EDUARDO STEINER

GERENCIAMENTO DINÂMICO DE ENERGIA EM SISTEMAS EMBARCARCADOS DE TEMPO REAL

Florianópolis

Dezembro de 2009

EDUARDO STEINER

GERENCIAMENTO DINÂMICO DE ENERGIA EM SISTEMAS EMBARCARCADOS DE TEMPO REAL

Rascunho do relatório do Trabalho de Conclusão de Curso

Orientador:

Profº Dr. Antônio Augusto Fröhlich

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Florianópolis

Dezembro de 2009

Título: Gerenciamento Dinâmico de Energia em Sistemas Embarcados de Tempo Real

Autor: Eduardo Steiner

Banca Examinadora:

Prof. Dr. Antônio Augusto Fröhlich
Orientador

M.sc Geovani Ricardo Wiedenhoft

M.sc Giovani Gracioli

RESUMO

Sistemas embarcados são sistemas computacionais que geralmente são dedicados à uma aplicação específica. Seus projetos são caracterizados por possuírem um grande número de restrições como tempo real, confiabilidade, eficiência energética, etc.

Este trabalho apresenta uma proposta de um escalonador de tempo real para sistemas embarcados cujo a energia é fornecida por bateias. Esse escalonador deverá ter noção dos níveis de energia nas baterias do sistema e fazer com que o tempo de duração definido para ele seja alcançado. O principal foco do trabalho, é fazer com que as tarefas que não possuam requisitos de tempo real tenham a melhor QoS possível, desde que isso não entre em conflito com o escalonamento das tarefas de tempo real, e não prejudique o tempo que o sistema tem que durar.

O escalonador proposto foi criado no sistema EPOS(Embedded Paralell Operating System), e faz uso de suas estruturas de escalonamento de tempo real e gerenciamento de energia.

ABSTRACT

Embedded systems are computing systems usually dedicated to a specific application. Embedded systems projects are characterized by the great number of constraints such as real time, reliability, energectic eficiency, etc.

This work presents a power aware real time scheduler for embedded system. This scheduler must have knowledge about the energy levels in the system battery, and must make the system reach the time defined for it's duration. The main focus in this work is to provide the greatest QoS possible to non-real-time tasks, since it doesn't get conflict with the scheduling of real-time tasks and doesn't affect the setted duration the system must reach.

The proposed scheduler was created in EPOS system (Embedded Paralell Operating System), and uses its real-time scheduling and power management structures.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

O crescente número de sistemas computacionais presentes no dia a dia dos indivíduos da sociedade é um fato levantado por muitos pesquisadores (Marwedel, 2003; Weiser, 2001). Estudos (Marwedel, 2003) mostram que um americano comum entra em contato com uma média de 60 processadores diariamente. A maior parte desses processadores não se encontra em computadores de propósito geral, mas sim em sistemas embarcados.

Sistemas embarcados são sistemas computacionais, que se caracterizam por estar completamente embarcados em um produto maior. Geralmente possuem restrições temporais (em alguns casos críticas), de eficiência computacional e energética. Sistemas embarcados estão presentes em carros, televisões, máquinas de lavar e em uma infinidade de dispositivos; os quais entramos em contato todos os dias. Seguindo o sucesso da tecnologia da informação para aplicações de escritórios e fluxo de trabalho, a área de sistemas embarcados está sendo considerada a mais importante da teclonogia da informação dos próximos anos (Marwedel, 2003).

De acordo com Eggermont (Eggermont, 2002) a energia é considerada a restrição mais importante em sistemas embarcados. No caso de sistemas cujo a energia provém de baterias, essa importancia se torna ainda maior uma vez que isso implicará diretamente no tempo de vida do sistema. Desse modo, é possível ver a grande importância que o gerenciamento de energia tem, no projeto de sistemas embarcados.

1.1 MOTIVAÇÃO

Dada a relevancia da economia energética no projeto de sistemas embarcados, nota-se que a proposta de novas abordagens para o gerenciamento de energia é muito importante - motivo o qual fez com que um crescente número de cientistas da industria e academia tenham cada vez mais pesquisado sobre o assunto.

Apesar disso, essa área ainda reune muitos desafios. O gerenciamento de energia eficaz em sistemas embarcados não é algo simples de se desenvolver, pois deve-se levar em conta diversos itens como por exemplo implicações em performance que modos de consumo mais

baixos possam causar, overheads associados à trocas de modos de operações, influências que diferentes técnicas exercem uma sobre a outra entre outros.

O *hardware* de sistemas embarcados costuma reunir caracteristicas de eficiência e economia energética, porém elas ainda não são usadas de modo ideal pelo software de gerencia de energia. Desse modo, as novas abordagens para a gerência de energia por software devem ser propostas e estudadas afim de se alcançar um melhor uso dessas características.

1.2 OBJETIVO

O principal objetivo deste trabalho é a implementação de um escalonador sistemas embarcados de tempo real, o qual tenha noção dos níveis de energia presente nas baterias do sistema.

Como a definição de um tempo mínimo o qual o sistema deverá durar é uma pratica muito comum no projeto de sistemas embarcados, um dos objetivos do escalonador proposto será fazer com que o tempo de duração atribuído ao sistema seja alcançado.

Uma vez que diferentes tarefas do sistema têm diferentes requisitos (de tempo real, importância para a aplicação, etc), a abordagem proposta irá classificar as tarefas em dois grupos: *hard real-time*(tempo real rígido) e *best-effort* (melhor esforço).

Nesse contexto, os seguintes requisitos foram levantados:

- Tarefas *hard real-time* deverão sempre ter seus prazos respeitados e prioridade de execução sobre as tarefas *best-effort*.
- O escalonador deverá fazer com que o tempo estipulado para o sistema pelo programador da aplicação seja alcançado. Caso seja constatado que o nível de energia nas baterias não seja o suficiente para isso, ele deverá efetuar o descarte das tarefas *best-effort*, e escalonar tarefas energeticamente mais economicas em seu lugar.
- Nenhuma tarefa *best-effort* deve ficar ”faminta”de recursos, todas as tarefas devem ter a sua chance na CPU (desde que isso não entre em conflito com os itens anteriores).
- A QoS(Qualidade de Serviço) atribuída as tarefas *best-effort* deverá ser proporcional à carga de energia presente nas baterias do sistema em relação à quantidade de tempo que falta para se alcançar o tempo estipulado para a duração do sistema.

1.3 ESTRUTURA DO TEXTO

O restante do texto é organizado da seginte forma:

- O capítulo 2 apresenta o resultado de um estudo envolvendo os conceitos e o estado da arte das principais áreas de estudo relacionados ao trabalho: sistemas embarcados, sistemas de tempo real e gerenciamento de energia.
- O capitulo 3 apresenta a abordagem proposta, e mostra como os mecanismos implementados para se alcançar os objetivos propostos.
- O capitulo 4 descreve um estudo de caso, e apresenta a análise dos resultados.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 SISTEMAS EMBARCADOS

Sistemas embarcados são sistemas computacionais, porém de grande contraste com sistemas de propósito geral. Primeiramente, eles são caracterizados por possuírem aplicação ou propósito específico. Também costumam ter capacidade de processamento e memória bastante limitada, exigindo dos projetistas e programadores das aplicações máxima atenção e conhecimento sobre os detalhes das tecnologias envolvidas nos sistemas. Além disso, possuem uma série de restrições de projeto, entre as quais podem ser citadas: baixo consumo de energia, segurança, confiabilidade.

A Figura 1 (Marwedel, 2003) mostra um *loop*, no qual está contextualizado a grande parte dos sistemas embarcados existentes. Por terem uma forte interação com o ambiente, esse tipo de sistema possui sensores e atuadores, além de componentes de *hardware* que são comuns a computadores(como memória, CPU, entrada/saída, entre outros). Sistemas embarcados utilizam sensores para receber estímulos do ambiente no qual estão inseridos, e atuadores para atuar no ambiente, quase sempre como resposta a um estímulo recebido. Entre o processo de recepção de estímulo e atuação no ambiente há um processamento. Uma vez que a computação é feita em um mundo digital e o sistema deve atuar no mundo real(i.e. analógico) é necessária uma conversão de analógico para digital, e digital para analógico.

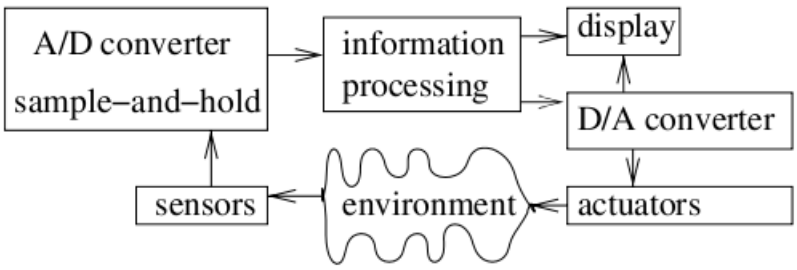

Figura 1: Visão geral da organização de um sistema embarcado

Um exemplo clássico de um sistema embarcado que segue esse *loop* é o controle de fluxo

de fluidos em canos industriais, muito comum na fabricação de equipamentos. Nesse caso, há um sensor no cano conectado ao sistema embarcado que consegue capturar informações sobre o fluxo dos fluidos nos canos. A esse sistema embarcado tambem é conectado uma válvula (atuador do sistema) que permite aumentar ou diminuir o fluxo dos fluidos.

É importante notar que nem todos os sistemas embarcados seguem essa organização. Aplicações multimídia são exemplos de sistema tipicamente embarcados que não têm necessidade de monitorar e atuar no ambiente que estão inseridos.

Muitos sistemas embarcados demandam a utilização de um sistema operacional. Os principais motivos que levam a isso são:

- Necessidade de suporte a escalonamento inteligente, e troca de tarefas em aplicações complexas.
- Necessidade de entrada/saída, comnuicação e sensoriamento
- Restrições (controle sobre consumo de energia, *threading*, etc)

2.2 SISTEMAS DE TEMPO REAL

Um sistema de tempo real é caracterizado por ter comportamento temporal como uma de suas principais restrições. Ele pode ser visto como um conjunto de tarefas ou processos associados a um sistema operacional de tempo real ou a um executivo cíclico(o código das tarefas é executado sequencialmente num ciclo infinito). A corretude de tais tarefas depende não só do resultado lógico dos processamentos, mas também do momento em que os resultados são produzidos.

Uma característica presente nos sistemas de tempo real é a definição de *deadlines* para suas tarefas. Essas *deadlines* servem para simbolizar o tempo limite que a tarefa tem para terminar a sua execução.

Nem todos os sistemas embarcados são de tempo real, e nem todos os sistemas de tempo real são embarcados (conforme ilustra a Figura 2). Porém, uma vez que quase todos os sistemas embarcados precisam interagir com o mundo real, tempo real passa a ser um item presente na maior parte desses sistemas, desde os pequenos sistemas para uso pessoal como mp3 players, aparelhos de DVD ou máquianas fotográficas digitais, até sistemas embarcados mais críticos, como os de aviões, carros ou de automação industrial.

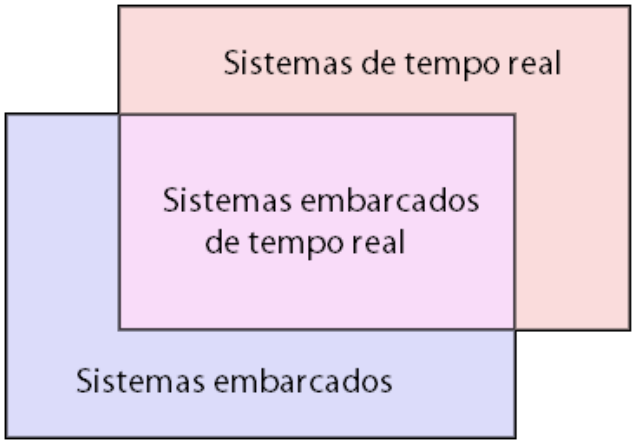

Figura 2: Conjunto dos sistemas embarcados e de tempo real

2.2.1 TAREFAS DE TEMPO REAL

Devido à diversa gama de sistemas de tempo real existentes, é possível observar que determinados sistemas possuem necessidades de tempo real diferentes, ou mesmo que determinadas tarefas de cada sistema apresentam restrições temporais mais rígidas que outras. Surge daí a necessidade de se modelar as tarefas de cada sistema de acordo com a sua necessidade de tempo real.

Na literatura de tempo real, fala-se principalmente em três tipos de tarefas: *hard real-time*, *soft real-time* e *best-effort*. Cada qual têm suas características e restrições explicadas abaixo:

- **Tarefas *hard real-time*** devem alcançar seus *deadlines* com um grau de flexibilidade muito próximo a zero (Li; Yao, 2003). A perda de uma *deadline* desse tipo de tarefa além de significar uma grave falha do sistema, significa um erro que na prática pode resultar em altos prejuízos, ou até mesmo a perda de vida humanas. Exemplos desse tipo de tarefa podem ser encontrados nos sistemas que controlam os freios *ABS* de carros.

- **Tarefas *soft real-time*** são tarefas que também possuem restrições temporais, mas há um certo grau de flexibilidade em relação a perda de seus *deadlines*(Li; Yao, 2003). A perda de um *deadline* desse tipo de tarefa não resulta em uma falha grave do sistema, mas sim em uma queda na qualidade de serviço. Um exemplo comum disso pode ser a codificação de um frame num aparelho DVD; caso haja a perda de um frame, o resultado pode nem ser percebido pelo usuário.

- **Tarefas *best-effort*** não têm restrições temporais. O nome delas é auto-explicativo, tentarão obter o **melhor esforço** em sua execução, porém, não devem ficar famintas de re-

cursos. Em muitos sistemas, tarefas best effort são utilizadas como tarefas secundárias. Atualização de dados em um monitor de LCD pode ser um exemplo de tarefa best effort, uma demora a mais para se realizar não é algo crítico para o funcionamento do sistema, desde que em algum momento a atualização seja feita.

2.2.2 ESCALONAMENTO DE TEMPO REAL

Sistemas baseados em tarefas geralmente armazenam suas tarefas em uma lista ordenada, que representa a ordem na qual as tarefas deverão executar. O escalonador é a parte do sistema responsável por ordenar essa lista, e o faz, utilizando algum critério. É um dos mecanismos mais importantes e críticos de um sistema de tempo real, pois o cumprimento dos *deadlines* (e o momento do cumprimento das tarefas) irá depender diretamente das decisões tomadas por ele, assim como a QoS do sistema como um todo.

Alguns exemplos de algoritmos clássicos de escalonamento:

- ***Earliest Deadline First*** (*deadline* mais próximo primeiro) - Nesse algoritmo, quanto mais eminente for o *deadline* da tarefa, maior será sua prioridade de execução (ou mais perto do inicio da fila de escalonamento se encontrará a tarefa). É amplamente utilizado em sistemas de tempo real.

- ***Rate-monotonic*** - Também é um algoritmo de escalonamento de tempo real. Ele utiliza prioridade estática, definida com base no período de cada tarefa. Quanto menor for o período da tarefa, mais alta será a sua prioridade. Geralmente é implementado com preemptividade (i.e. o escalonador pode interromper a tarefa que está executando, não precisando esperar seu fim para escalonar a próxima) e tem garantias determinísticas em relação aos tempos de resposta.

Um sistema de tempo real **escalonável**, deve satisfazer o seguinte critério (Tanenbaum, 2007):

$$\sum_{i=1}^{m} \frac{Ci}{Pi} \leq 1 \tag{2.1}$$

Onde há m eventos periódicos, cada qual com seu período Pi, e tempo de processamento Ci.

Nesse contexto, Yuan (Yuan et al., 2001) propôs um algoritmo chamado R-EDF (*Reserved EDF*), para o escalonamento de tarefas de tempo real e best-effort. O algoritmo reserva tempo

de processamento para as tarefas de tempo real e utiliza o conceito de *"time-sharing capacity"* que é a capacidade não reservada do tempo de processamento, a qual é compartilhada por todas as tarefas best-effort. Essa capacidade tem um limite inferior, que serve para impedir que as tarefas best-effort fiquem famintas de recursos.

2.3 GERENCIAMENTO DE ENERGIA

Computadores, aparelhos eletrônicos e sistemas embarcados podem ser encarados como um conjunto de componentes de *hardware*. A característica que esses aparelhos possuem de desligar a energia ou trocar o modo de consumo de seus componentes é denominada "Gerenciamento de energia".

A diminuição do consumo de energia, como citado anteriormente, é um item de extrema importância na industria de sistemas embarcados. Primeiramente, pois há um grande volume de calor produzido pelo *hardware* do sistema em funcionamento. Isso implica na necessidade de requisitos de resfriamento, que por sua vez, significam integrar ao produto dispositivos caros ou que ocupem um espaço indesejado. Quanto menor for o consumo, mais flexíveis serão os requisitos de resfriamento associados ao sistema em questão, dando ao projeto possibilidades de dispositivos de resfriamentos mais baratos ou menores.

No caso de sistemas embarcados cujo a energia é fornecida por baterias, existe além da questão do resfriamento o fato de a energia de alimentação ser limitada, o que faz com que o tempo de vida do sistema seja também limitado. Dada a existência desse limite no tempo de vida, o prolongamento do tempo de vida do sistema é mais um item a se adicionar na responsabilidade do gerenciamento de energia de sistemas embarcados.

Além da diminuição dos requisitos de resfriamento e prolongamento da bateria, outras características que podem ser citadas de sistemas embarcados com um bom gerenciamento ou baixo conumo de energia é a redução dos custos das operações e maior confiabilidade.

Tendo esses fatores em vista, quase sempre o *hardware* de sistemas embarcados é projetado para baixo consumo de energia, e não raro possui modos de operações diferenciados; cada modo possui um consumo de energia diferente do outro, e isso se reflete em sua performance. DVS (*Dynamic voltage scayling*) e DPM (*dynamic power management*) são duas técnicas amplamente utilizadas na indústria que exploram essas características do *hardware* e possiblitam mudanças no modo de operação de alguns componentes, para adicionar performance ou economizar energia. Essas tecnicas serão expostas a seguir.

2.3.1 *DYNAMIC VOLTAGE SCALING*

Através de modificações na tensão, essa técnica permite que o processador opere em frequências diferentes. A energia total **E** consumida pela computação em um processador é proporcional ao quadrado da tensão:

$$E \propto V^2 \tag{2.2}$$

A tabela abaixo (Ishihara; Yasuura, 1998) ilustra o comportamento da técnica. Nela há a energia consumida por ciclo, a frequência máxima e a duração do ciclo de um processador com DVS em três diferentes níveis de tensão.

Vdd (V)	5.0	4.0	2.5
Energy per cycle (nJ)	40	25	10
fmax (MHz)	50	40	25
cycle time (ns)	20	25	40

Através da observação da tabela é possível inferir que a tensão mais baixa possui a melhor eficiência energética - cerca de quatro vezes mais do que a primeira. No entanto, a tensão mais baixa possui o tempo de ciclo duas vezes maior do que a maior tensão, logo ela não é desejavel quando houver necessidade de performance ou eminência de algum *deadline* (no caso de um sistema de tempo real).

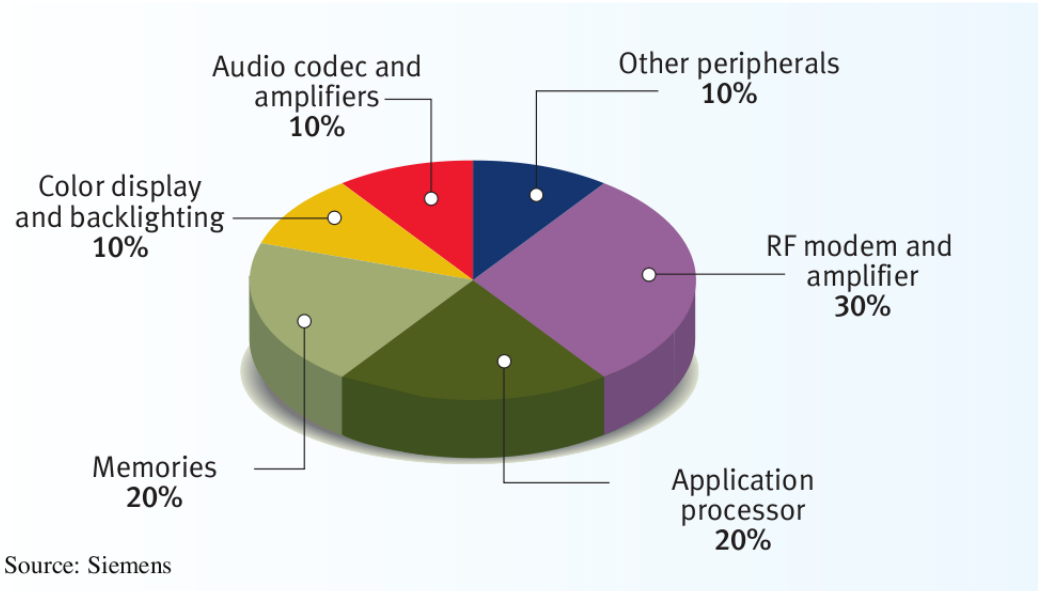

Figura 3: Consumo de energia em dispositivos móveis

A Figura 3 (VARGAS, 2005) mostra o consumo de energia dos componentes de *hardware* de dispositivos móveis. Nela é possível ter uma idéia que processadores consomem uma grande quantidade de energia se comparados aos resto do sistema. Por esse fato, e pelo impacto da diminuição da tensão na energia consumida pelo processador, a técnica DVS ajuda a alcançar grandes economias de energia, principalmente em momentos em que não é exigido grande performance da CPU.

2.3.2 *DYNAMIC POWER MANAGEMENT*

Essa técnica visa o desligamento ou hibernação de dispositivos *off-chip* com o objetivo de evitar o desperdício de energia quando os mesmos não são usados. Tipicamente os dispositivos *off-chip* tem modo de operação ativo e ao menos um *sleep*, o que faz com que essa técnica possa ser aplicada quase sempre a aparelhos eletrônicos.

A imagem abaixo mostra diferentes modos de operação do processador StrongArm SA 1100. Cada estado possui um consumo de potência bastante diferente, e também um tempo associado a cada transição de modo de consumo possível.

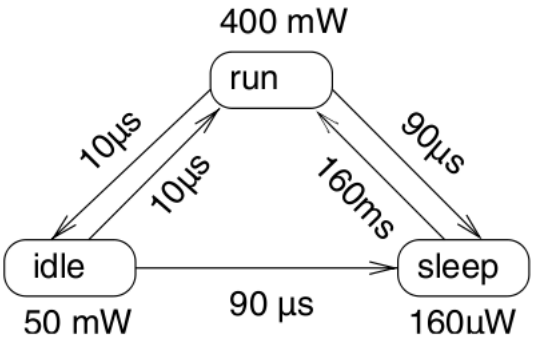

Figura 4: Estados DPM do processador *StrongArm SA 1100*

2.3.3 REQUISITOS DO GERENCIAMENTO DE ENERGIA

No que tange o gerenciamento de energia nos sistemas de tempo real, há três itens muito importantes que devem ser levados em consideração e analisados delicadamente no desenvolvimento dessa prática.

- O primeiro cabe tanto à técnica DVS quanto DPM e é relacionado ao desempenho dos modos de operação escalonados (ou da tensão, no caso da técnica DVS) - ao se diminuir a energia consumida, também diminui-se a performance e em sistemas de tempo real isso

pode significar perdas de deadlines ja que será necessário um tempo maior para se realizar as computações.

- O segundo é em relação principalmente à técnica DPM, e diz respeito ao sobrecusto em tempo associado à troca no modo de consumo. No caso da Figura 4, é possível observar que há uma quantidade de tempo muito grande associada a troca do estado ***sleep*** para o estado ***run***. Se esse sobrecusto não for prevista num projeto que faça grande uso da tecnica, há grandes possibilidades de em algum momento algum *deadline* ser perdido.

- Uma outra questão delicada relacionada ao gerenciamento de energia é o sobrecusto relacionado à troca de modo de consumo de cada componente. Se um componente do sistema é colocado em modo ocioso e em um curto período de tempo ele é transicionado novamente para modo ativo, o sobrecusto associado a essas transições causará maior consumo de energia do que se tivesse ficado apenas no modo ativo. Nesse contexto, alguns autores (Devadas; Aydin, 2008) utilizam o conceito de *break-even time*, que refere-se ao tempo no qual o dispositivo deve permanecer em um modo de operação para não causar sobrecusto em termos de energia.

As técnicas de gerenciamento de energia DVS e DPM exercem influências uma sobre a outra (Devadas; Aydin, 2008). No caso do DVS, um processador que opera com frequência baixa tem um consumo baixo de energia, mas isso resulta em um tempo alongado de execução das tarefas. Esse fato faz com que além dos dispositivos ficarem no estado ativo por mais tempo, haja menor possibilidade de transição para um estado *sleep*. Por outro lado, se o processador operar com uma frequência mais alta, haverá mais possibilidades para os dispositivos realizarem alguma transição para um estado *sleep*.

2.3.4 ESCALONAMENTO COM BASE NOS NÍVEIS ENERGÉTICOS DAS BATERIAS DO SISTEMA

Uma abordagem interessante para o gerenciamento de energia que foi e está sendo muito explorada(Wiedenhoft, 2008; Liu, 2001; Wang, 2003) é o desenvolvimento de escalonadores de tarefas que possuem noções da energia presente no sistema(também conhecido como escalonadores *power aware*). Ações comuns em um escalonador desses variam desde a escolha de tarefas mais viáveis energeticamente(dependendo da carga do sistema) até o descarte efetivo de tarefas que possam comprometer de alguma forma a energia do sistema.

Em (Wang, 2003), um algoritmo de escalonamento chamado PA-BTA (*Power-Aware Best-Effort Real-Time Algorithm*) foi proposto. Esse algoritmo como objetivo otimizar a performance de tempo real e o consumo de energia do sistema em que estiver inserido. O algoritmo

realiza euristicamente(porém de modo eficiente) o escalonamento de tarefas para maximizar o ERG(*Energy and Real-Time Performance Grade*), uma métrica proposta para medir a performance tanto de tempo real como de consumo de energia.

Em (Wiedenhoft; Fröhlich, 2008), houve a proposta de um escalonador de tarefas que tem noção dos níveis de energia das baterias do sistema. Nesse trabalho, as tarefas foram divididas em obrigatórias e opcionais. As tarefas obrigatórias tem que executar sempre, e não podem sofrer descartes em hipótese nenhuma. Já as tarefas opcionais só poderão ser escalonadas, se dois critérios forem atendidos: tempo e energia, ou seja só serão escalonadas se não houver interferência nos *deadlines* das tarefas obrigatórias, e se houver energia o suficiente para completar o escalonamento de todas as tarefas obrigatórias.

Para satisfazer esses critérios, os autores implementaram um escalonador que realiza descartes das tarefas opcionais caso seja verificado (atravéz de medições nas baterias do sistema) que o tempo definido ao sistem não será alcançado, em outras palavras, o escalonamento das tarefas obrigatórias será interrompido antes do tempo certo. Esse trabalho foi feito no EPOS, o mesmo sistema operacional em que a abordagem proposta nesse trabalho será implementada.

3 IMPLEMENTAÇÃO DO MECANISMO DE GERENCIAMENTO DE ENERGIA

Nesse capítulo apresentamos a abordagem proposta em detalhes, explicando como é o gerenciamento de energia do EPOS, o que foi modificado e as estruturas mantidas. Conforme levantado na introdução, as maiores motivações por traz desse trabalho são fazer com que o sistema alcance o tempo de vida definido a ele e fornecer a maior QoS possível às suas tarefas *best-effort*.

O termo QoS (acrônimo de *"Quality of Service"*, em português "Qualidade de Serviço") designa a capacidade de fornecer um serviço, o qual pode ter um grau de satisfação para seu usuário bastante variável, dependendo da configuração de uma série de características qualitativas e quantitativas. No contexto desse trabalho, a QoS atribuída às tarefas *best-effort* pode ser encarada como o seu percentual de descarte (um percentual alto significa uma baixa QoS, e vice versa).

3.1 EPOS

O EPOS (*Embedded Paralell Operating System*) (Fröhlich; Schroder-Preikschat, 1999) é um sistema operacional orientado a aplicação que foi concebido segundo a ADESD (*Application-Driven Embedded System Design*) (Fröhlich, 2001), e permite ao programador da aplicação gerar sistemas específicos, agregando apenas os componentes necessários à aplicação definida, que são selecionados e configurados automaticamente por uma série de ferramentas.

Esse foi o sistema escolhido para implementação do modelo proposto pois ele possui excelentes características de escalonamento de tarefas e gerenciamento de energia. Além disso, agregar e desenvolver componentes de software para ele é simples e prático, devido à sua arquitetura.

3.2 MODELAGEM DAS TAREFAS

Para esse trabalho, as tarefas serão abstraídas como *threads*, e precisarão ser *hard real-time* ou *best-effort*. Como todas as suas funcionalidades serão as mesmas, não houve a necessidade de implementar um novo objeto; portanto ambas serão o próprio objeto ***thread*** do EPOS.

A prioridade das *threads* será o que irá as diferenciar. Como as *threads hard real-time* devem sempre ter prioridade de execução superior às *best-effort*, nesse trabalho iremos modelar uma ”fronteira de prioridade”da seguinte maneira:

- ***threads hard real-time*** terão sua prioridade entre 0 e (**max int**)/2
- ***threads best-effort*** terão sua prioridade entre (**max int**)/2 e **max int**

Onde **max int** é o maior inteiro possível na plataforma alvo. O valor das prioridades seguirá o atual modelo do EPOS, onde quanto menor for o valor, maior será sua posição na fila de escalonamento (i.e. maior a sua prioridade).

As tarefas fundamentais do sistema (ou que possuem requisitos de tempo real) deverão ser expressadas pelo programador da aplicação como *threads hard real-time*. Já as tarefas que possuem um caráter opcional poderão ser expressadas como *threads best-effort*. O código abaixo ilustra a instância de duas *threads* uma *best-effort* com a função *optional-function* e uma *hard real-time* com a função *mandatory-function*. No caso desse exemplo, o maior inteiro da arquitetura é 65535, logo as *threads* que possuírem prioridade maior que 32767 serão *best-effort*, e as que possuírem prioridade menor, *hard real-time*.

```
1        typedef Traits<Thread>::Criterion Criterion;
2
3        Criterion hard_realtime_criterion(150e3,170e3);
4        Criterion besteffort_criterion(60000);
5
6        Thread hard_realtime_task(&mandatory_function, System::Thread::
             READY, hard_realtime_criterion);
7        Thread besteffort_task(&optional_function, System::Thread::
             READY, besteffort_criterion);
```

Algoritmo 1 : Instanciação de uma tarefa *hard real-time* e uma *best-effort*

Em sistemas operacionais, quando há diferentes categorias de tarefas é comum ser utilizada diferentes filas (uma para cada categoria). Porém, como o trabalho se encontar no contexto de sistemas embarcados, sistemas os quais o número de tarefas é bastante reduzido, foi decidido utilizar apenas uma fila, afim de se evitar sobrecustos.

3.3 MEDIÇÕES NA CARGA DA BATERIA

O EPOS conta com um monitor, que verifica a tensão existente nas baterias, e pode determinar a quantidade aproximada de energia restante nelas (Gonçalves, 2007). Isso é um pilar muito importante para o seu gerenciamento de energia, uma vez que é esse mecanismo que possibilita os cálculos para verificar se o tempo atribuído ao sistema será alcançado.

As medições nas baterias do sistema costumam causar um sobrecusto relativamente alto em termos de energia(maiores detalhes desse sobrecusto serão dados nos estudos de casos). Por isso, as medições deverão ocorrer com a menor frequência possível.

Nesse trabalho iremos fazer com que ao se verificar que o tempo estipulado para o sistema não será alcançado, as medições não fiquem mais frequentes, ao contrário do atual modelo implementado no EPOS (Wiedenhoft; Fröhlich, 2008). Uma desvantagem para realização de medições com frequências menores é que o sistema demorará mais tempo para se dar conta de que as tarefas de carater opcional podem voltar a serem escalonadas normalmente.

Por outro lado, o sistema terá um **menor consumo de energia acumulado** (uma vez que o sobrecusto associado as medições será menor). Como nessa abordagem a QoS será diretamente proporcional aos níveis de energia existentes na bateria do sistema, a desvantagem citada não será tão impactante.

3.4 CÁLCULO DO TEMPO E ENERGIA

O cálculo do tempo e energia é possível no EPOS, pois além do monitor de carga de bateria, o sistema armazena a quantidade de energia da ultima medição, o tempo decorrido desde o início da aplicação, e o tempo que a aplicação deve alcançar.

Em (Wiedenhoft, 2008), a seguinte equação foi proposta, para verificar se o tempo definido ao sistema será alcançado:

$$\frac{E_{tk}}{E_{tk-1} - E_{tk}} \times (T_{tk-1} - T_{tk}) \leq T_{tk} \tag{3.1}$$

Onde Ttk é o tempo que o sistema ainda deve durar, Etk a carga da bateria no instante, Ttk-1 e Etk-1 são o tempo que o sistema deveria durar e a carga da bateria no instante *k-1* (i.e. na medição anterior). A equação verifica se a perda de carga foi proporcionalmente maior do que o tempo decorrido.

Nesse trabalho, utilizaremos o termo **deficit de energia** para denotar que o sistema não

alcançará o tempo estipulado a ele. Como nesse trabalho há a necessidade de uma informação mais precisa em relação ao quão crítica está a carga de bateria em relação a esse tempo (ou quão crítico está o *deficit* de energia), ao invés de um valor booleano, utilizaremos um valor do tipo inteiro como resultado da equação que realiza a verificação.

A equação 4.2 é usada para se calcular o tempo estimado para o sistema:

$$TE = \frac{E_{tk}}{E_{tk-1} - E_{tk}} \times (T_{tk-1} - T_{tk}) \tag{3.2}$$

Usamos a saída da equação a cima (TE - *Tempo estimado*) para calcular a *taxa de deficit de energia*, por sua vez será utilizada para o cálculo do deficit de energia. As equações para o cálculo da *taxa de deficit de energia* e do *deficit de energia* são as seguintes:

$$TDE = \frac{T_{tk}}{TE} \tag{3.3}$$

$$DE = \frac{maxint}{2} \times \frac{TDE - 1}{CDE}; \tag{3.4}$$

A **taxa de deficit de energia** (Equação 4.3) é calculada pela divisão do tempo que o sistema deve durar pelo tempo estimado.

O *deficit de energia* terá um valor entre zero e **maxint**/2. Isso acontece porque ele deverá significar um bloqueio de prioridade (isso será explicado em detalhes na próxima seção). Como apenas as tarefas *best-effort* deverão ser descartadas em função desse deficit, o valor não deve exceder esse limite.

A equação 4.4 que calcula o *deficit de energia* fará uso da *taxa do deficit de energia* e do *CDE* (Coeficiente do deficit de energia), uma variável que pode ter valor configurado pelo programador da aplicação e será usada para indicar o quão radical será o sistema ao entrar em deficit de energia. A Figura 5 mostra o deficit de energia calculado com diferentes coeficientes de deficit de energia, para uma arquitetura onde o maior inteiro é 65535 (logo o maior deficit de energia é 32767):

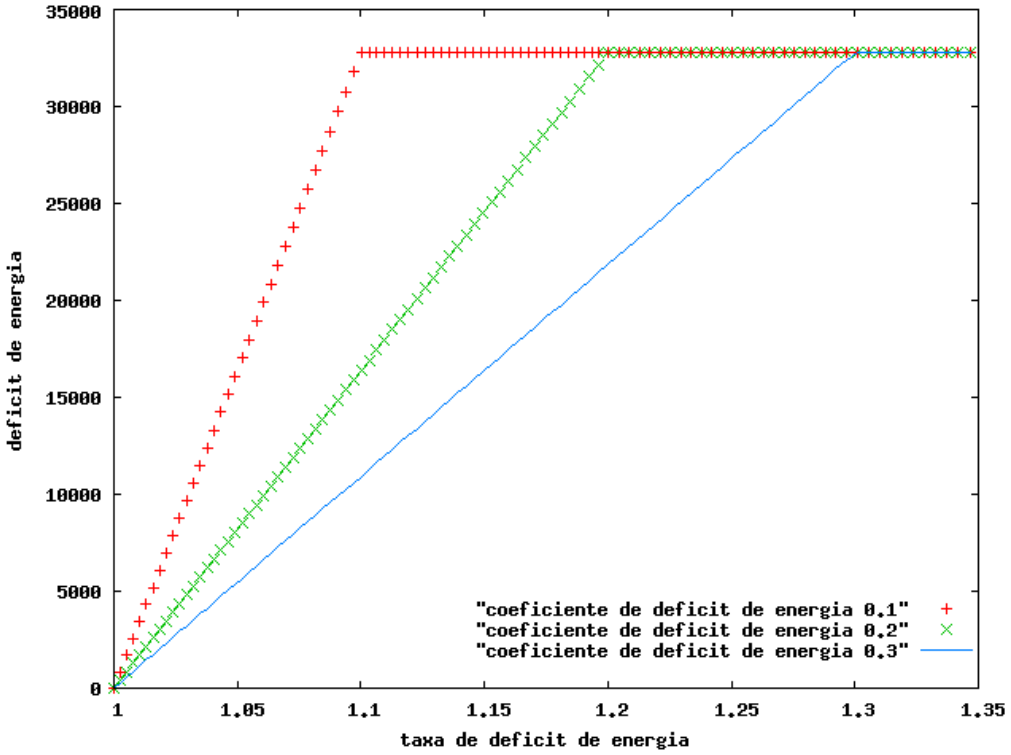

Figura 5: *Deficit de energia* calculado em função da *taxa do deficit de energia* para diferentes *coeficientes do deficit de energia*

3.5 CRITÉRIO PARA DESCARTE DE TAREFAS

Caso os níveis de energia nas baterias do sistema estejam críticos(i.e. o tempo definido para o sistema não será alcançado), o sistema deverá ter um comportamento diferente, economizando energia até ser constado que há energia o suficiente nas baterias para alcançar o tempo de duração definido ao sistema.

Para fazer isso, a atual implementação do gerenciamento de energia do EPOS (Wiedenhoft; Fröhlich, 2008) propôs um modelo no qual o sistema realiza descartes de tarefas opcionais. De modo semelhante, nessa implementação, o sistema passará a ter dois comportamentos distintos em função da energia de suas baterias em relação ao tempo que deverá ser alcançado:

- **ausência de *deficit* de energia no sistema**: as tarefas do sistema serão escalonadas normalmente.
- **presença de *deficit* de energia no sistema**: as tarefas de tempo real continuarão a ser escalonadas normalmente, porém as *best-effort* poderão ser descartadas, e no lugar delas será escalonada uma tarefa ociosa, a qual possui um comportamento econômico no ponto de vista energético.

A Figura 6 ilustra a faixa de prioridade do sistema, que varia de 0 (prioridade mais alta) à **maxint**(prioridade mais baixa). Nela há também um exemplo da faixa de prioridade das tarefas que são afetadas por um deficit de energia. Essa fronteira de prioridade que chamaremos de **limite do deficit de energia** serve para separar as tarefas *best-effort* que sofrerão algum descarte das que não sofrerão nenhum.

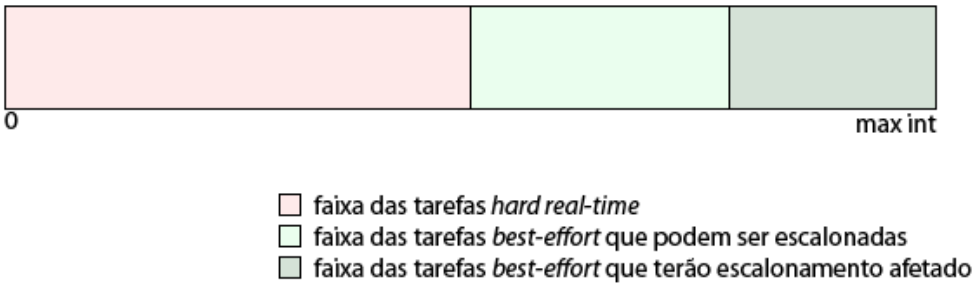

Figura 6: Faixa de prioridade do sistema em deficit de energia: tarefas *hard real-time*, *best-effort* que não sofreram descarte e *best-effort* que sofrerão

Apesar das tarefas *best-effort* estarem sujeitas a descartes, elas não devem ficar famintas de recursos. Desse modo, foi criado um atributo **fome**, que faz com que as tarefas *best-effort* possam, independente dos níveis de energia no qual o sistema se encontre e da prioridade da tarefa em questão, ser escalonadas em algum momento. O escalonador do sistema, conforme será introduzido na próxima seção, irá então, realizar o descarte das tarefas baseado no deficit de energia, na prioridade da tarefa e na sua fome.

O Algoritmo 2 apresenta um pseudo-código referente à atualização da fome das threads. Além de atualizar a fome, esse mecanismo serve para hibernar e deshibernar *threads* que estão sujeitas à descarte.

```
1
2  update_hunger(new_hunger, energectic_deficit_bound){
3
4          //verifica   o se a thread est  entre as afetadas pelo deficit
5        if( priority() > energectic_deficit_bound){
6
7              _hunger += new_hunger;
8
9                  //tarefa vai desihibernar
10             if( priority() <= energectic_deficit_bound + _hunger and
                   new_hunger>0){
```

```
11
12                      if( energectic_deficit_bound <= ( max_int * 0.75 ) ){
13                              _hunger += 10;
14                      }else{
15                                      _hunger += max_int -
                                            energectic_deficit_bound;
16                      }
17              }
18
19                  //tarefa vai hibernar
20              if( priority() >= energectic_deficit_bound + _hunger and
                    new_hunger<0){
21                      _hunger = 0;
22              }
23  }
```

Algoritmo 2 : Atualização da fome da *thread*

O método recebe dois inteiros: **new hunger** (nova fome) que servirá para incrementar a fome da *thread*, e **energectic deficit bound** (limite do deficit de energia) que conforme dito anteriormente, indica o deficit de energia do sistema em relação à prioridade das tarefas. O inteiro *new hunger* terá um valor negativo caso a *thread* que esteja tendo a sua fome atualizada esteja sendo escalonada, ou com um valor positivo caso ela não esteja.

Uma *thread* que está sujeita a descarte tem o valor atribuído à sua prioridade maior que o limite do deficit de energia, logo se a sua fome for zero, ela não terá possibilidade de ser escalonada e estará **hibernada**. Cada vez que ela deixar de ser escalonada, a sua fome vai ser incrementada até que a fome, somada com o *limite do deficit de energia* tenha um valor maior que a sua prioridade. Aí então essa *thread* será **deshibernada**, passará a ser escalonada, e o mecanismo usado para isso é o reaproveitamento do seu próprio atributo fome que passará a sofrer um grande incremento em seu valor. A razão para isso é que o escalonador faz a verificação do descarte ou não da tarefa usando esse atributo, e desse modo foi possível obter uma maior eficiência de código.

Caso o sistema esteja com um nível grave de deficit de energia (entre o pior possível e o nível médio), a fome da tarefa que estiver deshibernando será incrementado em apenas 10 (o suficiente para ela ser escalonada dez vezes).

A Figura 7 e a Figura 8, ilustram um exemplo onde há o número de descartes de tarefas *best-effort*, medições nas baterias e o *deficit* de energia no sistema em função do tempo. No caso, em ambos os diagramas, fizemos suposições de dois *deficits* de energia, um pequeno e um

grande, e ilustramos o comportamento do sistema em termos de descarte de tarefas e frequências de medições. O primeiro diagrama é referente ao modelo atual do sistema EPOS (Wiedenhoft; Fröhlich, 2008), e o segundo o modelo que pretendemos implementar nesse trabalho.

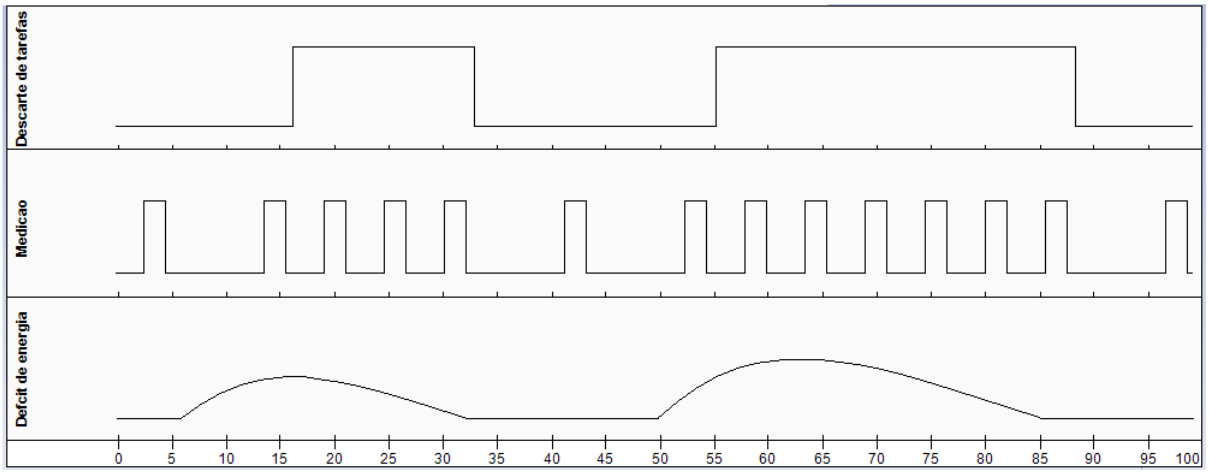

Figura 7: Modelo atual do EPOS referente ao descarte de tarefas, e frequência de medições em função do *deficit* de eneriga

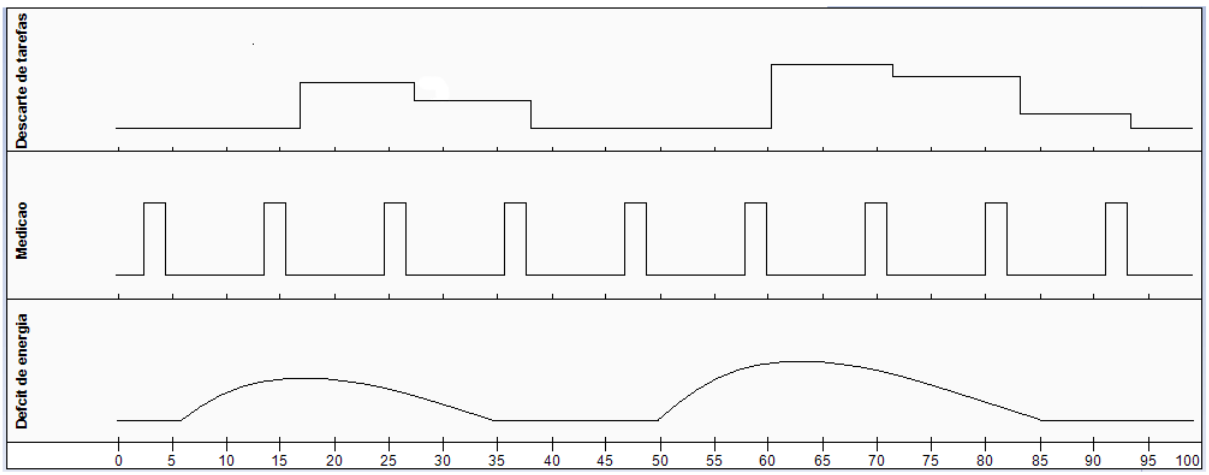

Figura 8: Modelo proposto para o sistema EPOS referente ao descarte de tarefas, e frequência de medições em função do *deficit* de energia

3.6 ESCALONADOR

Para se realizar as operações previstas houve a necessidade de implementação de um novo escalonador e ele foi chamado de *Scheduler RBPA* (Real-time Best-effort Power Aware). O

EPOS já possui dois escalonadores conscientes de energia, o *power aware scheduler*(Wiedenhoft; Fröhlich, 2008) e o *imprecise scheduler*(Wiedenhoft, 2008). A figura 9 mostra as hierarquias de todos os escalonadores *power aware* do EPOS. O critério de escalonamento utilizado para o escalonamento das tarefas de tempo real é o *scheduling criteria* EDF (*Earliest Deadline First*) do EPOS.

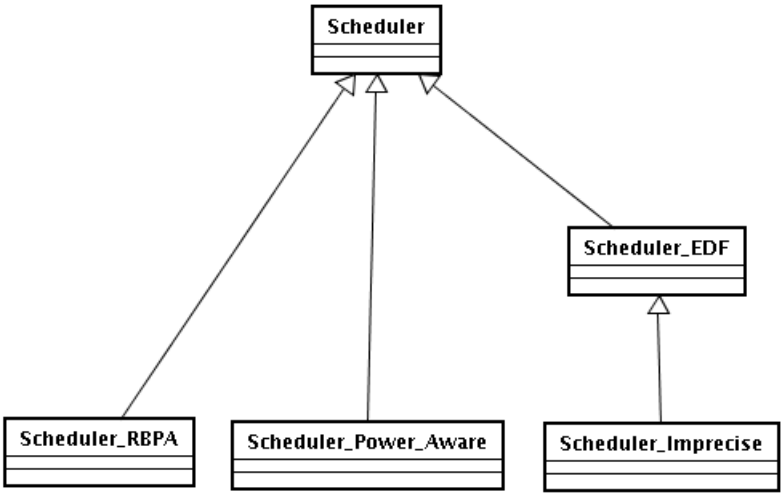

Figura 9: Escalonador proposto na hierarquia de escalonadores *power aware* do EPOS

Os dois escalonadores existentes utilizam energia como base para QoS, e separam as tarefas em mandatórias e opcionais. As mandatórias tem a garantia que sempre serão executadas, independente dos níveis de energia nas baterias do sistema. Já as tarefas opcionais, serão todas descartadas caso seja verificado que o tempo estipulado ao sistema não será alcançado.

O escalonador proposto segue uma linha de comportamento parecida, porém ele realizará o descarte das tarefas baseado na prioridade, no deficit de energia no qual se encontra o sistema e no atributo fome de cada tarefa, o que melhora a qualidade de serviço atribuída as tarefas opcionais, ou no caso desse trabalho *best-efforts*.

A Figura 10 mostra o diagrama de sequência do escalonador proposto no contexto do mecanismo de escalonamento do EPOS. O método *choose* do escalonador é responsável por escolher a próxima *thread* da fila de *threads* a ser executada. Caso a tarefa escolhida seja uma *best-effort*, e a sua prioridade seja maior do que o *limite do deficit de energia* mais a sua fome, ela não será escalonada e terá a sua fome incrementada. Caso sua prioridade seja menor, ela poderá ser escalonada, e sua fome será decrementada.

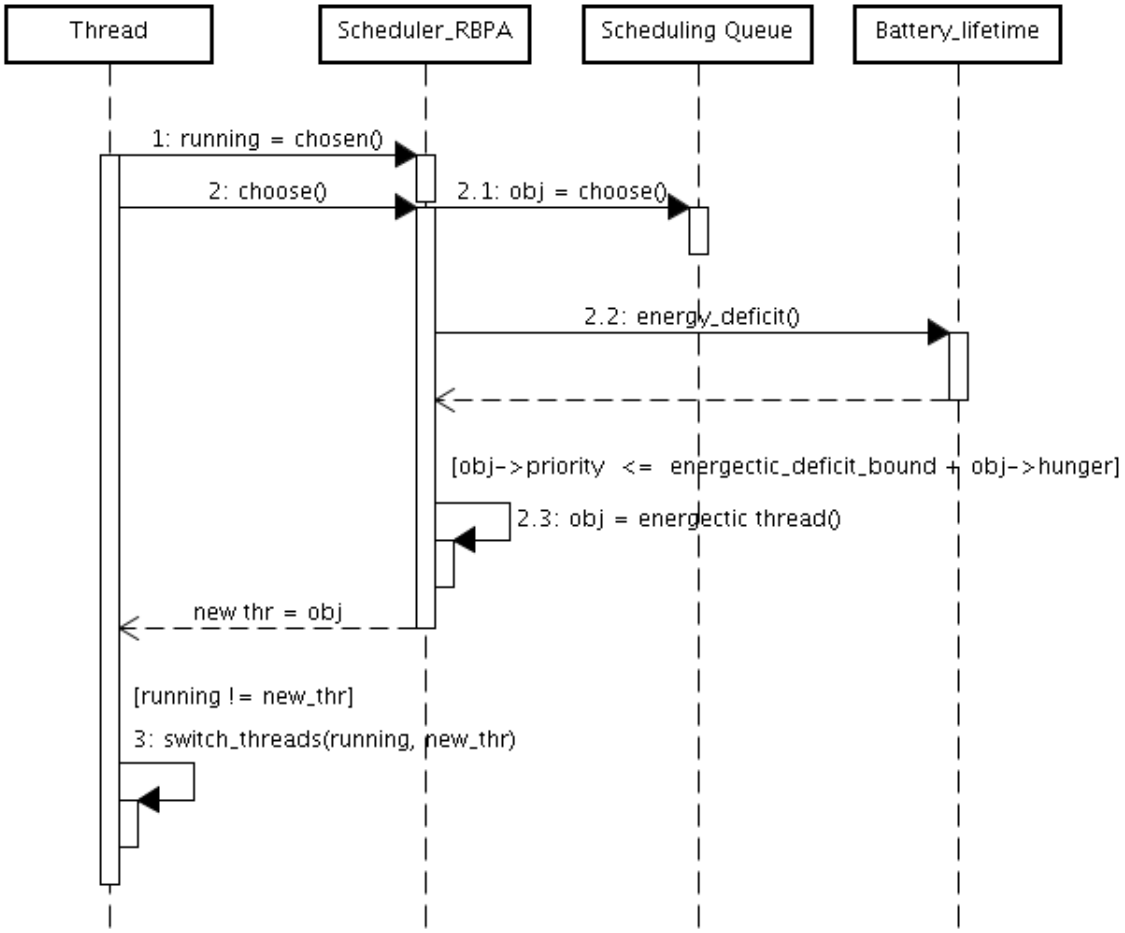

Figura 10: Diagrama de sequência do escalonador proposto

O Algoritmo 3, mostra um trecho do método choose do escalonador proposto. Nele é possível observar como a função ***update hunger*** é chamada. No caso, se a prioridade da thread for menor do que o *energectic deficit bound* mais a sua fome, ela poderá ser escalonada mas como pode ser visto no algoritmo 2, se a thread estiver sujeita a descarte, em algum momento acontecerá de ela hibernar e sofrer ao menos alguns descartes.

```
1
2  T * choose(){
3
4          T * obj = Base::choose_another();
5
6          (...)
7
8          if(priority() <= energectic_deficit_bound + obj->_hunger){
9                  obj->update_hunger(-1, _lifetime.
                       get_energectic_deficit_bound());
10         }
11         else{
12                 obj->update_hunger(1, _lifetime.
                       get_energectic_deficit_bound());
13                 obj = Base::choose(_energy_t);
14         }
15
16         return obj;
17 }
```

Algoritmo 3 : Método *choose* do escalonador implementado

4 ESTUDO DE CASOS

Para se testar a abordagem proposta nesse trabalho, foi desenvolvida uma aplicação na área de redes de sensores, semelhante a (Wiedenhoft, 2008). Essa aplicação utiliza a plataforma de sensoriamento *Mica2 mote* (Crossbow Tech, 2005), na qual fará uso de dois dispositivos: rádio e sensor de temperatura.

A aplicação consistirá em três *threads*. Duas irão utilizar o sensor para se obter a temperatura do ambiente, das quais uma será *hard real-time* e a outra *best-effort*. A *thread hard real-time* que utiliza o sensor irá realizar uma medição e liberar a CPU. Já a *thread best-effort* de sensoriamento fará 10 medições e irá tirar a média delas, afim de se obter um resultado mais preciso, porém que causa um maior consumo de energia no sistema. Além disso haverá uma terceira *thread hard real-time* que será responsável por utilizar o rádio para enviar à rede de sensores as temperaturas captadas pelas outra *threads*. Os diagramas de sequência abaixo ilustram essas *threads*:

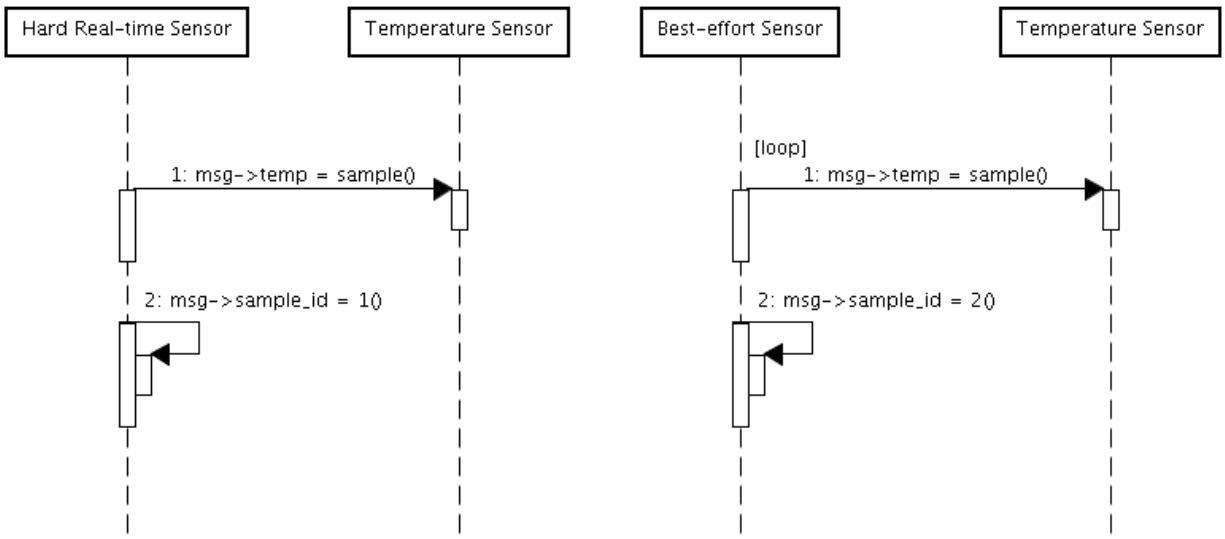

Figura 11: As *threads* da aplicação que utilizam o sensor

O maior inteiro da arquitetura é 65535, portanto as tarefas com prioridade superior a 32767

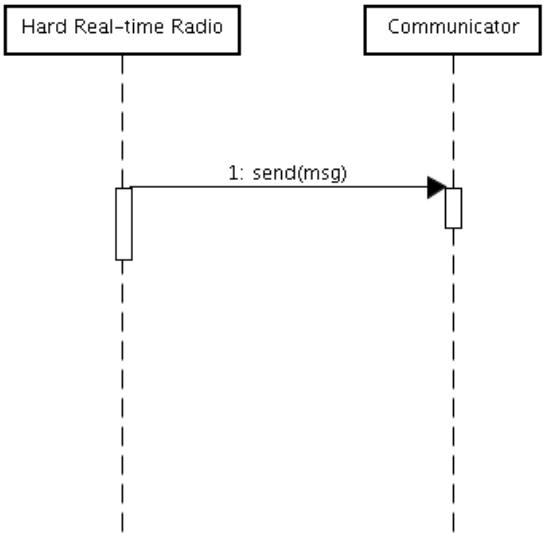

Figura 12: A *thread* da aplicação que utiliza o rádio

serão *best-efforts* e as com prioridade inferior serão *hard real-time*. No estudo de casos, a tarefa *best-effort sensor* terá prioridade 62000 (portanto ela só terá seu escalonamento afetado quando o deficit de energia for maior que 3535).

Usando um osciloscópio digital com dois canais foi possível obter a potência consumida pelos motes do estudo de casos. O primeiro canal foi utilizado lendo a tensão das baterias e o segundo lendo a corrente elétrica, conforme pode ser observado na Figura 13. Multiplicando-se a potência observada pelo tempo conseguimos estimar o consumo de energia.

Figura 13: Osciloscópio usado para se estimar o consumo de energia nos estudos de casos

Neste trabalho foram realizados dois estudos de casos, cada qual com um objetivo específico; ambos serão mostrados nas subseções seguintes.

4.1 IMPACTO DA ABORDAGEM PROPOSTA NO CONSUMO DE ENERGIA DO SISTEMA

No primeiro estudo de casos, realizamos medições em um sistema com a abordagem proposta com um deficit de energia baixo artificialmente inserido, um com a abordagem atual do sistema se descartando todas as tarefas opcionais, e um com a abordagem atual escalonando todas as tarefas, cada qual com uma bateria nova. Cada uma das execuções rodou por aproximadamente vinte horas.

Com os resultados obtidos, é possível ver que a energia a mais consumida com o escalonamento da tarefa *best-effort* com deficit de energia baixo, não implica em grandes sobrecustos para a aplicação. No caso da abordagem proposta foi observado um descarte de 56,4 porcento de seu escalonamento, sendo que a energia a mais consumida por esses escalonamentos ficou 30 porcento maior.

A tabela abaixo mostra o consumo de energia com cada uma das configurações desse estudo de casos:

Modelo	Potência consumida W)
Modelo atual se descartando todas as tarefas opcionais	0.04784
Modelo proposto com deficit de energia baixo (56,4 porcento de descartes)	0.04927
Modelo atual se escalonando todas as tarefas	0.05265

4.2 PERCENTUAL DE DESCARTE DE TAREFAS BEST-EFFORT EM FUNÇÃO DO DEFICIT DE ENERGIA

No segundo estudo de casos, foi deixado a aplicação rodando por três dias e foi armazenada as informações de todos os pacotes enviados pelo *mote* usado. Além das informações sobre a temperatura e o tipo da *thread* que realizou a medição, foi enviado informações sobre o valor do deficit de energia, e a o numero de medições ja feitas.

Com essas informações foi possível calcular o porcentual de tarefas *best-effort* descartadas para diferentes deficits de energia. A 14 mostra isso:

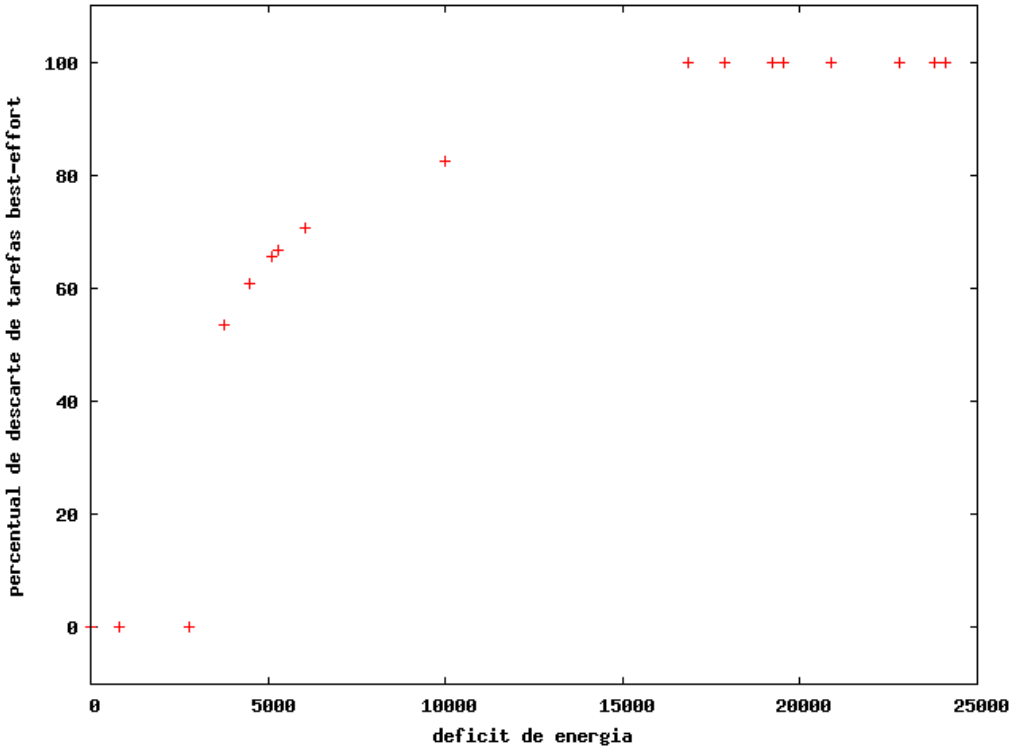

Figura 14: Percentual de descarte de tarefas em função do deficit de energia

O gráfico possui três regiões, ja previstas pela implementação. A primeira é quando o deficit de energia é inferior a 3535, e não é grande o suficiente para afetar o escalonamento da tarefa *best-effort*. A segunda região, é quando o deficit está entre 3535 e 16383 (não é um nível crítico ainda) e o descarte das tarefas é bastante variável em relação a cada valor desse intervalo; os descartes observados nesse intervalo variaram de 53,2 à 82,8 por cento. A terceira região, se dá quando a energia do sistema se encontra em um nível crítico, e a tarefa *best-effort* é escalonada apenas 10 vezes antes de seu atributo *fome* ser zerado; isso faz com que o percentual de descarte da tarefa *best-effort* quando o deficit de energia está acima de 16383 seja muito próximo de 100 por cento.

REFERÊNCIAS

CROSSBOW Technology. MPR/MIB Mote Hardware User's Manual. Crossbow Technology, Inc, San Jose, CA, September 2005

DEVADAS, Vinay; AYDIN, Hakan. On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications. In: *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*. New York, NY, USA: ACM, 2008. p. 99–108. ISBN 978-1-60558-468-3.

EGGERMONT, L.D.J. Embedded systems roadmap 2002. 2002.

FRÖHLICH, A.A.M. *Application Oriented Operating Systems*. [S.l.]: GMD-Forschungszentrum Informationstechnik, 2001.

FROHLICH, A.A.; SCHRODER-PREIKSCHAT, W. Epos: An object-oriented operating system. *Lecture Notes in Computer Science*, Springer-Verlag; 1999, p. 27–27, 1999.

GONÇALVES, R.T. Monitoramento da Capacidade de Baterias em Sistemas Embarcados.

JUNIOR, A. S. H. *Gerência do Consumo de Energia Dirigida pela Aplicação em Sistemas Embarcados*. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2007.

ISHIHARA, T; YASUURA, H. Voltage scheduling problem for dynamically variable voltage processors. In: *Proceedings of the 1998 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 1998. p. 197–202.

LI, Qing; YAO, Caroline. *Real-Time Concepts for Embedded Systems*. [S.l.]: CMP Books, 2003. ISBN 1578201241.

LIU, Jinfeng and CHOU, Pai H. and BAGHERZADEH, Nader and KURDAHI. Power-aware scheduling under timing constraints for mission-critical embedded systems. In: *DAC '01: Proceedings of the 38th annual Design Automation Conference* Las Vegas, Nevada, USA: ACM, 2001 p. 840–845 s

MARWEDEL, P. *Embedded system design*. [S.l.]: Springer, 2003.

TANENBAUM, Andrew S. *Modern Operating Systems*. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

VARGAS, O. Minimum power consumption in mobile-phone memory subsystems. 2005.

WANG, J. and RAVINDRAN, B. and MARTIN, T. A power aware best-effort real-time task scheduling algorithm. 2003.

WEISER, M.. Ubiquitous computing *IEEE Computer* 1993

WIEDENHOFT, G.R.; FROHLICH, A.A. Using Imprecise Computation Techniques for Power Management in Real-Time Embedded Systems. 2008.

WIEDENHOFT, G.R. et al. Power management in the EPOS system. ACM New York, NY, USA, 2008.

YUAN, Wanghong; NAHRSTEDT, Klara; KIM, Kihun. R-edf: A reservation-based edf scheduling algorithm for multiple multimedia task classes. In: *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*. Washington, DC, USA: IEEE Computer Society, 2001. p. 149.

```
1  // EPOS-- Scheduler Abstraction Declarations
2
3  #ifndef __scheduler_h
4  #define __scheduler_h
5
6  #include <utility/queue.h>
7  #include <rtc.h>
8  #include <tsc.h>
9  #include <battery_lifetime.h>
10
11 __BEGIN_SYS
12
13 // All scheduling criteria, or disciplins, must define operator int() with
14 // semantics of returning the desired order of a given object in the
15 // scheduling list
16 namespace Scheduling_Criteria
17 {
18     enum { INFINITE = -1 };
19
20     // Priority (static and dynamic)
21     class Priority
22     {
23     public:
24         enum {
25             MAIN   = 0,
26             HIGH   = 1,
27             NORMAL = (unsigned(1) << (sizeof(int) * 8 - 1)) -3,
28             LOW    = (unsigned(1) << (sizeof(int) * 8 - 1)) -2,
29             IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) -1
30         };
31
32     public:
33         Priority(int p = NORMAL): _priority(p) {}
34
35         operator const volatile int() const volatile { return _priority; }
36
37     protected:
38         volatile int _priority;
39     };
40
41     class PriorityLong
42     {
```

```
43      public:
44          enum {
45              MAIN    = 0,
46              HIGH    = 1,
47              NORMAL =  (unsigned(1) << (sizeof(long) * 8 )) - 3,
48              LOW     =  (unsigned(1) << (sizeof(long) * 8 )) - 2,
49              IDLE    = (unsigned(1) << (sizeof(long) * 8 )) - 1
50          };
51
52      public:
53          PriorityLong(unsigned long p = NORMAL): _priority(p) {
54
55
56          db<Priority>(TRC) << "PriorityLong::Priority(" << p << ")\n";
57          }
58
59          operator const volatile unsigned long() const volatile { return
                _priority; }
60
61      public:
62          void priority(unsigned long p) {_priority = p;}
63
64          int times() { return 0; }
65          RTC::Microsecond phase() { return 0; }
66          RTC::Microsecond period() { return 0; }
67
68      protected:
69          volatile unsigned long _priority;
70      };
71
72      // Round-Robin
73      class Round_Robin: public Priority
74      {
75      public:
76          enum {
77              MAIN    = 0,
78              NORMAL = 1,
79              IDLE    = (unsigned(1) << (sizeof(int) * 8 - 1)) -1
80          };
81
82      public:
83          Round_Robin(int p = NORMAL): Priority(p) {}
84      };
```

```
85
86      // First-Come, First-Served (FIFO)
87      class FCFS: public Priority
88      {
89      public:
90          enum {
91              MAIN    = 0,
92              NORMAL = 1,
93              IDLE    = (unsigned(1) << (sizeof(int) * 8 - 1)) -1
94          };
95
96      public:
97          FCFS(int p = NORMAL)
98              : Priority((p == IDLE) ? IDLE : TSC::time_stamp()) {}
99      };
100
101     // Rate Monotonic
102     class RM: public Priority
103     {
104     public:
105         enum {
106             MAIN        = 0,
107             PERIODIC    = 1,
108             APERIODIC = (unsigned(1) << (sizeof(int) * 8 - 1)) -2,
109             NORMAL      = APERIODIC,
110             IDLE        = (unsigned(1) << (sizeof(int) * 8 - 1)) -1
111         };
112
113     public:
114         RM(int p): Priority(p), _deadline(0) {} // Aperiodic
115         RM(const RTC::Microsecond & d): Priority(PERIODIC), _deadline(d) {}
116
117     private:
118         RTC::Microsecond _deadline;
119     };
120
121     // Super class for real-time
122     class RealTime: public PriorityLong
123     {
124     public:
125         RealTime(unsigned long p): PriorityLong(p), _relDeadline(INFINITE)
                {} // aperiodic non real-time
126         RealTime(unsigned long p, const RTC::Microsecond & deadline):
```

```
            PriorityLong(p), _relDeadline(deadline) {/*db<RealTime>(TRC) <<
            "RealTime::RealTime(" << p << "," << deadline << ")\n";*/} //
            Aperiodic real-time
127
128     public:
129         bool operator <=(const RealTime & other) const {return (unsigned
            long)*this < (unsigned long)other;}
130
131     public:
132        RTC::Microsecond deadline() {return _relDeadline;}
133     private:
134        RTC::Microsecond _relDeadline;
135     };
136
137     // Earliest Deadline First
138     class EDF: public Priority
139     {
140     public:
141         enum {
142             MAIN       = 0,
143             PERIODIC   = 1,
144             APERIODIC  = (unsigned(1) << (sizeof(int) * 8 - 1)) -2,
145             NORMAL     = APERIODIC,
146             IDLE       = (unsigned(1) << (sizeof(int) * 8 - 1)) -1
147         };
148
149     public:
150         EDF(int p): Priority(p), _deadline(0) {} // Aperiodic
151         EDF(const RTC::Microsecond & d): Priority(d >> 8), _deadline(d) {}
152
153     private:
154         RTC::Microsecond _deadline;
155     };
156 };
157
158 namespace Scheduling_Scheduler {
159 // Objects subject to scheduling by Scheduler must declare a type "
        Criterion"
160 // that will be used as the scheduling criterion (viz, through operators <,
        >,
161 // and ==) and must also define a method "link" to access the list element
162 // pointing to the object.
163
```

```
164 //escalonador certo:
165 template <typename T>
166 class Scheduler
167 {
168 protected:
169     typedef typename T::Criterion Rank_Type;
170
171     static const bool smp = Traits<Thread>::smp;
172
173     typedef Scheduling_Queue<T, Rank_Type, smp> Queue;
174
175 public:
176     typedef T Object_Type;
177     typedef typename Queue::Element Element;
178
179 public:
180     Scheduler() {}
181
182     unsigned int schedulables() { return _ready.size(); }
183
184     T * volatile chosen() {
185         return const_cast<T * volatile>(_ready.chosen()->object());
186     }
187
188     void insert(T * obj) {
189         db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen()
190                            << "]::insert(" << obj << ")\n";
191         _ready.insert(obj->link());
192     }
193
194     T * remove(T * obj) {
195         db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen()
196                            << "]::remove(" << obj << ")\n";
197         return _ready.remove(obj) ? obj : 0;
198     }
199
200     void suspend(T * obj) {
201         db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen()
202                            << "]::suspend(" << obj << ")\n";
203         _ready.remove(obj);
204     }
205
206     void resume(T * obj) {
```

```
207         db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen()
208                            << "]::resume(" << obj << ")\n";
209         _ready.insert(obj->link());
210     }
211
212     T * choose() {
213         db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen()
214                            << "]::choose() => ";
215         T * obj = _ready.choose()->object();
216         db<Scheduler>(TRC) << obj << "\n";
217         return obj;
218     }
219
220     T * choose_another() {
221         db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen()
222                            << "]::choose_another() => ";
223         T * obj = _ready.choose_another()->object();
224         db<Scheduler>(TRC) << obj << "\n";
225         return obj;
226     }
227
228     T * choose(T * obj) {
229         db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen()
230                            << "]::choose(" << obj;
231         if(!_ready.choose(obj))
232             obj = 0;
233         db<Scheduler>(TRC) << ") => " << obj << "\n";
234         return obj;
235     }
236
237 private:
238     Scheduling_Queue<Object_Type, Rank_Type, smp> _ready;
239 //      Queue<Object_Type, smp, Element> _suspended;
240 };
241
242 template <typename T>
243 class Scheduler_Eduardo: public Scheduler<T>
244 {
245 protected:
246     typedef Scheduler<T> Base;
247     typedef typename Base::Rank_Type Rank_Type;
248
249 public:
```

```
250     Scheduler_Eduardo():
251          _lifetime(Traits<Battery_Lifetime>::lifetime), _priority_bound( ((
                 unsigned(1) << (sizeof(long) * 8 )) - 1)/2 )
252       {
253                         _lifetime.init();
254       }
255
256     void insert(T * obj) {
257
258         if((unsigned long)(obj->link()->rank()) == Rank_Type::IDLE){
259                 _energy_t = obj;
260         }
261                 Base::insert(obj);
262     }
263
264     T * choose_another() {
265
266                 T * obj = Base::choose_another();
267
268                 if(!_lifetime.energy_deficit())
269                         return obj;
270
271                 if( obj->link()->rank() < _priority_bound){
272                         return obj;
273                 }
274
275                 if(obj->is_schedulable()){
276                         obj->update_hunger(-2, _lifetime.
                                get_energectic_deficit_bound());
277                 }
278                 else{
279                         obj->update_hunger(1, _lifetime.
                                get_energectic_deficit_bound());
280                         obj = Base::choose(_energy_t);
281                 }
282
283         return obj;
284     }
285
286     T * choose(){
287
288                 T * obj = Base::choose();
289
```

```
290                     if(!_lifetime.energy_deficit()){
291                             return obj;
292                     }
293
294                     if( obj->link()->rank() < _priority_bound){
295                             return obj;
296                     }
297
298                     if(obj->is_schedulable()){
299                             obj->update_hunger(-2, _lifetime.
                                    get_energectic_deficit_bound());
300                     }
301                     else{
302                             obj->update_hunger(1, _lifetime.
                                    get_energectic_deficit_bound());
303                             obj = Base::choose(_energy_t);
304                     }
305
306                     return obj;
307             }
308
309
310             T * choose(T * obj){
311                 return Base::choose(obj);
312             }
313
314     private:
315             Battery_Lifetime _lifetime;
316             T * _energy_t;
317             unsigned int _priority_bound;
318     };
319     };
320     __END_SYS
321
322     #endif
```

Algoritmo 4 : Código fonte do *scheduler.h*

```
1  // EPOS— Battery Lifetime Estimate Abstraction Declarations
2  //@edd: battery
3
4  #ifndef __battery_lifetime_h
5  #define __battery_lifetime_h
6
7  #include <battery.h>
8  #include <alarm.h>
9  #include <tsc.h>
10
11 __BEGIN_SYS
12 class Battery_Lifetime
13 {
14
15 public:
16     typedef unsigned int Battery_Charge;
17     typedef unsigned long Second;
18
19 public:
20
21     Battery_Lifetime(Second seconds)
22         : _threshold(Traits<Battery_Lifetime>::threshold),
23         _last_energy(true),
24         _duration(seconds),
25         _energy_deficit(0),
26         _energectic_deficit_bound(0)
27         {}
28
29     void init()
30     {
31         _battery_start = (Battery_Charge) remaining_charge();
32         Alarm::reset_n_quantums();
33     }
34
35     void system_lifetime(Second seconds)
36     {
37         _duration = seconds;
38     }
39
40     Second system_lifetime()
41     {
42         return _duration;
```

```
43      }
44
45      unsigned long energy_deficit()
46      {
47          if(!Traits<Battery_Lifetime>::enabled)
48              return 0;
49
50          if(!_duration)
51              return 0;
52
53          _threshold -= 1;
54
55          if(_threshold < 1){
56              _energy_deficit = estimate_deficit();
57
58              _energectic_deficit_bound = ((unsigned(1) << (sizeof(long) * 8
                    )) - 1) - _energy_deficit;
59              _threshold = Traits<Battery_Lifetime>::threshold;
60          }
61
62          return _energy_deficit;
63      }
64
65      unsigned long get_energy_deficit(){
66          return _energy_deficit;
67      }
68
69      unsigned long get_energectic_deficit_bound(){
70          return _energectic_deficit_bound;
71      }
72
73  private:
74
75      unsigned int remaining_charge();
76
77      Second time_interval(){ return (((Second)(Alarm::reset_n_quantums())) *
            Traits<Thread>::MLQUANTUM) / 1000; }
78
79      int estimate_deficit()
80      {
81
82          Second time_end = time_interval();
83
```

```
84          if(time_end >= _duration){
85              _duration = 0;
86              return 0;
87          }
88
89          _duration -= time_end;
90
91          Battery_Charge battery_end = (Battery_Charge) remaining_charge();
92
93          if(battery_end <= BOUNDARY){
94              return ((unsigned(1) << (sizeof(long) * 8 )) - 1)/2;
95          }
96
97          Second estimate_time;
98
99          if(battery_end < _battery_start){
100
101             if((_battery_start - battery_end) <= 0 || time_end <= 0){
102                 return 0;
103             }
104
105             estimate_time = ((Second)((battery_end - BOUNDARY) / (
                    _battery_start - battery_end))) * (time_end);
106
107         } else {
108
109             _battery_start = battery_end;
110             return 0;
111         }
112
113         _battery_start = battery_end;
114
115         if(estimate_time < _duration){
116
117                 double energy_deficit_ratio = (double) _duration /
                        estimate_time;
118
119                 if(energy_deficit_ratio > 1 + ((double)Traits<
                        Battery_Lifetime>::energectic_deficit_coeficient/100)){
120                     return 32767;
121                 }
122
123                 return 32767*( energy_deficit_ratio - 1)/((double)Traits<
```

```
                    Battery_Lifetime >::energectic_deficit_coeficient/100);
124         }
125
126         return 0;
127     }
128
129
130 private:
131     Battery_Charge _battery_start;
132     static const unsigned int BOUNDARY = Traits<Battery_Lifetime >::boundary
            ;
133     Second _duration;
134     double _threshold;
135
136     unsigned long _energy_deficit;
137     unsigned long _energectic_deficit_bound;
138 };
139
140 __END_SYS
141
142 #endif
```

Algoritmo 5 : Código fonte do *battery-lifetime.h*

```
1  // EPOS— Thread Abstraction Declarations
2
3  #ifndef __thread_h
4  #define __thread_h
5
6  #include <system/kmalloc.h>
7  #include <utility/queue.h>
8  #include <utility/handler.h>
9  #include <cpu.h>
10 #include <scheduler.h>
11
12 __BEGIN_SYS
13
14 class Thread
15 {
16
17     friend class Handler_Thread;     // to provide access to _preemptive
           attribute
18
19 protected:
20     static const unsigned int STACK_SIZE
21     = Traits<Machine>::APPLICATION_STACK_SIZE;
22
23     static const bool idle_waiting = Traits<Thread>::idle_waiting;
24     static const bool active_scheduler = Traits<Thread>::active_scheduler;
25     static const bool preemptive = Traits<Thread>::preemptive;
26     static const bool smp = Traits<Thread>::smp;
27
28     static const unsigned int QUANTUM = Traits<Thread>::QUANTUM;
29     typedef CPU::Log_Addr Log_Addr;
30     typedef CPU::Context Context;
31
32 public:
33     // Thread State
34     enum State {
35         BEGINNING,
36         READY,
37         RUNNING,
38         SUSPENDED,
39         WAITING,
40         FINISHING
41     };
```

```
42
43          typedef Traits<Thread>::Scheduler Scheduler;
44
45      // Thread Scheduling Criterion
46      typedef Traits<Thread>::Criterion Criterion;
47      enum {
48          NORMAL = Criterion::NORMAL,
49          MAIN = Criterion::MAIN,
50          IDLE = Criterion::IDLE
51      };
52
53      typedef Criterion Priority;
54
55
56      typedef Ordered_Queue<Thread, Criterion, smp, Scheduler::Element> Queue
            ;
57
58  public:
59      Thread(int (* entry)(),
60              const State & state = READY,
61              const Criterion & criterion = NORMAL,
62              unsigned int stack_size = STACK_SIZE)
63          : _state(state), _waiting(0), _joining(0), _link(this, criterion),
               _hunger(0), _schedulable(true)
64      {
65          prevent_scheduling();
66
67          _stack = kmalloc(stack_size);
68          _context = CPU::init_stack(_stack, stack_size, &implicit_exit,
               entry);
69
70          common_constructor(entry, stack_size);
71      }
72      template<typename T1>
73      Thread(int (* entry)(T1 a1), T1 a1,
74              const State & state = READY,
75              const Criterion & criterion = NORMAL,
76              unsigned int stack_size = STACK_SIZE)
77          : _state(state), _waiting(0), _joining(0), _link(this, criterion),
               _hunger(0), _schedulable(true)
78      {
79          prevent_scheduling();
80
```

```
81          _stack = kmalloc(stack_size);
82          _context = CPU::init_stack(_stack, stack_size, &implicit_exit,
                entry,
83                                     a1);
84
85          common_constructor(entry, stack_size);
86      }
87      template<typename T1, typename T2>
88      Thread(int (* entry)(T1 a1, T2 a2), T1 a1, T2 a2,
89             const State & state = READY,
90             const Criterion & criterion = NORMAL,
91             unsigned int stack_size = STACK_SIZE)
92          : _state(state), _waiting(0), _joining(0), _link(this, criterion),
                _hunger(0), _schedulable(true)
93      {
94          prevent_scheduling();
95
96          _stack = kmalloc(stack_size);
97          _context = CPU::init_stack(_stack, stack_size, &implicit_exit,
                entry,
98                                     a1, a2);
99
100         common_constructor(entry, stack_size);
101     }
102     template<typename T1, typename T2, typename T3>
103     Thread(int (* entry)(T1 a1, T2 a2, T3 a3), T1 a1, T2 a2, T3 a3,
104            const State & state = READY,
105            const Criterion & criterion = NORMAL,
106            unsigned int stack_size = STACK_SIZE)
107         : _state(state), _waiting(0), _joining(0), _link(this, criterion),
                _hunger(0), _schedulable(true)
108     {
109         prevent_scheduling();
110
111         _stack = kmalloc(stack_size);
112         _context = CPU::init_stack(_stack, stack_size, &implicit_exit,
                entry,
113                                    a1, a2, a3);
114
115         common_constructor(entry, stack_size);
116     }
117     ~Thread();
118
```

```
119     const volatile State & state() const { return _state; }
120     const volatile Criterion & criterion() const { return _link.rank(); }
121
122     Priority  priority() const { return _link.rank(); }
123     void priority(const Priority & p);
124
125     int join();
126     void pass();
127     void suspend();
128     void resume();
129     void ready_enqueue();  // resume without reschedule
130
131     void update_hunger(int new_hunger, unsigned int energy_deficit);
132     unsigned long hunger();
133     bool is_schedulable();
134
135     static Thread * self() { return running(); }
136     static void yield();
137     static void sleep(Queue * q);
138     static void wakeup(Queue * q);
139     static void wakeup_all(Queue * q);
140     static void exit(int status = 0);
141
142     static void init();
143
144 public:
145     Queue::Element * link() { return &_link; }
146
147 protected:
148     void common_constructor(Log_Addr entry, unsigned int stack_size);
149
150     static Thread * volatile running() { return _scheduler.chosen(); }
151
152     static void prevent_scheduling() {
153         if(active_scheduler)
154             CPU::int_disable();
155     }
156     static void allow_scheduling() {
157         if(active_scheduler)
158             CPU::int_enable();
159     }
160
161     static void reschedule();
```

```
162     static void time_reschedule(); // this is the master alarm handler
163
164     static void implicit_exit();
165
166     static void switch_threads(Thread * prev, Thread * next) {
167         // scheduling must be disabled at this point!
168         if(next != prev) {
169             if(prev->_state == RUNNING)
170                 prev->_state = READY;
171             next->_state = RUNNING;
172             db<Thread>(TRC) << "Thread::switch_threads(prev=" << prev
173                             << ",next=" << next << "\n";
174             CPU::switch_context(&prev->_context, next->_context);
175         }
176         allow_scheduling();
177     }
178
179     static int idle();
180
181
182
183 protected:
184     Log_Addr _stack;
185     Context * volatile _context;
186     volatile State _state;
187     Queue * _waiting;
188     Thread * volatile _joining;
189     Queue::Element _link;
190
191     unsigned long _hunger;
192     bool _schedulable;
193
194     static unsigned int _thread_count;
195     static Scheduler _scheduler;
196 };
197
198 // A thread event handler (see handler.h)
199 class Handler_Thread : public Handler
200 {
201 public:
202     Handler_Thread(Thread * h) : _handler(h) {}
203     ~Handler_Thread() {}
204
```

```
205     void operator()() {
206       _handler->ready_enqueue();
207     }
208
209 private:
210     Thread * _handler;
211 };
212
213 __END_SYS
214
215 #endif
```

Algoritmo 6 : Código fonte do *thread.h*

```
1  // EPOS— Thread Abstraction Implementation
2
3  #include <system/kmalloc.h>
4  #include <thread.h>
5  #include <alarm.h>
6  #include <machine.h>
7
8  __BEGIN_SYS
9
10 // Class attributes
11 unsigned int Thread::_thread_count = 0;
12 Thread::Scheduler Thread::_scheduler;
13
14 // Methods
15 void Thread::common_constructor(Log_Addr entry, unsigned int stack_size)
16 {
17     db<Thread>(TRC) << "Thread(entry=" << (void *)entry
18                     << ",state=" << _state
19                     << ",rank=" << _link.rank()
20                     << ",stack={b=" << _stack
21                     << ",s=" << stack_size
22                     << "},context={b=" << _context
23                     << "," << *_context << "}) => " << this << "\n";
24
25     _thread_count++;
26
27     if((_state == READY) || (_state == RUNNING))
28         _scheduler.insert(this);
29
30     if(preemptive)
```

```
31          reschedule();
32
33      allow_scheduling();
34  }
35
36  Thread::~Thread()
37  {
38      prevent_scheduling();
39
40      db<Thread>(TRC) << "~Thread(this=" << this
41                      << ",state=" << _state
42                      << ",rank=" << _link.rank()
43                      << ",stack={b=" << _stack
44                      << ",context={b=" << _context
45                      << "," << *_context << "})\n";
46
47      switch(_state) {
48      case BEGINNING:   _scheduler.resume(this); break;
49      case RUNNING: exit(-1); break; // Self deleted itself!
50      case READY: break;
51      case SUSPENDED: _scheduler.resume(this); break;
52      case WAITING: _waiting->remove(this); _scheduler.resume(this); break;
53      case FINISHING: break;
54      }
55
56      _scheduler.remove(this);
57
58      allow_scheduling();
59
60      kfree(_stack);
61  }
62
63  void Thread::priority(const Priority & p)
64  {
65      prevent_scheduling();
66
67      db<Thread>(TRC) << "Thread::priority(this=" << this
68                      << ",prio=" << p << ")\n";
69
70      _link.rank(p);
71
72      if(preemptive)
73          reschedule();
```

```
74
75      allow_scheduling();
76  }
77
78  int Thread::join()
79  {
80      prevent_scheduling();
81
82      db<Thread>(TRC) << "Thread::join(this=" << this
83                      << ",state=" << _state << ")\n";
84
85      if(_state != FINISHING) {
86          _joining = running();
87          _joining->suspend(); // implicitly allows scheduling
88      }
89
90      allow_scheduling();
91
92      return *static_cast<int *>(_stack);
93  }
94
95  void Thread::pass()
96  {
97      prevent_scheduling();
98
99      db<Thread>(TRC) << "Thread::pass(this=" << this << ")\n";
100
101     Thread * prev = running();
102
103     Thread * next = _scheduler.choose(this);
104     if(next)
105         switch_threads(prev, next);
106     else
107         db<Thread>(WRN) << "Thread::pass => thread (" << this
108                         << ") not ready\n";
109
110     allow_scheduling();
111 }
112
113 void Thread::suspend()
114 {
115     prevent_scheduling();
116
```

```
117     db<Thread>(TRC) << "Thread::suspend(this=" << this << ")\n";
118
119     Thread * prev = running();
120
121     _scheduler.suspend(this);
122     _state = SUSPENDED;
123
124 //grw Troquei de running para choose
125     Thread * next = _scheduler.choose();
126
127     Alarm::reset_master();
128     switch_threads(prev, next); // null if this != running() at the begin
129
130     allow_scheduling();
131 }
132
133 void Thread::ready_enqueue()
134 {
135     prevent_scheduling();
136
137     db<Thread>(TRC) << "Thread::ready_enqueue(this=" << this << ")\n";
138     if(_state == SUSPENDED) {
139         _state = READY;
140         _scheduler.resume(this);
141     } else
142         db<Thread>(WRN) << "Ready_enqueue called for unsuspended object!\n"
            ;
143
144     //allow_scheduling();    // alarm will allow
145 }
146
147 void Thread::resume()
148 {
149     prevent_scheduling();
150
151     db<Thread>(TRC) << "Thread::resume(this=" << this << ")\n";
152
153     if(_state == SUSPENDED) {
154         _state = READY;
155         _scheduler.resume(this);
156     } else
157         db<Thread>(WRN) << "Resume called for unsuspended object!\n";
158
```

```
159     if(preemptive)
160         reschedule();
161
162     allow_scheduling();
163 }
164
165
166 // Class methods
167
168 void Thread::yield()
169 {
170     prevent_scheduling();
171
172     db<Thread>(TRC) << "Thread::yield(running=" << running() << ")\n";
173         Thread * prev = running();
174         Thread * next = _scheduler.choose_another();
175     Alarm::reset_master();
176         switch_threads(prev, next);
177
178     allow_scheduling();
179 }
180
181 void Thread::exit(int status)
182 {
183     prevent_scheduling();
184
185     db<Thread>(TRC) << "Thread::exit(running=" << running()
186                     <<",status=" << status << ")\n";
187
188     Thread * thr = running();
189     _scheduler.remove(thr);
190     *static_cast<int *>(thr->_stack) = status;
191     thr->_state = FINISHING;
192
193     _thread_count--;
194
195     if(thr->_joining) {
196         thr->_joining->_state = READY;
197         _scheduler.resume(thr->_joining);
198         thr->_joining = 0;
199     }
200
201     Alarm::reset_master();
```

```
202     switch_threads(thr, _scheduler.choose());
203
204     allow_scheduling();
205 }
206
207 void Thread::sleep(Queue * q)
208 {
209     prevent_scheduling();
210
211     db<Thread>(TRC) << "Thread::sleep(running=" << running()
212                     << ",q=" << q << ")\n";
213
214     Thread * thr = running();
215
216     _scheduler.suspend(thr);
217     thr->_state = WAITING;
218     q->insert(&thr->_link);
219     thr->_waiting = q;
220
221     Alarm::reset_master();
222 //grw Troquei de chosen para choose
223     switch_threads(thr, _scheduler.choose());
224
225     allow_scheduling();
226 }
227
228 void Thread::wakeup(Queue * q)
229 {
230     prevent_scheduling();
231
232     db<Thread>(TRC) << "Thread::wakeup(running=" << running()
233                     << ",q=" << q << ")\n";
234
235     if(!q->empty()) {
236         Thread * t = q->remove()->object();
237         t->_state = READY;
238         t->_waiting = 0;
239         _scheduler.resume(t);
240     }
241
242     if(preemptive)
243         reschedule();
244
```

```
245     allow_scheduling();
246 }
247
248 void Thread::wakeup_all(Queue * q)
249 {
250     prevent_scheduling();
251
252     db<Thread>(TRC) << "Thread::wakeup_all(running=" << running()
253                     << ",q=" << q << ")\n";
254
255     while(!q->empty()) {
256         Thread * t = q->remove()->object();
257         t->_state = READY;
258         t->_waiting = 0;
259         _scheduler.resume(t);
260     }
261
262     if(preemptive)
263         reschedule();
264
265     allow_scheduling();
266 }
267
268 void Thread::time_reschedule()
269
270 {
271     // timer invokes the master handler with interrupts enabled!
272
273     prevent_scheduling();
274         Thread * prev = running();
275         Thread * next = _scheduler.choose();
276
277         switch_threads(prev, next);
278
279     // scheduling will be reenabled by switch_threads
280 }
281
282 void Thread::reschedule()
283 {
284     // scheduling must be disabled at this point
285
286     Thread * prev = running();
287     Thread * next = _scheduler.choose();
```

```
288
289         Alarm::reset_master();
290         switch_threads(prev, next);
291
292         // scheduling will be reenabled by switch_threads
293     }
294
295     void Thread::implicit_exit()
296     {
297         exit(CPU::fr());
298     }
299
300     int Thread::idle()
301     {
302
303     //TODO     Temperature_Sensor sensor; // = new Temperature_Sensor();
304
305         while(true) {
306             db<Thread>(TRC) << "Thread::idle()\n";
307
308             if(_thread_count <= 1) {
309                 db<Thread>(WRN) << "The last thread has exited!\n";
310                 db<Thread>(WRN) << "Halting the CPU ...\n";
311                 CPU::int_disable();
312             }
313
314     //TODO   sensor.disable();
315
316             CPU::halt();
317
318             if(_scheduler.schedulables() > 1)
319                 yield();
320         }
321
322         return 0;
323     }
324
325     void Thread::update_hunger(int new_hunger, unsigned int
        energectic_deficit_bound){
326
327             if( (volatile unsigned long) priority() > energectic_deficit_bound)
                    {
328                     _hunger += new_hunger;
```

```
329
330                 db<Thread>(TRC) << " > Thread " << link()->object() << "
                        teve sua fome atualizada! valor atual = " << _hunger <<
                        "\n";
331
332                 if((unsigned long) priority() <= energectic_deficit_bound +
                        _hunger and new_hunger>0){
333                     db<Thread>(TRC) << " > Thread " << link()->object()
                            << " ser  deshibernada! \n";
334
335                     // deficit de energia muito critico - tarefas
                            ser o escalonadas apenas 10 vezes apenas para
                            n o ficarem famintas de recursos.
336                     if(energectic_deficit_bound <= ( ((unsigned(1) << (
                            sizeof(long) * 8 )) - 1)*0.75)){
337                         db<Thread>(TRC) << " > Deshibernada com op
                                1 [deficit de energia cr tico]! \n";
338                         _hunger += 10;
339                     }else{
340                                 // deficit de energia n o critico.
341                                 db<Thread>(TRC) << " > Deshibernada
                                        com op 2! \n";
342
343                                 unsigned long biggest_increment =
                                        ((unsigned(1) << (sizeof(long) *
                                         8 )) - 1) -
                                        energectic_deficit_bound -
                                        _hunger;
344
345                                 _hunger += biggest_increment*0.5;
346                         }
347
348                     _schedulable = true;
349                 }
350
351                 if((volatile unsigned long) priority() >=
                        energectic_deficit_bound + _hunger and new_hunger<0){
352
353                     db<Thread>(TRC) << " > Thread " << link()->object()
                            << " ser  hibernada! \n";
354                     _hunger = 0;
355                     _schedulable = false;
356                 }
```

```
357             }
358 }
359
360 unsigned long Thread::hunger(){
361             return _hunger;
362 }
363
364 bool Thread::is_schedulable(){
365             return _schedulable;
366 }
367
368 __END_SYS
```

Algoritmo 7 : Código fonte do *thread.cc*

Gerenciamento Dinâmico de Energia em Sistemas Embarcados de Tempo Real

Eduardo M. Steiner

Departamenton de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

eduardo.steiner@gmail.com

Abstract. *Embedded systems are computing systems usually dedicated to a specific application. Embedded systems projects are characterized by the great number of constraints such as real time, reliability, energectic eficiency, etc. This work presents a power aware real time scheduler for embedded system. This scheduler must have knowledge about the energy levels in the system battery, and must make the system reach the time defined for it's duration. The main focus in this work is to provide the greatest QoS possible to non-real-time tasks, since it doesn't get conflict with the scheduling of real-time tasks and doesn't affect the setted duration the system must reach. The proposed scheduler was created in EPOS system (Embedded Paralell Operating System), and uses its real-time scheduling and power management structures.*

Resumo. *Sistemas embarcados são sistemas computacionais que geralmente são dedicados à uma aplicação específica. Seus projetos são caracterizados por possuírem um grande número de restrições como tempo real, confiabilidade, eficiência energética, etc. Este trabalho apresenta uma proposta de um escalonador de tempo real para sistemas embarcados cujo a energia é fornecida por bateias. Esse escalonador deverá ter noção dos níveis de energia nas baterias do sistema e fazer com que o tempo de duração definido para ele seja alcançado. O principal foco do trabalho, é fazer com que as tarefas que não possuam requisitos de tempo real tenham a melhor QoS possível, desde que isso não entre em conflito com o escalonamento das tarefas de tempo real, e não prejudique o tempo que o sistema tem que durar. O escalonador proposto foi criado no sistema EPOS(Embedded Paralell Operating System), e faz uso de suas estruturas de escalonamento de tempo real e gerenciamento de energia.*

1. Introdução

O crescente número de sistemas computacionais presentes no dia a dia dos indivíduos da sociedade é um fato levantado por muitos pesquisadores [Marwedel 2003, Weiser 1993]. Estudos [Marwedel 2003] mostram que um americano comum entra em contato com uma média de 60 processadores diariamente. A maior parte desses processadores não se encontra em computadores de propósito geral, mas sim em sistemas embarcados.

Sistemas embarcados são sistemas computacionais, que se caracterizam por estar completamente embarcados em um produto maior. Geralmente possuem restrições

temporais (em alguns casos críticas), de eficiência computacional e energética. Sistemas embarcados estão presentes em carros, televisões, máquinas de lavar e em uma infinidade de dispositivos; os quais entramos em contato todos os dias. Seguindo o sucesso da tecnologia da informação para aplicações de escritórios e fluxo de trabalho, a área de sistemas embarcados está sendo considerada a mais importante da teclonogia da informação dos próximos anos [Marwedel 2003].

De acordo com Eggermont [Eggermonot 2002] a energia é considerada a restrição mais importante em sistemas embarcados. No caso de sistemas cujo a energia provém de baterias, essa importancia se torna ainda maior uma vez que isso implicará diretamente no tempo de vida do sistema. Desse modo, é possível ver a grande importância que o gerenciamento de energia tem, no projeto de sistemas embarcados.

1.1. Motivação

Dada a relevancia da economia energética no projeto de sistemas embarcados, nota-se que a proposta de novas abordagens para o gerenciamento de energia é muito importante - motivo o qual fez com que um crescente número de cientistas da industria e academia tenham cada vez mais pesquisado sobre o assunto.

Apesar disso, essa área ainda reune muitos desafios. O gerenciamento de energia eficaz em sistemas embarcados não é algo simples de se desenvolver, pois deve-se levar em conta diversos itens como por exemplo implicações em performance que modos de consumo mais baixos possam causar, overheads associados à trocas de modos de operações, influências que diferentes técnicas exercem uma sobre a outra entre outros.

O *hardware* de sistemas embarcados costuma reunir caracteristicas de eficiência e economia energética, porém elas ainda não são usadas de modo ideal pelo software de gerencia de energia. Desse modo, as novas abordagens para a gerência de energia por software devem ser propostas e estudadas afim de se alcançar um melhor uso dessas características.

1.2. Objetivo

O principal objetivo deste trabalho é a implementação de um escalonador sistemas embarcados de tempo real, o qual tenha noção dos níveis de energia presente nas baterias do sistema.

Como a definição de um tempo mínimo o qual o sistema deverá durar é uma pratica muito comum no projeto de sistemas embarcados, um dos objetivos do escalonador proposto será fazer com que o tempo de duração atribuído ao sistema seja alcançado.

Uma vez que diferentes tarefas do sistema têm diferentes requisitos (de tempo real, importância para a aplicação, etc), a abordagem proposta irá classificar as tarefas em dois grupos: hard real-time (tempo real rígido) e best-effort (melhor esforço).

Nesse contexto, os seguintes requisitos foram levantados:

- Tarefas *hard real-time* deverão sempre ter seus prazos respeitados e prioridade de execução sobre as tarefas *best-effort*.

- O escalonador deverá fazer com que o tempo estipulado para o sistema pelo programador da aplicação seja alcançado. Caso seja constatado que o nível de energia nas baterias não seja o suficiente para isso, ele deverá efetuar o descarte das tarefas *best-effort*, e escalonar tarefas energeticamente mais economicas em seu lugar.
- Nenhuma tarefa *best-effort* deve ficar "faminta" de recursos, todas as tarefas devem ter a sua chance na CPU (desde que isso não entre em conflito com os itens anteriores).
- A QoS(Qualidade de Serviço) atribuída as tarefas *best-effort* deverá ser proporcional à carga de energia presente nas baterias do sistema em relação à quantidade de tempo que falta para se alcançar o tempo estipulado para a duração do sistema.

O restante do texto é organizado da seginte forma: A sessão 2 apresenta a abordagem proposta, e mostra como os mecanismos implementados para se alcançar os objetivos propostos. O capitulo 3 descreve um estudo de caso, e apresenta a análise dos resultados. No capitulo 4 é apresentado as considerações finias do trabalho.

2. Abordagem Proposta

Nessa sessão apresentamos a abordagem proposta em detalhes, explicando como é o gerenciamento de energia do EPOS, o que foi modificado e as estruturas mantidas. Conforme levantado na introdução, as maiores motivações por traz desse trabalho são fazer com que o sistema alcance o tempo de vida definido a ele e fornecer a maior QoS possível às suas tarefas *best-effort*.

O termo QoS (acrônimo de *"Quality of Service"*, em português "Qualidade de Serviço") designa a capacidade de fornecer um serviço, o qual pode ter um grau de satisfação para seu usuário bastante variável, dependendo da configuração de uma série de características qualitativas e quantitativas. No contexto desse trabalho, a QoS atribuída às tarefas best-effort pode ser encarada como o seu percentual de descarte (um percentual alto significa uma baixa QoS, e vice versa).

2.1. EPOS

O EPOS (Embedded Paralell Operating System) [Frölich 1999] é um sistema operacional orientado a aplicação que foi concebido segundo a ADESD (Application-Driven Embedded System Design) [Frölich 2001], e permite ao programador da aplicação gerar sistemas específicos, agregando apenas os componentes necessários à aplicação definida, que são selecionados e configurados automaticamente por uma série de ferramentas.

Esse foi o sistema escolhido para implementação do modelo proposto pois ele possui excelentes características de escalonamento de tarefas e gerenciamento de energia. Além disso, agregar e desenvolver componentes de software para ele é simples e prático, devido à sua arquitetura.

2.2. Modelagem das tarefas

Para esse trabalho, as tarefas serão abstraídas como *threads*, e precisarão ser *hard real-time* ou *best-effort*. Como todas as suas funcionalidades serão as mesmas, não houve a necessidade de implementar um novo objeto; portanto ambas serão o próprio objeto ***thread*** do EPOS.

A prioridade das threads será o que irá as diferenciar. Como as *threads hard real-time* devem sempre ter prioridade de execução superior às *best-effort*, nesse trabalho iremos modelar uma "fronteira de prioridade" da seguinte maneira:

- ***threads hard real-time*** terão sua prioridade entre 0 e (**max int**)/2
- ***threads best-effort*** terão sua prioridade entre (**max int**)/2 e **max int**

Onde **max int** é o maior inteiro possível na plataforma alvo. O valor das prioridades seguirá o atual modelo do EPOS, onde quanto menor for o valor, maior será sua posição na fila de escalonamento (i.e. maior a sua prioridade).

As tarefas fundamentais do sistema (ou que possuem requisitos de tempo real) deverão ser expressadas pelo programador da aplicação como *threads hard real-time*. Já as tarefas que possuem um caráter opcional poderão ser expressadas como *threads best-effort*.

Em sistemas operacionais, quando há diferentes categorias de tarefas é comum ser utilizada diferentes filas (uma para cada categoria). Porém, como o trabalho se encontar no contexto de sistemas embarcados, sistemas os quais o número de tarefas é bastante reduzido, foi decidido utilizar apenas uma fila, afim de se evitar sobrecustos.

2.3. Medições nas cargas das baterias

O EPOS conta com um monitor, que verifica a tensão existente nas baterias, e pode determinar a quantidade aproximada de energia restante nelas [Gonçalves 2007]. Isso é um pilar muito importante para o seu gerenciamento de energia, uma vez que é esse mecanismo que possibilita os cálculos para verificar se o tempo atribuído ao sistema será alcançado.

As medições nas baterias do sistema costumam causar um sobrecusto relativamente alto em termos de energia(maiores detalhes desse sobrecusto serão dados nos estudos de casos). Por isso, as medições deverão ocorrer com a menor frequência possível.

Nesse trabalho iremos fazer com que ao se verificar que o tempo estipulado para o sistema não será alcançado, as medições não fiquem mais frequentes, ao contrário do atual modelo implementado no EPOS [Wiendenhoft e Frölich 2008]. Uma desvantagem para realização de medições com frequências menores é que o sistema demorará mais tempo para se dar conta de que as tarefas de carater opcional podem voltar a serem escalonadas normalmente.

Por outro lado, o sistema terá um **menor consumo de energia acumulado** (uma vez que o sobrecusto associado as medições será menor). Como nessa abordagem a QoS

será diretamente proporcional aos níveis de energia existentes na bateria do sistema, a desvantagem citada não será tão impactante.

2.4. Cálculo de tempo e energia

O cálculo do tempo e energia é possível no EPOS, pois além do monitor de carga de bateria, o sistema armazena a quantidade de energia da ultima medição, o tempo decorrido desde o início da aplicação, e o tempo que a aplicação deve alcançar.

Em [Wiedenhoft 2008], a seguinte equação foi proposta, para verificar se o tempo definido ao sistema será alcançado:

$$\frac{E_{tk}}{E_{tk-1} - E_{tk}} \times (T_{tk-1} - T_{tk}) \leq T_{tk}$$

Equação 2.1

Onde Ttk é o tempo que o sistema ainda deve durar, Etk a carga da bateria no instante, Ttk-1 e Etk-1 são o tempo que o sistema deveria durar e a carga da bateria no instante **k-1** (i.e. na medição anterior). A equação verifica se a perda de carga foi proporcionalmente maior do que o tempo decorrido.

Nesse trabalho, utilizaremos o termo **deficit de energia** para denotar que o sistema não alcançará o tempo estipulado a ele. Como nesse trabalho há a necessidade de uma informação mais precisa em relação ao quão crítica está a carga de bateria em relação a esse tempo (ou quão crítico está o deficit de energia), ao invés de um valor booleano, utilizaremos um valor do tipo inteiro como resultado da equação que realiza a verificação.

A equação 2.2 é usada para se calcular o tempo estimado para o sistema:

$$TE = \frac{E_{tk}}{E_{tk-1} - E_{tk}} \times (T_{tk-1} - T_{tk})$$

Equação 2.2

Usamos a saída da equação a cima (TE - *Tempo estimado*) para calcular a *taxa de deficit de energia*, por sua vez será utilizada para o cálculo do deficit de energia. As equações para o cálculo da *taxa de deficit de energia* e do *deficit de energia* são as seguintes:

$$TDE = \frac{T_{tk}}{TE}$$

Equação 2.3

$$DE = \frac{maxint}{2} \times \frac{TDE - 1}{CDE};$$

Equação 2.4

A **taxa de deficit de energia** (Equação 4.3) é calculada pela divisão do tempo que o sistema deve durar pelo tempo estimado.

O *deficit de energia* terá um valor entre zero e **maxint**/2. Isso acontece porque ele deverá significar um bloqueio de prioridade (isso será explicado em detalhes na próxima seção). Como apenas as tarefas *best-effort* deverão ser descartadas em função desse deficit, o valor não deve exceder esse limite.

A equação 4.4 que calcula o *deficit de energia* fará uso da *taxa do deficit de energia* e do *CDE* (Coeficiente do deficit de energia), uma variável que pode ter valor configurado pelo programador da aplicação e será usada para indicar o quão radical será o sistema ao entrar em deficit de energia. A Figura 5 mostra o deficit de energia calculado com diferentes coeficientes de deficit de energia, para uma arquitetura onde o maior inteiro é 65535 (logo o maior deficit de energia é 32767):

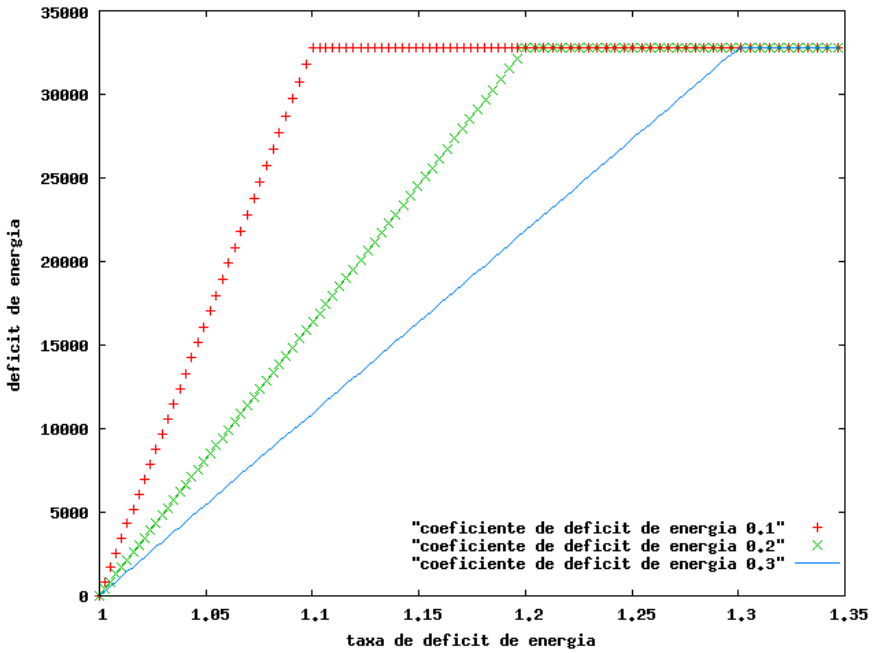

Figura 1:Deficit de energia calculado em função da taxa do deficit de energia para diferentes coeficientes do deficit de energia

2.5. Critério para descarte de tarefas

Caso os níveis de energia nas baterias do sistema estejam críticos(i.e. o tempo definido para o sistema não será alcançado), o sistema deverá ter um comportamento diferente, economizando energia até ser constado que há energia o suficiente nas baterias para alcançar o tempo de duração definido ao sistema.

Para fazer isso, a atual implementação do gerenciamento de energia do EPOS [Wiendenhoft e Frölich 2008] propôs um modelo no qual o sistema realiza descartes de tarefas opcionais. De modo semelhante, nessa implementação, o sistema passará a ter dois comportamentos distintos em função da energia de suas baterias em relação ao tempo que deverá ser alcançado:

- **ausência de deficit de energia no sistema**: as tarefas do sistema serão escalonadas normalmente.
- **presença de deficit de energia no sistema**: as tarefas de tempo real continuarão a ser escalonadas normalmente, porém as *best-effort* poderão ser descartadas, e no lugar delas será escalonada uma tarefa ociosa, a qual possui um comportamento econômico no ponto de vista energético.

A Figura 2 ilustra a faixa de prioridade do sistema, que varia de 0 (prioridade mais alta) à **maxint**(prioridade mais baixa). Nela há também um exemplo da faixa de prioridade das tarefas que são afetadas por um deficit de energia. Essa fronteira de prioridade que chamaremos de **limite do deficit de energia** serve para separar as tarefas *best-effort* que sofrerão algum descarte das que não sofrerão nenhum.

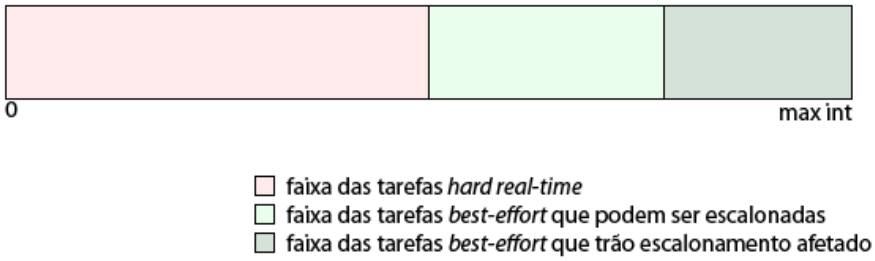

Figura 2: Faixa de prioridade do sistema em deficit de energia: tarefas *hard real-time*, *best-effort* que não sofreram descarte e *best-effort* que sofrerão

Apesar das tarefas *best-effort* estarem sujeitas a descartes, elas não devem ficar famintas de recursos. Desse modo, foi criado um atributo **fome**, que faz com que as tarefas *best-effort* possam, independente dos níveis de energia no qual o sistema se encontre e da prioridade da tarefa em questão, ser escalonadas em algum momento. O escalonador do sistema, conforme será introduzido na próxima seção, irá então, realizar o descarte das tarefas baseado no deficit de energia, na prioridade da tarefa e na sua fome.

O Algoritmo 1 apresenta um pseudo-código referente à atualização da fome das threads. Além de atualizar a fome, esse mecanismo serve para hibernar e deshibernar *threads* que estão sujeitas à descarte.

```
update_hunger(new_hunger, energectic_deficit_bound){

        //verifica  o se a thread est  entre as afetadas pelo deficit
      if( priority() > energectic_deficit_bound){

            _hunger += new_hunger;

                //tarefa vai desihibernar
            if( priority() <= energectic_deficit_bound + _hunger and
                new_hunger>0){

                    if( energectic_deficit_bound <= ( max_int * 0.75 ) ){
                            _hunger += 10;
                    }else{
                                    _hunger += max_int -
                                        energectic_deficit_bound;
                    }
            }

                //tarefa vai hibernar
            if( priority() >= energectic_deficit_bound + _hunger and
                new_hunger<0){
                    _hunger = 0;
            }
}
```

Algoritmo 1: Atualização da fome da *thread*

O método recebe dois inteiros: **new hunger** (nova fome) que servirá para incrementar a fome da *thread*, e **energectic deficit bound** (limite do deficit de energia) que conforme dito anteriormente, indica o deficit de energia do sistema em relação à prioridade das tarefas. O inteiro *new hunger* terá um valor negativo caso a *thread* que esteja tendo a sua fome atualizada esteja sendo escalonada, ou com um valor positivo caso ela não esteja.

Uma *thread* que está sujeita a descarte tem o valor atribuído à sua prioridade maior que o limite do deficit de energia, logo se a sua fome for zero, ela não terá possibilidade de ser escalonada e estará **hibernada**. Cada vez que ela deixar de ser escalonada, a sua fome vai ser incrementada até que a fome, somada com o *limite do deficit de energia* tenha um valor maior que a sua prioridade. Aí então essa *thread* será **deshibernada**, passará a ser escalonada, e o mecanismo usado para isso é o reaproveitamento do seu próprio atributo fome que passará a sofrer um grande incremento em seu valor. A razão para isso é que o escalonador faz a verificação do descarte ou não da tarefa usando esse atributo, e desse modo foi possível obter uma maior eficiência de código.

Caso o sistema esteja com um nível grave de deficit de energia (entre o pior possível e o nível médio), a fome da tarefa que estiver deshibernando será incrementado em apenas 10 (o suficiente para ela ser escalonada dez vezes).

A Figura 3 e a Figura 4, ilustram um exemplo onde há o número de descartes de tarefas *best-effort*, medições nas baterias e o deficit de energia no sistema em função do tempo. No caso, em ambos os diagramas, fizemos suposições de dois deficits de energia, um pequeno e um grande, e ilustramos o comportamento do sistema em termos de descarte de tarefas e frequências de medições. O primeiro diagrama é referente ao modelo atual do sistema EPOS [Wiendenhoft e Frölich 2008], e o segundo o modelo que pretendemos implementar nesse trabalho.

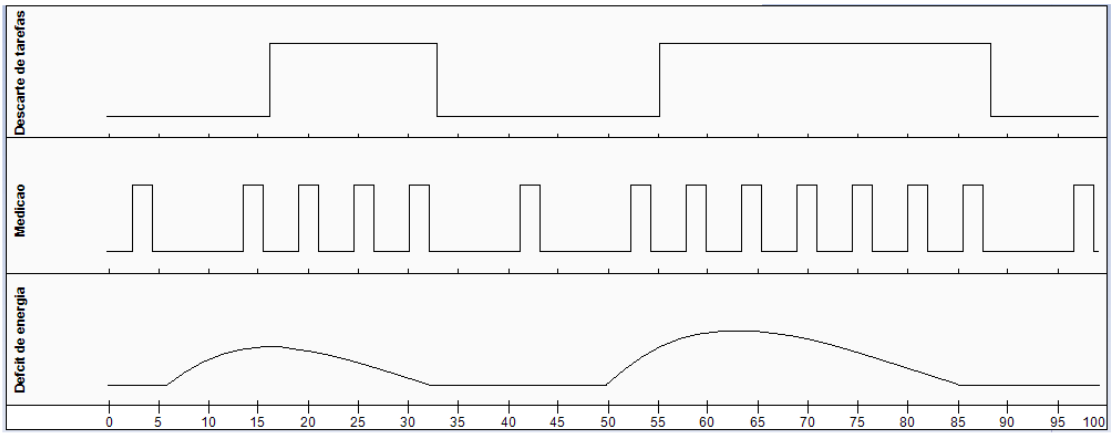

Figura 3: Modelo atual do EPOS referente ao descarte de tarefas, e frequência de medições em função do deficit de energia

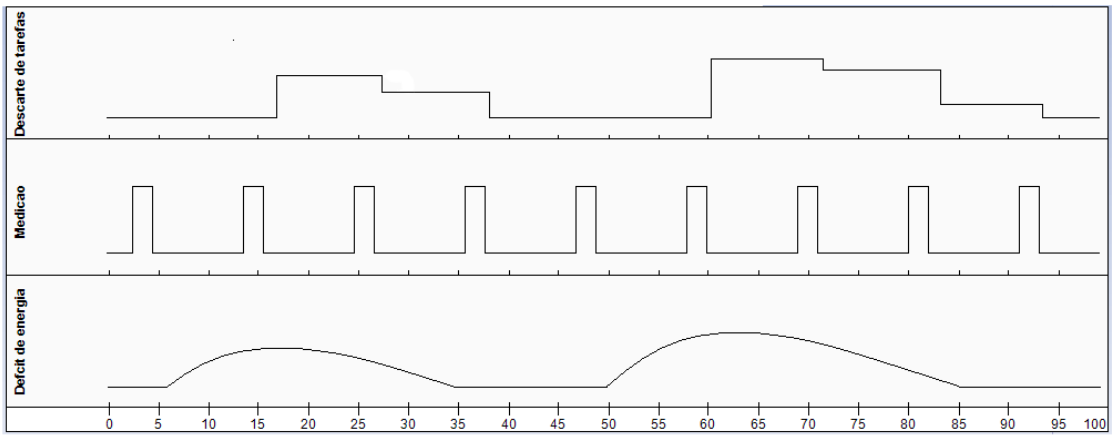

Figura 4: Modelo proposto para o sistema EPOS referente ao descarte de tarefas, e frequência de medições em função do deficit de energia

3. Estudo de casos

Para se testar a abordagem proposta nesse trabalho, foi desenvolvida uma aplicação na área de redes de sensores, semelhante a [Wiedenhoft 2008]. Essa aplicação utiliza a plataforma de sensoriamento *Mica2 mote* [Crossbow 2005], na qual fará uso de dois dispositivos: rádio e sensor de temperatura.

A aplicação consistirá em três *threads*. Duas irão utilizar o sensor para se obter a temperatura do ambiente, das quais uma será *hard real-time* e a outra *best-effort*. A *thread hard real-time* que utiliza o sensor irá realizar uma medição e liberar a CPU. Já a *thread best-effort* de sensoriamento fará 10 medições e irá tirar a média delas, afim de se obter um resultado mais preciso, porém que causa um maior consumo de energia no sistema. Além disso haverá uma terceira *thread hard real-time* que será responsável por utilizar o rádio para enviar à rede de sensores as temperaturas captadas pelas outra *threads*. Os diagramas de sequência abaixo ilustram essas *threads*:

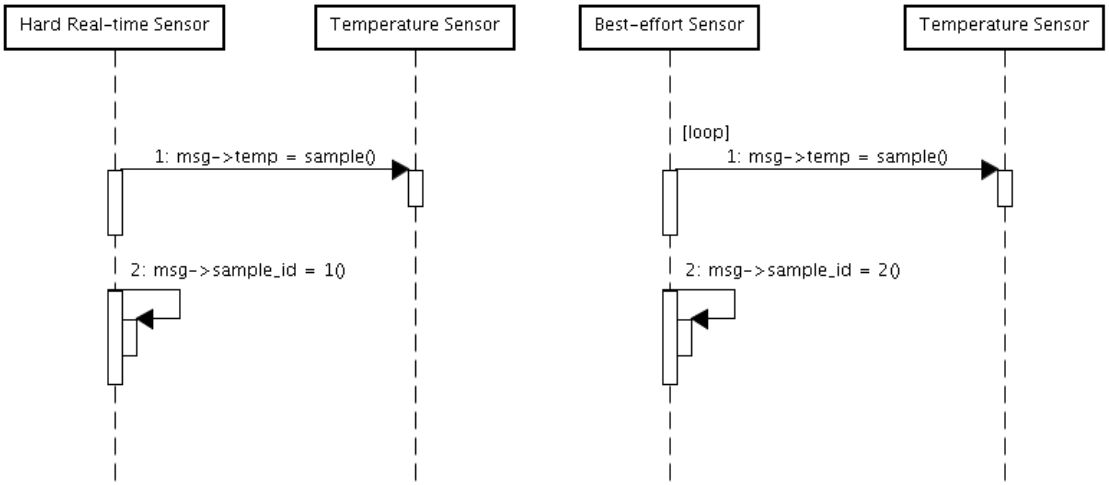

Figura 5: As *threads* da aplicação que utilizam o sensor

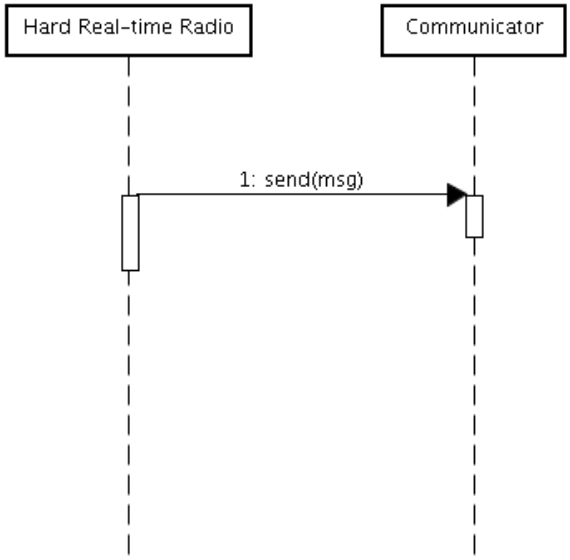

Figura 6: A *thread* da aplicação que utiliza o rádio

O maior inteiro da arquitetura é 65535, portanto as tarefas com prioridade superior a 32767 serão *best-efforts* e as com prioridade inferior serão *hard real-time*. No estudo de casos, a tarefa *best-effort sensor* terá prioridade 62000 (portanto ela só terá seu escalonamento afetado quando o deficit de energia for maior que 3535).

Usando um osciloscópio digital com dois canais foi possível obter a potência consumida pelos motes do estudo de casos. O primeiro canal foi utilizado lendo a tensão das baterias e o segundo lendo a corrente elétrica, conforme pode ser observado na Figura 13. Multiplicando-se a potência observada pelo tempo conseguimos estimar o consumo de energia.

2.2. Impacto da abordagem proposta no consumo de energia do sistema

No primeiro estudo de casos, realizamos medições em um sistema com a abordagem proposta com um deficit de energia baixo artificialmente inserido, um com a abordagem atual do sistema se descartando todas as tarefas opcionais, e um com a abordagem atual escalonando todas as tarefas, cada qual com uma bateria nova. Cada uma das execuções rodou por aproximadamente vinte horas.

Com os resultados obtidos, é possível ver que a energia a mais consumida com o escalonamento da tarefa *best-effort* com deficit de energia baixo, não implica em grandes sobrecustos para a aplicação. No caso da abordagem proposta foi observado um descarte de 56,4 porcento de seu escalonamento, sendo que a energia a mais consumida por esses escalonamentos ficou 30 porcento maior.

A tabela abaixo mostra o consumo de energia com cada uma das configurações desse estudo de casos:

Modelo	Potencia consumida (W)
Modelo atual se descartando todas as tarefas opcionais	0.04784
Modelo proposto com deficit de energia baixo	0.04927
Modelo atual se escalonando todas as tarefas	0.05265

Tabela 1. consumo médio de potência para diferentes abordagens em diferentes contextos

2.2. Percentual de descarte de tarefas best-effort em função do deficit de energia

No segundo estudo de casos, foi deixado a aplicação rodando por três dias e foi armazenada as informações de todos os pacotes enviados pelo *mote* usado. Além das informações sobre a temperatura e o tipo da *thread* que realizou a medição, foi enviado informações sobre o valor do deficit de energia, e a o numero de medições ja feitas.

Com essas informações foi possível calcular o porcentual de tarefas *best-effort* descartadas para diferentes deficits de energia. A Figura 7 mostra isso:

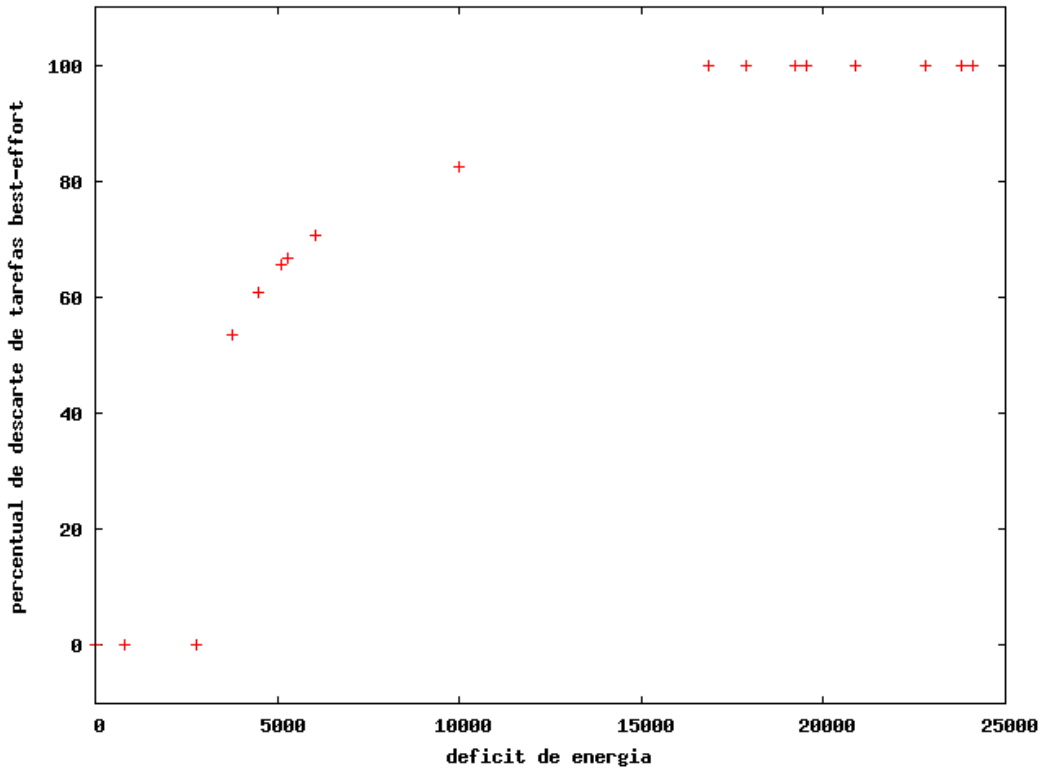

Tabela 1. Percentual de descarte de tarefas em função do deficit de energia

Referencias

Eggermont, L.D.J. “Embedded systems roadmap 2002”. 2002.

Crossbow Technology. “MPR/MIB Mote Hardware User’s Manual”. Crossbow Technology,

Inc, San Jose, CA, September 2005

Fröhlich, A.A.M. “Application Oriented Operating Systems”. Forschungszentrum Informationstechnik, 2001.

Fröhlich, A.A.; Schroder-Preikschat, W. Epos: An object-oriented operating system. Lecture Notes in Computer Science, Springer-Verlag; 1999, p. 27–27, 1999.

Gonçalves, R.T. “Monitoramento da Capacidade de Baterias em Sistemas Embarcados”.

Marwedel, P. “Embedded system design”. [S.l.]: Springer, 2003.

Weiser, M.. Ubiquitous computing IEEE Computer 1999

Wiendehoft, G.R.; Frölich, A.A. “Using Imprecise Computation Techniques for Power

Management in Real-Time Embedded Systems”. 2008.

Wiendehoft, G.R. et al. “Power management in the EPOS system”. ACM New York, NY, USA, 2008.