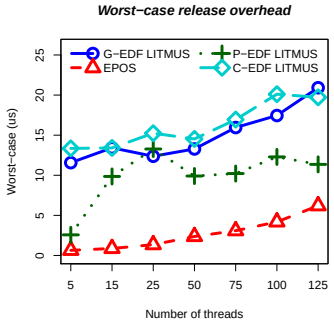

Figura 5. Sobrecusto de liberação das *threads* para o pior tempo de execução.

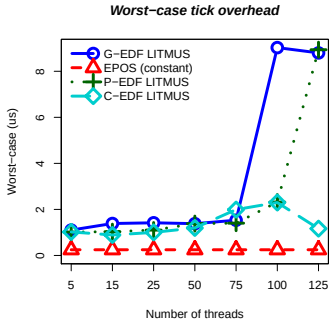

Figura 6. Sobrecusto de contagem de *tick* para o pior tempo de execução.

C. Taxa de Escalonabilidade

Para determinar a taxa de escalonabilidade das tarefas considerando o sobrecusto do SO, o WCET de cada tarefa foi inflado com os tempos apresentados nas Subseções anteriores usando o método de contagem de interrupção centrado em preempção [5, 16]. Neste método, cada fonte de sobrecusto do sistema é corretamente incorporada à análise de escalonabilidade. Para maiores detalhes sobre o método, consulte os trabalhos relacionados [5, 16]. Foram gerados 1000 conjuntos de tarefas para cada utilização, variando-a entre 2, 2.1 até 8, com as mesmas distribuições uniformes apresentadas na Seção IV-A. Para verificar a taxa de escalonabilidade do G-EDF, foram executados oito testes de escalonabilidade suficientes, conforme descrito em [5]. Um conjunto de tarefas é considerado escalonável se passa em pelo menos um dos oito testes. Para o P-EDF, foram usadas as heurísticas de particionamento *first-fit*, *worst-fit* e *best-fit*. Um conjunto de tarefas é considerado escalonável se pelo menos uma das heurísticas de particionamento consegue particionar as tarefas. Para o C-EDF, as tarefas foram primeiramente particionadas usando as mesmas três heurísticas do P-EDF e posteriormente os mesmos oito testes de escalonabilidade do G-EDF em cada partição. Um conjunto de tarefas é escalonável pelo C-EDF se pelo menos uma heurística consegue particionar as tarefas e se

pelo menos um testes de escalonabilidade do G-EDF retorna verdadeiro.

A Figura 7 apresenta a taxa de escalonabilidade dos três algoritmos usando a metodologia descrita acima. Nota-se que os algoritmos particionados, P-EDF e C-EDF, obtiveram melhores desempenhos para tempo real crítico. Isso é devido ao extremo pessimismo dos testes de escalonabilidade do G-EDF. Nota-se também que o G-EDF considerando o sobrecusto do EPOS possui uma melhor taxa de escalonabilidade quando comparado aos algoritmos com o sobrecusto do LITMUS$^{\mathrm{RT}}$. O P-EDF com o sobrecusto do EPOS teve desempenho similar ao C-EDF sem nenhum sobrecusto. Com este experimento, é possível concluir que o sobrecusto do SOTR tem um grande impacto na taxa de escalonabilidade das tarefas de tempo real críticas. O EPOS, por ser um SOTR sem enfoque a aplicações de propósito geral como Linux, apresentou baixo valores de sobrecusto de tempo de execução e, consequentemente, uma melhor taxa de escalonabilidade do que o LITMUS$^{\mathrm{RT}}$.

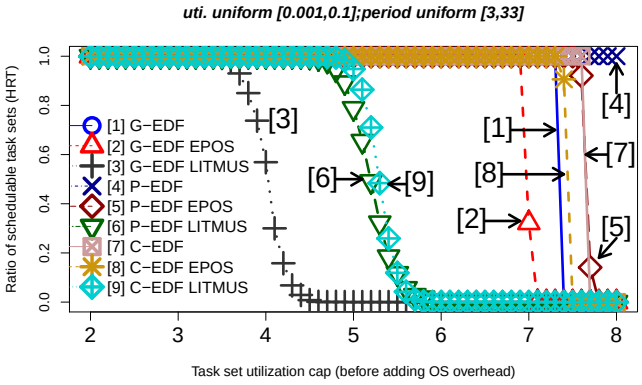

Figura 7. Taxa de escalonabilidade do P-EDF, C-EDF e G-EDF considerando o sobrecusto de tempo de execução do EPOS e do LITMUS$^{\mathrm{RT}}$.

V. Conclusão

Este trabalho apresentou o projeto e implementação do escalonador C-EDF no EPOS, além da comparação empírica em termos do sobrecusto do SOTR e taxa de escalonabilidade das implementações do G-EDF, P-EDF e C-EDF no EPOS e no LITMUS$^{\mathrm{RT}}$. O projeto e implementação do C-EDF comprovou que é relativamente simples implementar novos escalonadores no EPOS, devido a sua estrutura ser orientada a objetos e baseada em componentes.

A avaliação mostrou que o EPOS, por ser um SOTR sem enfoque a aplicações de propósito geral como Linux, possui baixos valores de sobrecusto de tempo de execução, obtendo assim, melhor desempenho que o LITMUS$^{\mathrm{RT}}$. Isso foi provado nos testes de escalonabilidade, onde o G-EDF do EPOS, que foi o escalonador com menor taxa de escalonabilidade em comparação ao P-EDF e C-EDF do EPOS, se mostrou com melhor desempenho que todos os escalonadores do LITMUS$^{\mathrm{RT}}$. Além disso, o C-EDF e P-EDF, obtiveram melhores desempenhos para tempo real crítico. Sendo que o P-EDF com o sobrecusto do EPOS teve desempenho similar ao C-EDF sem nenhum sobrecusto, e o C-EDF com o sobrecusto do EPOS, teve um desempenho melhor que o G-EDF sem nenhum sobrecusto. Logo, percebemos que o P-EDF do EPOS é uma ótima alternativa para sistemas HRT, assim como o C-EDF do EPOS, mesmo quando são considerados os sobrecustos de tempo de execução.

Como trabalhos futuros, o EPOS será portado para um processador Arm Cortex-A9 *dual-core*, o que permitirá a implementação do controle de um quadcóptero usando essa plataforma, juntamente com os escalonadores desenvolvidos neste trabalho.

Referências

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[2] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "Litmusrt: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, ser. RTSS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 111–126.

[3] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers," in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, ser. RTSS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 14–24.

[4] ——, "Is semi-partitioned scheduling practical?" in *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*, ser. ECRTS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 125–135.

[5] G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, "Implementation and evaluation of global and partitioned scheduling in a real-time OS," *Real-Time Systems*, 2013.

[6] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[7] J. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[8] J. Calandrino, J. Anderson, and D. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *19th Euromicro Conference on Real-Time Systems, 2007. ECRTS '07.*, 2007, pp. 247–258.

[9] H. Leontyev and J. H. Anderson, "Generalized tardiness bounds for global multiprocessor scheduling," in *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 413–422.

[10] B. B. Brandenburg and J. H. Anderson, "Feather-trace: A light-weight event tracing toolkit," in *In Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'07*, 2007, pp. 61–70.

[11] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[12] EPOS. (2013, Sep.) Epos website. [Online]. Available: http://epos.lisha.ufsc.br

[13] L. F. Wanner and A. A. Fröhlich, "Operating System Support for Wireless Sensor Networks," *Journal of Computer Science*, vol. 4, no. 4, pp. 272–281, 2008.

[14] M. K. Ludwich and A. A. Fröhlich, "Abstracting hardware devices to embedded Java applications," in *IADIS Applied Computing 2011*, Rio de Janeiro, Brazil, Nov. 2011, pp. 371–378.

[15] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *Proceedings of the 2008 Real-Time Systems Symposium*, ser. RTSS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 157–169.

[16] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.