Gustavo Roberto Nardon Meira

Real-Time Dynamic Voltage and Frequency Scaling no sistema EPOS

31 de outubro de 2011

Gustavo Roberto Nardon Meira

Real-Time Dynamic Voltage and Frequency Scaling no sistema EPOS

Apresentado como requisito à obtenção do grau de Bacharel em Ciência da Computação

Orientador:
Prof. Dr. Antônio Augusto Medeiros Fröhlich

Co-orientador:
Arliones Hoeller Júnior, Me.

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

31 de outubro de 2011

Monografia sob o título “Real-Time Dynamic Voltage and Frequency Scaling no sistema EPOS”, defendida em 4 de outubro de 2011 por Gustavo Roberto Nardon Meira, como parte dos requisitos à obtenção do grau de Bacharel em Ciência da Computação, e aprovada pela seguinte banca examinadora:

Prof. Dr. Antônio Augusto M. Fröhlich
Orientador

Me. Arliones Stevert Hoeller Junior
Co-orientador

Prof. Dr. Rafael Luiz Cancian

Me. Giovani Gracioli

Resumo

Dynamic Voltage and Frequency Scaling (DVFS) é uma técnica de gerenciamento de energia que permite ao sistema operacional decidir quando alterar a tensão e frequência do processador, sendo possível reduzir o desempenho e consumo de energia durante uma baixa demanda computacional. Em sistemas embarcados, além do tempo de vida de bateria, muitas vezes responder em tempo real também se torna uma restrição. Nesse cenário, economizar energia em detrimento do desempenho nem sempre pode ser uma boa ideia. Explorando este problema, o presente trabalho porta o sistema operacional embarcado de tempo real EPOS para a plataforma Intel PXA255, que permite o uso de DVFS. Além disso, um suporte a heurísticas para adequação de desempenho também foi criado. Os experimentos realizados demonstram a economia de energia sem que as tarefas do sistema percam prazos.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

1 Introdução

O emprego de sistemas computacionais embarcados tornou-se muito comum no entretenimento, saúde e conforto. Esta invasão no dia-a-dia das pessoas atribui a estes dispositivos uma gama de características desejáveis. Eles devem ser leves, autônomos e ainda possuir um baixo custo. Citadas apenas três restrições, pode-se notar como o consumo de energia transformou-se em um fator estratégico no projeto destes sistemas. O uso eficiente da energia reflete um maior tempo de vida do dispositivo, o que pode ser trocado, por exemplo, por uma bateria menor, mais leve e mais barata.

O esforço dedicado à redução de energia em sistemas computacionais tem-se concentrado, até hoje, em três pontos principais: **circuitos eletrônicos**, **arquitetura de hardware** e **soluções em software** (RANGANATHAN, 2010).

Para a redução de energia em **circuitos eletrônicos**, o foco científico está principalmente no projeto de elementos digitais básicos eficientes quanto ao uso de energia. Isto significa explorar propriedades elétricas de itens como células de memória, latches ou flip-flops, até a distribuição do sinal de clock entre estes componentes. Quanto a **arquitetura de hardware**, pode-se citar os esforços na criação de unidades funcionais de hardware energeticamente eficientes: particionamento de bancos de memória, caches, estados de baixo consumo e inclusive permitir a alteração da tensão e frequência de funcionamento de uma unidade. Quanto a **soluções em software**, pode-se destacar compiladores que buscam eficiência energética e aplicações ou sistemas operacionais que exploram, principalmente, o modo de funcionamento e a utilização de recursos, como rede, memória e processador.

O foco deste trabalho está situado na **solução em software**, mais precisamente no sistema operacional embarcado. A ideia é que, nas rotinas de escalonamento de tarefas, possa-se explorar uma característica arquitetural possivelmente presente no processador: a alteração de tensão e frequência durante a execução de um programa.

Esta característica é chamada de **DVFS** (*Dynamic Voltage and Frequency Scaling*), ou também **DVS** (*Dynamic Voltage Scaling*). Um escalonador, ciente dessa técnica, pode então identificar momentos nos quais o processador pode funcionar com menor desempenho, com frequência e tensão reduzidas. Assim, o tempo de execução das tarefas que estão sendo executadas aumenta, mas em contrapartida economiza-se energia.

De fato, seria simplificar a realidade utilizar esta técnica em sistemas embarcados e não levar em conta restrições temporais. Muitas vezes estes sistemas são utilizados em **aplicações de tempo real**, ou seja, aplicações que necessitam executar tarefas em tempo hábil. Como exemplo, suponha que um veículo aéreo não tripulado é baseado em um sistema embarcado, onde as tarefas de controle da aeronave e o sistema operacional são executados em um processador que suporta DVFS. Se em um momento de menor carga o sistema operacional decide reduzir a frequência do processador, deve ser garantido que tarefas críticas continuem cumprindo seus prazos. Neste exemplo, as rotinas que devem responder à variação de altitude, ao sofrerem qualquer atraso, podem acarretar consequências catastróficas. Em um cenário como este, a qualidade ou ação de se utilizar DVFS levando em conta as restrições temporais é chamada **RT-DVFS** (*Real Time Dynamic Voltage and Frequency Scaling*), ou simplesmente **RT-DVS** (PILLAI; SHIN, 2001).

1.1 Objetivos

O objetivo geral deste trabalho é fornecer uma implementação real de um sistema operacional embarcado com suporte a RT-DVFS. De maneira mais específica, este trabalho possui dois componentes fundamentais: o primeiro é o porte do EPOS (*Embedded Parallel Operating System*) [1](FRöHLICH, 2001) para uma plataforma de hardware com suporte a DVFS. O segundo componente do trabalho é adaptar o ambiente de tempo real oferecido pelo EPOS, transformando-o em um sistema operacional com suporte a RT-DVFS.

1.1.1 Desenvolvimento do Porte

A plataforma escolhida para desenvolvimento foi o processador Intel PXA255, capaz de realizar DVFS para vários dispositivos internos, como barramento, memória e

seu núcleo Intel XScale. Para os fins deste trabalho, portar o sistema EPOS para tal plataforma de hardware significa:

- Identificar os meios para depuração, programação e inicialização da plataforma de hardware.
- Como o EPOS é desenvolvido em C++, é necessário que seja criado um suporte de baixo-nível para a linguagem, conhecido como *run-time C++*. É este suporte que torna algumas características dinâmicas da linguagem C++ funcionais, essenciais para a inicialização do sistema operacional em questão.
- Identificar e criar os mediadores de hardware necessários. Estes mediadores são os componentes responsáveis por oferecer à camada mais abstrata do sistema uma interface para itens de hardware, isolando as porções de software dependentes de máquina (FRöHLICH, 2001). Vale salientar que o suporte a DVFS resume-se em disponibilizar, na interface de um mediador, comandos que permitam que itens mais abstratos do sistema, como por exemplo um escalonador ou gerente de energia, possam alterar o modo (tensão e frequência) de funcionamento do processador.
- Finalmente, após identificados e implementados os mediadores de interesse, utilizar as aplicações de teste já presentes no EPOS para verificá-los.

1.1.2 Desenvolvimento de Suporte a RT-DVFS

Para o desenvolvimento do suporte a RT-DVFS, busca-se a implementação de heurísticas para tomadas de decisão do sistema. Estas heurísticas a serem implementadas são baseadas no escalonador de tempo real EDF (*Earliest Deadline First*), componente já presente no sistema operacional EPOS. Mais especificadamente, são incorporadas ao sistema duas heurísticas propostas por Pillai e Shin (2001), chamadas *Static DVS* para EDF e *Cycle-conserving DVS* para EDF. A primeira tem caráter estático, decidindo, anteriormente à execução do conjunto de tarefas, qual a configuração de tensão e frequência escolher, sem que as tarefas percam seus prazos. A segunda toma decisões de maneira dinâmica, baseando-se em dados obtidos durante a execução das tarefas. Por final, experimentos devem ser realizados demonstrando o comportamento das heurísticas e sua validade quanto às restrições de tempo real.

1.2 Justificativa

Dispositivos embarcados alimentados por bateria trazem consigo restrições conflitantes: autonomia e desempenho. Ao mesmo tempo que devem economizar energia, ganham a responsabilidade sobre tarefas inteligentes que necessitam de processamento poderoso. Projetar estes dispositivos levando em consideração somente cenários de pior caso pode não ser uma boa ideia. Mesmo existindo aplicações que exijam alto processamento, o pico de computação necessário durante um curto período é maior que a média necessária geralmente em maior parte do tempo de funcionamento destes sistemas, causando assim desperdício de energia (PILLAI; SHIN, 2001). Sistemas embarcados, onde se tem um maior conhecimento sobre o comportamento da aplicação, tornam-se um palco interessante para a exploração do uso de DVFS, de modo que o poder computacional do dispositivo seja utilizado eficientemente.

O estudo de caso utilizado neste trabalho, o EPOS, é um sistema operacional que segue a metologia **ADESD** (*Application-Driven Embedded System Design*) (FRöHLICH, 2001). Uma implementação real de RT-DVFS em um ambiente que segue a metodologia citada é a maior contribuição deste trabalho. Isto se dá, primeiramente, porque grande parte dos estudos em RT-DVFS são realizados através de simulações, que possuem resultados restritos, devido a fatores imprevisíveis ou não modelados (LIN; SONG; CHENG, 2010). Em segundo, observa-se que, dos trabalhos que desenvolvem implementações reais de suporte a RT-DVFS, comumente as demonstrações ocorrem em sistemas Linux com a adição de módulos ou extensões (PILLAI; SHIN, 2001; SNOWDOWN; RUOCCO; HEISER, 2005; ZHU; MUELLER, 2007; LIN; SONG; CHENG, 2010), enquanto o EPOS já possui suporte nativo a tarefas de tempo real (MARCONDES et al., 2009).

Com a grande demanda de dispositivos que exigem baixo consumo de energia, o suporte a DVFS está presente em muitos processadores conhecidos (CHEN; KUO, 2007). Por estar sendo portado para uma plataforma Intel/Marvell XScale, o EPOS se beneficia duplamente. Primeiro pelo fato de estar disponível para um processador dedicado a aplicações de baixo consumo. Em segundo, há também o benefício experimental e comparativo, já que esta mesma plataforma está presente em trabalhos com implementações reais de sistemas RT-DVFS, onde a heurística *Cycle-conserving* para EDF, utilizada por este trabalho, é usualmente tomada como referência (KUMAR; MANIMARAN, 2005; CHEN; KUO, 2007; TSAI; WANG; CHEN, 2007; ZHU; MUELLER, 2007).

2 Fundamentação

Este capítulo apresenta os fundamentos necessários para o entendimento do desenvolvimento do trabalho. As duas primeiras seções abordam os conceitos de tempo real e DVFS utilizados. A terceira seção aborda estes dois primeiros conceitos somados, apresentando então a ideia de DVFS em tempo real. A quarta e última seção é uma breve explanação sobre o EPOS, o sistema operacional usado como estudo de caso.

2.1 Sistema de Tempo Real

Esta seção apresenta o modelo de sistema de tempo real que será utilizado neste trabalho. Ele é baseado diretamente no modelo que Pillai e Shin (2001) utilizam para propor as heurísticas escolhidas para implementação. Neste cenário, um sistema de tempo real é um conjunto de tarefas periódicas com prazos definidos, todas concorrendo por um único processador. Ainda, para os efeitos deste trabalho, as tarefas apresentadas são passíveis de preempção, ou seja, a qualquer momento podem ser interrompidas para que cedam o uso processador.

2.1.1 Modelo de Sistema de Tempo Real

No modelo em questão, o sistema é composto por um conjunto de tarefas periódicas que não compartilham nenhum recurso entre si. Assim sendo, a toda tarefa T_i, tem-se atribuído um *período* P_i, de modo que a tarefa T_i sempre é *requisitada* para execução a cada P_i unidades de tempo. Observe que mesmo sendo requisitada no início de cada período, a tarefa pode iniciar sua execução efetiva a qualquer momento dentro do período, devido, por exemplo, a uma outra tarefa de maior prioridade estar ocupando o processador.

Denomina-se *tempo de resposta* o tempo entre a requisição e a conclusão da tarefa. Neste trabalho, assume-se que para efeitos de estabilidade do sistema, o tempo de resposta de uma tarefa nunca deve exceder seu respectivo período. Neste sentido, se uma tarefa T_i começa a ser executada entre os tempos t e $t+P_i$, sua conclusão não deve exceder $t+P_i$, sendo este limite também chamado de *deadline*.

Como pode-se esperar em um sistema comum, cada execução de T_i pode possuir um tempo de resposta diferente. Para que o escalonador de tarefas de tempo real seja capaz de ordenar múltiplas tarefas de modo que estas não ultrapassem seus deadlines, este modelo também utiliza o *pior tempo de computação* C_i associado à tarefa T_i, ou seja, toda tarefa que é iniciada em t, caso ela nunca deixe o processador, sempre tem sua conclusão antes ou exatamente no momento $t+C_i$. Finalmente, a figura 2.1 exemplifica este modelo com um sistema composto por duas tarefas T_1 e T_2 sendo executadas.

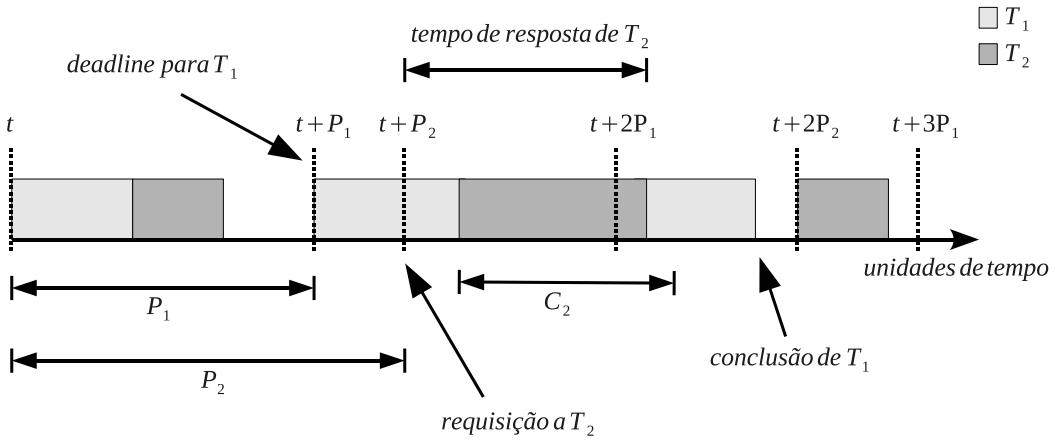

Figura 2.1: Exemplo do modelo de sistema de tempo real utilizado

Dado o modelo apresentado, é possível definir a *utilização* U_i de uma tarefa T_i em um único processador como sendo $U_i = C_i/P_i$. Diz-se que a *utilização total* por um conjunto de n tarefas T_i é, então,

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i}.$$

Pode-se supor que a troca de contexto entre tarefas de uma aplicação dá-se através de uma rotina em software. Obviamente esta rotina também ocupa tempo de processamento no sistema, mas esta faixa de tempo pode ser negligenciada no modelo, pois é possível tratá-la como parte do pior tempo de computação das tarefas em execução.

2.1.2 Escalonador de Tempo-Real

Um sistema operacional de tempo real, também chamado de RTOS (*Real-Time Operating System*), é o programa de computador que, além de atender a outras responsabilidades, ordena a execução de tarefas da aplicação de modo a responder em tempo hábil alguma requisição. Mais detalhadamente, o componente do RTOS responsável por esta ordenação é o *escalonador de tempo real* (LI; YAO, 2003; PILLAI; SHIN, 2001). É este componente que deve prover um ou mais algoritmos que resolvem as prioridades de execução das tarefas do sistema, a fim de que nenhuma delas perca seu deadline.

Para que o escalonador de tempo real seja capaz de garantir a execução em tempo hábil de um conjunto de tarefas, duas condições são estabelecidas:

- O conjunto de tarefas deve ser *escalonável*. Isto significa que o conjunto de tarefas deve passar por um ou mais testes propostos pelo algoritmo do escalonador, justamente para que este conjunto satisfaça as premissas de modo que o algoritmo mantenha suas propriedades. Estes testes são chamados de *testes de escalonabilidade*.
- Nenhuma das tarefas excede seu pior tempo de computação (como elaborado na seção 2.1.1).

Ainda quanto à ordenação realizada pelo escalonador, esta pode ser entendida como a atribuição de uma prioridade a cada tarefa. Essa atribuição pode ser realizada durante a execução ou previamente. Quando as prioridades são decididas durante execução, diz-se que o escalonador ou o escalonamento é *dinâmico*. No caso em que as prioridades são atribuídas anteriormente à execução, diz-se que o escalonador ou o escalonamento é *estático*.

Quando um escalonador pode interromper a execução de uma tarefa para que uma de maior prioridade ganhe o processador, diz-se que este é um *escalonador preemptivo*.

Dentre os algoritmos para escalonadores de tempo real existentes, este trabalho utiliza o *Earliest Deadline First*, ou simplesmente EDF. Usualmente, diz-se então que o escalonamento ou escalonador utilizado é EDF, e é nele que a heurística *Cycle-conserving RT-DVS* para EDF se baseia. O mecanismo de ordenação do algoritmo EDF é explicado na subseção seguinte (2.1.3).

2.1.3 Escalonador *Earliest Deadline First*

O algoritmo *Earliest Deadline First* é um algoritmo para escalonamento dinâmico. Seu funcionamento dá-se com a atribuição de prioridades às tarefas, de modo que quanto mais próxima do deadline a tarefa está, maior é sua prioridade (LIU; LAYLAND, 1973). A figura 2.2 demonstra um exemplo de escalonamento EDF para duas tarefas, onde cada d_i representa deadlines para a tarefa T_i. Ambas tarefas tem sua primeira requisição em t. Observe que no primeiro deadline d_1 acontece a preempção de T_2, pois neste momento há uma nova requisição de T_1, que por sua vez apresenta um deadline mais próximo que T_2.

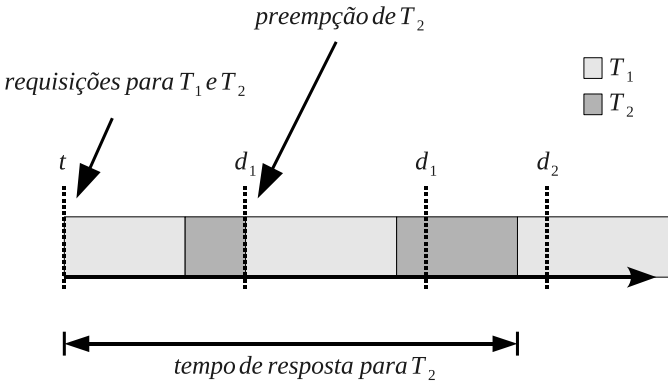

Figura 2.2: Exemplo de escalonamento EDF para duas tarefas

Um teste de escalonabilidade para o EDF é o seguinte:

Dado um conjunto T de n tarefas T_i, para $1 \leq i \leq n$, tais que:

- sejam independentes entre si;
- sejam passíveis de preempção;
- sejam periódicas;
- possuam deadlines iguais a seus respectivos períodos.

Então, T é escalonável por um escalonador EDF se, e somente se

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1.$$

O teste descrito nada mais é do que a verificação da utilização total do processador para conjunto de tarefas T. Este teste, por si só, já é necessário e suficiente para que o conjunto de tarefas seja escalonável.

2.2 *Dynamic Voltage and Frequency Scaling*

Dynamic Voltage and Frequency Scaling (DVFS), ou também *Dynamic Voltage Scaling* (DVS), é a capacidade ou atitude de um processador alterar sua tensão e frequência durante a execução de um programa, ou seja, alterar sua tensão e frequência dinamicamente (PILLAI; SHIN, 2001).

A ideia desta técnica é oferecer um contrato entre desempenho de processamento e consumo de energia. A fundamentação desta troca se estabelece sobre duas importantes características da tecnologia de sistemas computacionais atual:

- Primeiramente, quase a totalidade destes sistemas são baseados em lógica CMOS.
- Em segundo, o pico de computação das aplicações, sejam elas embarcadas ou não, é usado apenas durante uma pequena fração do tempo de funcionamento desses sistemas.

A primeira característica apresentada influi no contrato, pois, pelas propriedades físicas dos circuitos CMOS atuais, a energia dissipada (P) é fortemente relacionada à frequência (f) e à tensão de funcionamento (V) do sistema, dois fatores fundamentais para o desempenho (SNOWDOWN; RUOCCO; HEISER, 2005). Esta proporção é visível da seguinte forma:

$$P \propto fV^2.$$

Levando em conta que o tempo de computação é inversamente proporcional a f, pode-se dizer que a energia E, utilizada na computação de uma tarefa em um processador CMOS, é estabelecida conforme a seguinte proporção:

$$E \propto V^2.$$

Detalhando esta primeira característica, diminuindo a frequência para a redução de consumo de energia, o tempo para o término da dada tarefa aumenta, ou seja, mesmo com um consumo energético menor, este consumo será mantido por mais tempo. Por outro lado, diminuindo-se a frequência, aumenta-se o período entre os chaveamentos do circuito digital, tornando o efeito temporal da capacitância mais tolerável, daí a possibilidade de redução da tensão. Mesmo que o tempo de execução da tarefa seja linearmente proporcional à frequência, a energia gasta é quadraticamente

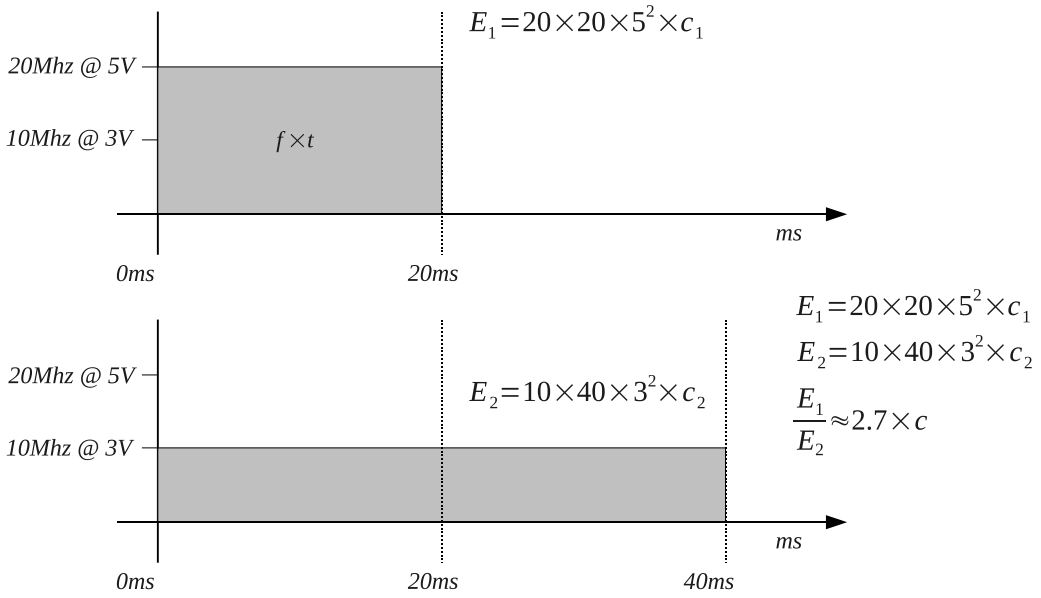

Figura 2.3: Exemplo de ganho energético obtido através de DVFS

proporcional à tensão, permitindo um ganho energético. Na figura 2.3 um exemplo é ilustrado para uma tarefa que possui tempo de computação exato de 20 milisegundos a 20MHz. Mesmo reduzindo-se pela metade a frequência do processador, tomando-se o dobro de tempo para a execução da tarefa exemplo, ainda há redução no consumo de energia.

Finalmente, tendo em foco a segunda característica, que diz respeito ao pico e a média de computação necessária em um sistema computacional, pode-se então, ajustando-se a frequência às necessidades da aplicação, consumir a energia necessária para alto desempenho somente nos momentos realmente requisitados pela aplicação.

Os processadores que implementam esta tecnologia oferecem uma interface que permite, ao software em execução, a alteração dinâmica da tensão. Essa interface, geralmente, é disponível na forma de registradores ou instruções especiais. Um exemplo seria a implementação Intel SpeedStep para núcleos ARM. Neste caso, expande-se a arquitetura ARM do processador através de um co-processador, o qual possui registradores acessíveis através do conjunto de instruções ARM padrão. É possível então, através destes registradores, configurar dinamicamente modos de operação não somente para o núcleo (CPU), mas também para o barramento principal e memória. Cada um destes modos de operação estabelece uma tensão e sua respectiva frequência de funcionamento para estes dispositivos (SNOWDOWN; RUOCCO; HEI-

SER, 2005).

2.3 *Real-Time Dynamic Voltage and Frequency Scaling*

Real-Time Dynamic Voltage (and Frequency) Scaling, ou simplesmente RT-DVS, é a qualidade que Pillai e Shin (2001) atribuíram aos algoritmos que propuseram. Em um sentido mais amplo, pode-se dizer que RT-DVS, ou RT-DVFS, é capacidade de se alterar a tensão e frequência do processador, para fins de economia de energia, de modo que as tarefas do sistema continuem respondendo em tempo hábil.

Já que nem sempre as tarefas de uma aplicação de tempo real acabam executando como seu pior caso (C_i), e a utilização do processador nem sempre é total, é possível usufruir da economia energética oferecida pelas técnicas de DVFS em sistemas de tempo real. Por exemplo, se uma tarefa termina sua execução mais cedo que o esperado, como mostrado na figura 2.4, pode-se então utilizar a faixa de tempo não utilizada para que uma próxima tarefa seja executada, mas agora com menor desempenho. Esta técnica também é conhecida como *slack reclamation* (CHEN; KUO, 2007). Ainda assim, esse tipo de decisão deve ser tomada com cautela, de modo que nenhum prazo seja perdido.

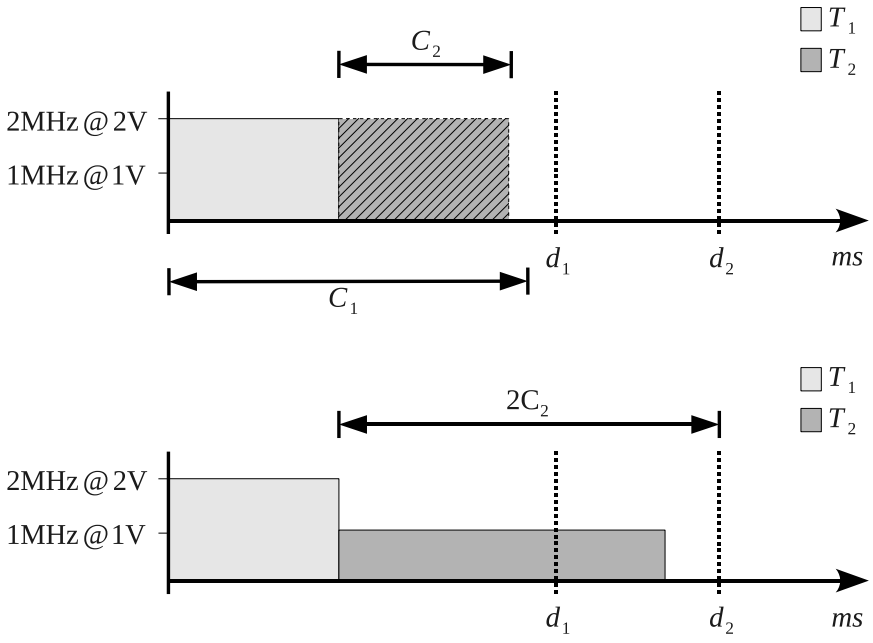

Figura 2.4: Exemplo de utilização de DVFS em sistemas de tempo real

2.3.1 Escalonamento em Sistemas RT-DVFS

As primeiras prototipagens de sistemas operacionais embarcados RT-DVFS foram apresentas por Pillai e Shin (2001). Em seu trabalho, foram desenvolvidas heurísticas que, fracamente acopladas ao escalonador do sistema operacional, oferecem consciência energética aos algoritmos *Rate-Monotonic* (LIU; LAYLAND, 1973) e EDF. Pillai e Shin (2001) propõem três classes de heurísticas: as de alteração de tensão estáticas (*Static Voltage Scaling*), as de conservação de ciclos (*Cycle-conserving RT-DVS*) e as de previsão (*Look-ahead RT-DVS*). Das classes citadas, este trabalho utiliza em sua implementação as duas primeiras em conjunto ao algoritmo de escalonamento EDF. Esta escolha é justificada pela simplicidade de implementação e, ao mesmo tempo, por permitir a comparação entre heurísticas que tomam decisões estaticamente e heurísticas que tomam decisões durante a execução do sistema. O funcionamento das duas heurísticas escolhidas é explicado nas seções seguintes.

2.3.2 *Static Voltage Scaling* para EDF

A primeira abordagem de Pillai e Shin (2001) é um mecanismo simples que se beneficia da alteração de tensão e frequência, mantendo a execução das tarefas dentro de seus respectivos prazos. Como o próprio nome da técnica sugere, esta é uma configuração estática da frequência do processador (não se deve confundir com escalonamento estático, como apresentado na seção 2.1.2). A ideia é obter a menor frequência possível de modo que, mesmo neste baixo desempenho, o conjunto de tarefas satisfaça o teste de escalonabilidade do escalonador EDF. A frequência, por ser configurada estaticamente, só é alterada se o conjunto de tarefas é alterado. Assim sendo, se durante a execução, uma nova tarefa é adicionada ao sistema, o conjunto de tarefas está sendo alterado, então é necessária novamente a escolha de uma nova frequência de funcionamento.

É possível perceber que, alterando a frequência de operação por um fator α $(0 < \alpha \leq 1)$, será necessário alterar os piores tempos de computação para cada tarefa em um fator $1/\alpha$. Observe que os períodos e deadlines, seguindo o modelo apresentando na seção 2.1.1, mantêm-se inalterados. Com um novo pior tempo de computação, haverá alteração no teste de escalonamento.

Para verificar a escalonabilidade dos conjuntos de tarefas, utiliza-se o teste apresentado na seção 2.1.3. Este teste, agora com a frequência alterada pelo fator α,

ficaria como o seguinte[1]:

Dado um conjunto T de n tarefas T_i, para $1 \leq i \leq n$, tais que:

- sejam independentes entre si;
- sejam passíveis de preempção;
- sejam periódicas;
- possuam deadlines iguais a seus respectivos períodos.

T é escalonável por um escalonador EDF se, e somente se

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \alpha.$$

A consciência energética da heurística *Static Voltage Scaling* para EDF consiste então na otimização do conjunto das m frequências possíveis para o processador, sendo escolhida a menor delas capaz de manter o conjunto de tarefas escalonável segundo o teste apresentado. Assim, escreve-se o procedimento *seleciona-frequência*, como expresso no algoritmo 1.

```
teste-EDF(α):
    se Σ_{i=1}^{n} C_i/P_i ≤ α então
        retorna verdadeiro
    senão
        retorna falso
    fim

seleciona-frequência:
    use a menor frequência f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
    tal que teste-EDF(f_i/f_m) retorne verdadeiro.
```

Algoritmo 1: Alteração estática de tensão para EDF

Se o conjunto de tarefas passa no teste de escalonabilidade para a nova frequência e, ainda, as tarefas também não ultrapassam seu novo pior caso de resposta, este mecanismo assegura que os deadlines não serão comprometidos. Se o teste de escalonabilidade falha para todas configurações possíveis, então não é possível escalonar o conjunto de tarefas.

[1] O novo teste de escalonabilidade nada mais é do que a aplicação do fator $\frac{1}{\alpha}$ a cada C_i na antiga inequação. Multiplicando os dois lados da inequação por α, obtém-se a expressão apresentada aqui.

Pela seleção da frequência ser realizada anteriormente à execução do conjunto de tarefas, o novo algoritmo torna-se fracamente acoplado ao escalonador de tempo real (PILLAI; SHIN, 2001), ou seja, uma prévia configuração do sistema pode ser feita, então um escalonador EDF comum pode realizar seu trabalho normalmente, sem necessidade desta entidade entrar em contato com a heurística. Por outro lado, este mecanismo não oferece economia de energia nos momentos onde uma tarefa não atinge seu pior tempo de computação, o que pode acontecer em sistemas reais. Para este caso, algoritmos que levam em conta a utilização real da tarefa, como os expostos adiante, conseguem um melhor benefício.

Um exemplo, apresentado na figura 2.5, pode ser realizado com as tarefas mostradas na tabela 2.1. Observe que o pior tempo de resposta (C_i) está avaliado para a máxima frequência de funcionamento do processador, ou seja, o caso onde $\alpha = 1$ (cenário *a* na imagem). Em um conjunto de três configurações possíveis para frequência, a correspondente a $\alpha = 0,75$ é a menor delas que ainda mantém o conjunto de tarefas escalonável (cenário *b* na imagem). Para $0,75 > \alpha > 0$, algumas tarefas podem perder seus prazos de execução, como é o caso no cenário *c* apresentado na figura, onde a tarefa T_2 perde seu deadline.

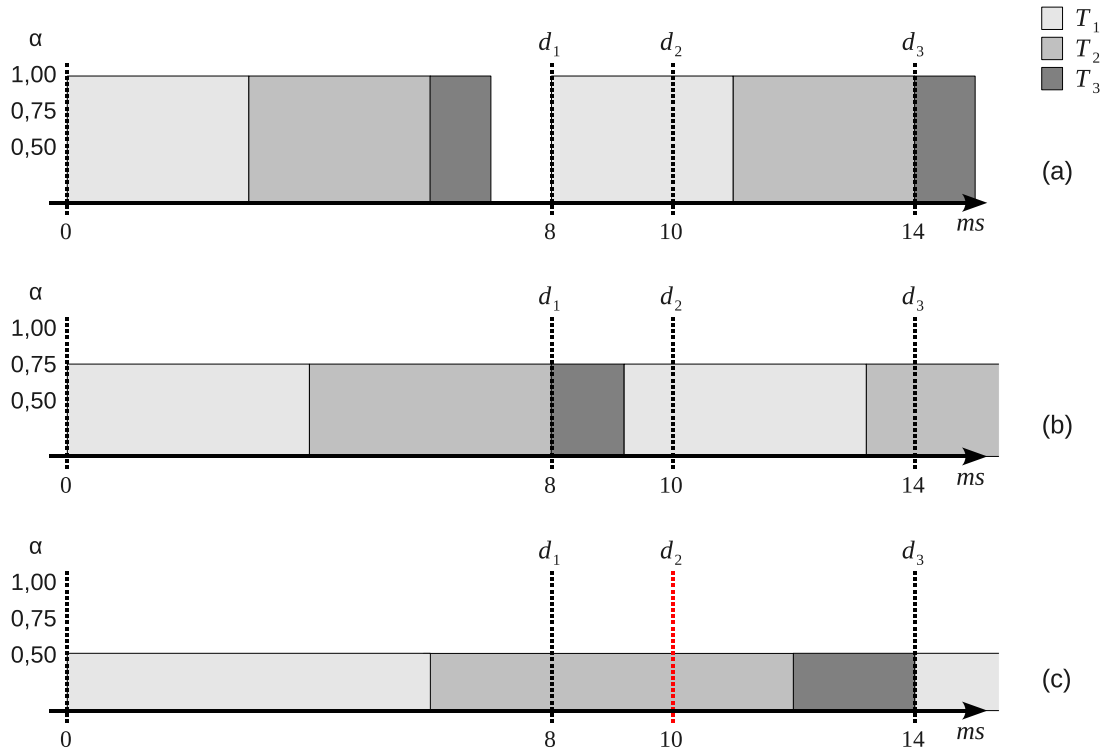

Figura 2.5: Exemplo de escalonamento EDF com alteração estática de frequência

Tarefa (T_i)	Pior Tempo de Computação (C_i)	Período (P_i)	Utilização da tarefa (U_i)
T_1	$3ms$	$8ms$	$0,375$
T_2	$3ms$	$10ms$	$0,300$
T_3	$1ms$	$14ms$	$0,071$

Tabela 2.1: Exemplo de conjunto de tarefas

2.3.3 *Cycle-conserving RT-DVS* para EDF

Uma maneira de tirar maior proveito do uso de DVFS durante o escalonamento EDF é aproveitar as folgas de tempo obtidas quando tarefas não atingem seu pior tempo de computação. De qualquer modo, quando uma tarefa é requisitada para sua execução, não é possível prever o tempo de computação real que será utilizado. Uma solução, proposta por Pillai e Shin (2001), está em assumir que a tarefa inicialmente ocupa seu pior tempo de resposta, mas ao seu término, verificar qual foi o real uso do processador na execução. A ideia desta heurística é então, utilizando estes dados capturados durante a execução das tarefas, adequar o desempenho do processador próximo à utilização real que o conjunto de tarefas promove.

Na heurística *Cycle-conserving RT-DVS* para EDF, três eventos do sistema devem ser capturados: a alteração do conjunto de tarefas, o início da execução de uma tarefa e a conclusão de uma tarefa. A estes eventos, ações são atribuídas:

Na alteração do conjunto de tarefas, é possível comportar-se como na heurística estática apresentada anteriormente, ou seja, aplicar o algoritmo 1 apresentado anteriormente. Isto garante configuração inicial válida para o sistema, sem nenhum prejuízo, já que a heurística *Static Voltage Scaling* é baseada em um cenário de pior caso.

No início da execução da tarefa T_i, calcula-se sua utilização no pior caso de computação. Ou seja, assume-se $U_i = C_i/P_i$ até a conclusão da tarefa. Por final, calcula-se a utilização total e aplica-se o algoritmo 1 para a seleção da configuração desejada.

Na conclusão da tarefa T_i, calcula-se também sua utilização, mas baseando-se agora em seu tempo de computação real cc_i. Ou seja, assume-se $U_i = cc_i/P_i$ até que haja uma nova requisição para a tarefa. Novamente, com a utilização calculada, aplica-se o algoritmo 1 para a seleção da configuração desejada.

A figura 2.6 demonstra a heurística *Cycle-conserving RT-DVS* para EDF sendo aplicada. O conjunto de tarefas da tabela 2.1 é utilizado como exemplo. A tabela 2.2 mostra o tempo de resposta destas tarefas com $\alpha = 1$, para a primeira e segunda invocação de cada tarefa. As setas apontam a utilização total calculada no início da execução da tarefa (utilizando-se dados estáticos), e também a utilização total calculada com os dados obtidos *online*, ao final da execução das tarefas.

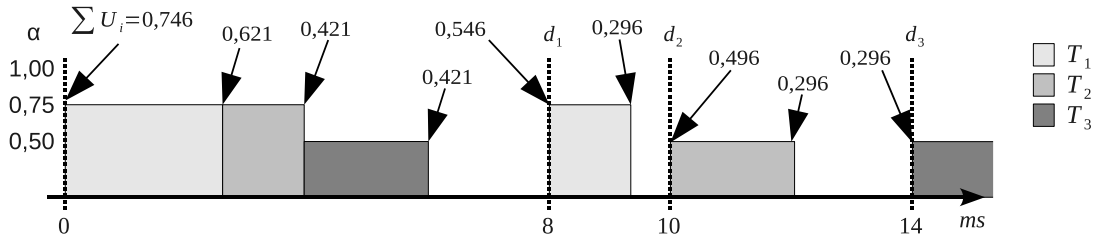

Figura 2.6: Exemplo de escalonamento *Cycle-conserving RT-DVS* para EDF com a utilização calculada em cada início e término de tarefa

Tarefa	Tempo de Computação na Primeira Invocação	Tempo de Computação na Segunda Invocação
T_1	$2ms$	$1ms$
T_2	$1ms$	$1ms$
T_3	$1ms$	$1ms$

Tabela 2.2: Tempo de computação para cada invocação do exemplo

O algoritmo 2 representa o comportamento da heurística *Cycle-conserving RT-DVS* para EDF. Os procedimentos *tarefa-requisitada* e *tarefa-concluída* são executados em seus respectivos eventos para uma dada tarefa T_i. A cada alteração do conjunto de tarefas, *seleciona-frequência* (algoritmo 1) é chamado. Observe que este algoritmo necessita do dado cc_i, que é o tempo de computação real utilizado pela tarefa T_i. Este é um dado que deve ser calculado em tempo de execução pelo sistema operacional, e além disso, disponibilizado na interface de programação oferecida para

a implementação das heurísticas.

```
seleciona-frequência:
    use a menor frequência f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
    tal que U_1 + ··· + U_n ≤ f_i/f_m.

tarefa-requisitada(T_i):
    U_i ← C_i/P_i
    seleciona-frequência

tarefa-concluída(T_i):
    U_i ← cc_i/P_i
    seleciona-frequência
```

Algoritmo 2: *Cycle-conserving RT-DVS* para escalonador EDF

2.4 EPOS e Portabilidade

O EPOS (*Embedded Parallel Operating System*) (FRöHLICH, 2001) é um sistema operacional direcionado a aplicações embarcadas de alto desempenho. Segue a metodologia ADESD (*Application-Driven Embedded System Design*), utilizando técnicas de orientação a aspectos e programação estática em busca de se adequar às restrições presentes nas aplicações para o qual é utilizado.

Ao mesmo tempo em que o EPOS busca especialização para a aplicação, deve lidar com a gama de plataformas utilizadas para a produção e durante o ciclo de vida das aplicações. O sistema operacional hoje conta com portes para diversas plataformas, como IA32, ARM7, AVR8, MIPS e PowerPC. Seguindo a metodologia ADESD, ou seja, sendo um sistema orientado a aplicação, a troca da plataforma a ser utilizada na aplicação deve permanecer transparente ao programador do sistema.

O EPOS busca balanço entre desempenho e portabilidade através de dois artefatos: primeiro, rotinas de pré-configuração, presentes no sistema como a entidade *Setup Utility*; em segundo, os mediadores de hardware.

2.4.1 *Setup Utility*

As rotinas de pré-configuração do sistema são porções de código dependentes de hardware. Elas são executadas anteriormente ao ambiente do sistema operacional. Seu papel é construir o contexto fundamental para a execução do EPOS. Isto significa

oferecer um cenário seguro de execução e inicialização do sistema, garantido, por exemplo, a existência de uma pilha consistente ou o desligamento de interrupções. Estas rotinas facilitam a portabilidade devido ao fato de reduzirem a complexidade de outros componentes do sistema operacional.

Ao final da execução do componente *Setup Utility*, o controle do sistema é entregue ao suporte dinâmico (*run-time*) da linguagem C++ (linguagem na qual maior parte do sistema é escrita), que desencadeia a inicialização dos outros componentes do EPOS.

2.4.2 Mediadores de Hardware e Abstrações

Os mediadores de hardware do EPOS oferecem interfaces simples para acesso a itens dependentes de máquina. Assim, componentes mais abstratos do sistema (chamados de *abstrações*), como por exemplo, *Thread* ou *Scheduler*, podem permanecer com suas implementações portáveis independentemente da plataforma alvo.

3 Desenvolvimento

Este capítulo apresenta o desenvolvimento deste trabalho em duas seções. A primeira descreve o desenvolvimento do porte do sistema operacional para a plataforma alvo. A segunda seção trata da criação do suporte a RT-DVFS sobre o sistema operacional EPOS.

3.1 Porte do Sistema Operacional

Esta seção apresenta a avaliação de uso da plataforma Gumstix Connex (GUMSTIX, 2011b), que possui um processador Intel PXA255 (INTEL, 2004), com a arquitetura Intel XScale, vastamente utilizada em estudos de energia na última década (CHEN; KUO, 2007). Além disso, também é descrita a estruturação do sistema EPOS necessária para o desenvolvimento do porte para a plataforma de hardware citada.

3.1.1 Plataforma Alvo

A plataforma Connex faz parte de uma linha descontinuada de placa-mães da companhia Gumstix. Esta placa possui um processador Intel PXA255 com microarquitetura Intel XScale. O PXA255 mostrou-se interessante para os fins deste trabalho, pois o núcleo Intel XScale é justamente direcionado para aplicações embarcadas que exijam alta performance e baixo consumo de energia (INTEL, 2004).

O processador PXA255 suporta DVFS para três componentes do processador: a CPU, o barramento principal e memória (SNOWDOWN; RUOCCO; HEISER, 2005), fator fundamental para o desenvolvimento deste trabalho. Além disso:

- O laboratório onde este trabalho foi desenvolvido possui um conjunto de placas Connex e outros módulos de expansão.

- Mesmo com o processador PXA255 oferecendo uma interface JTAG (IEEE, 2010) para depuração, as placas da linha Connex não oferecem acesso necessário a este recurso. Por outro lado, estas placas Gummstix podem ser emuladas através do QEMU (QEMU PROJECT, 2011). Felizmente, o QEMU possui opções de depuração junto ao GDB (GNU, 2011), de modo que seja possível a supervisão da execução de instruções passo a passo na plataforma.

- Processadores compatíveis com o PXA255 e sua arquitetura já estão bem estabelecidos na comunidade de sistemas embarcados e RT-DVFS. Muitos trabalhos exibem experimentos utilizando a plataforma (CHEN; KUO, 2007). Mesmo que a linha Connex tenha sido descontinuada no ano de 2009 pela Gumstix, ainda assim, versões recentes de placa-mães do mesmo fabricante utilizam arquiteturas compatíveis com a Intel XScale (GUMSTIX, 2011a). Assim sendo, além de um cenário de estudo presente em outras publicações, o projeto EPOS também se beneficia, já que o sistema será portado para uma plataforma que é amplamente utilizada em aplicações embarcadas que necessitam de baixo consumo de energia e alto desempenho.

- A arquitetura Intel XScale utiliza o conjunto de instruções ARMv5TE, mantendo compatibilidade binária com o conjunto ARMv4T (ARM, 2005). Felizmente o EPOS já possui porte para plataformas ARM7, sendo que estas seguem a arquitetura ARMv4T. Assim, é possível herdar parte considerável do suporte de baixo nível necessário ao EPOS e também reaproveitar a implementação de alguns componentes, como o mediador de CPU, responsável, por exemplo, pelas rotinas de troca de contexto.

A Gumstix Connex foi escolhida como plataforma de implementação deste trabalho, não somente por oferecer o suporte a DVFS, mas também pelos fatores facilitadores de implementação, além de benefícios experimentais ao EPOS.

3.1.2 Programação da Plataforma

Os primeiros passos tomados para portar o sistema operacional EPOS foram na direção da programação da plataforma Connex. Ela possui 64MB de RAM e uma memória *flash* programável de 128kB, onde o processador busca, em seu endereço base, a primeira instrução para inicialização do sistema.

De qualquer modo, esta memória programável já vem, desde a fabricação da plataforma, tomada pelo U-Boot, um *bootloader* que facilita a inicialização de sistemas operacionais na plataforma. É necessário citar que o U-Boot possui rotinas que auxiliam na programação do sistema, como escrita em memória RAM ou própria *flash*, sendo possível até carregar imagens executáveis utilizando o protocolo Kermit, isto através da porta serial disponível em uma das placas de expansão da plataforma Connex.

Como a placa Connex não oferece acesso à interface JTAG oferecida pelo PXA255, a qual facilitaria a escrita da *flash*, a melhor alternativa para programação da plataforma foi a utilização do próprio U-Boot. A partir da imagem de memória resultante da compilação do EPOS, que contém o sistema operacional, pode-se criar uma outra compatível com a rotina inicializadora de imagens do *bootloader*. Para tanto, usa-se a ferramenta *mkimage*. Criando a imagem EPOS compatível com o U-Boot, basta carregá-la em memória, utilizando a porta serial através do protocolo Kermit implemetado pelo U-Boot.

3.1.3 Depuração

Confirmado o método de programação da plataforma, procurou-se então testar as facilidades de depuração oferecidas pelo QEMU versão *0.12.5*.

O QEMU necessita de que uma imagem da memória programável seja especificada, logicamente para que a plataforma emulada seja inicializada conforme os moldes da seção anterior (3.1.2). Para que a emulação seja a mais fiel possível à plataforma disponível no laboratório, esta imagem foi criada com a versão *1.2.0* do U-Boot, a mesma presente na placa Connex utilizada para desenvolvimento.

A facilidade ofertada pelo QEMU é colocar-se como monitor remoto do GDB. Assim, é possível conectar o GDB (utilizando TCP) ao QEMU, e supervisionar a execução passo a passo de instruções na plataforma emulada. Até a programação pode ser feita através do GDB, escrevendo a imagem EPOS inicializável na memória emulada, e então, delegando ao U-Boot a inicialização do EPOS.

Com os processos de programação da plataforma e depuração, o suporte técnico necessário para construção do novo porte é concluída.

3.1.4 *Setup Utility* e *Run-time C++*

Para os fins deste projeto, o EPOS é configurado para uso em modo *library*. Isto significa que ele é ligado à aplicação como uma biblioteca estática. Sendo assim, por exemplo, não há a necessidade do uso de chamadas de sistema (usualmente nomeadas *System Calls*) (FRöHLICH, 2001).

Para a inicialização do sistema operacional, basta indicar ao U-Boot o endereço da primeira instrução contida na imagem do EPOS. Em seu fluxo normal de inicialização, fluxo *a* na figura 3.1, *Setup Utility* é o primeiro componente do sistema a entrar em ação. No presente trabalho, pelos motivos apontados a seguir, elimina-se o componente *Setup Utility*, seguindo o fluxo *b* apresentado na figura 3.1.

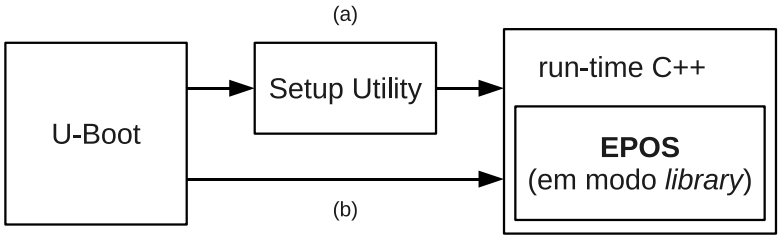

Figura 3.1: Sequência de inicialização do EPOS

Observe que a mesma abordagem é utilizada em outros portes, como por exemplo os para máquinas com arquitetura AVR8, que possuem também uma baixa complexidade relativa.

Remoção do *Setup Utility*

Como explicado anteriormente em 2.4.1, o EPOS possui uma rotina de pré-configuração do hardware. O objetivo dessa rotina é reduzir a complexidade de alguns componentes mais abstratos do sistema operacional. Se comparada a outras plataformas para quais o EPOS foi portado, como IA32 ou PowerPC, o processador PXA255 possui uma complexidade relativamente reduzida, justificando a não necessidade da criação do componente *Setup Utility* para este porte.

As atividades de pré-configurações reconhecidas como necessárias, para a execução segura do EPOS, foram: o estabelecimento de uma pilha única consistente e o remapeamento dos vetores de interrupção do processador. Para reduzir o tempo de desenvolvimento, estas duas atividades foram delegadas, respectivamente, ao suporte dinâmico da linguagem C++ e às rotinas de inicialização dos mediadores de

hardware. O funcionamento destes mecanismos são explicados em subseções seguintes.

Run-time C++

O *run-time* C++ necessário é responsável por um fator essencial do sistema: inicialização de componentes do sistema, como mediadores ou entidades mais abstratas.

Algumas instâncias de componentes do sistema são tratados pela linguagem como objetos de escopo global, isto faz com que as suas inicializações sejam dependentes da implementação dos mecanismos dinâmicos da linguagem C++ (STROUSTRUP, 1997). Em outras palavras, isto significa que deve haver a implementação em software do suporte para que estes componentes tenham suas rotinas de inicialização, que foram geradas pelo compilador C++, chamadas antes da execução da função inicial da aplicação.

Este suporte de baixo-nível deve ser escrito em linguagem *assembly*, e portanto é dependente de arquitetura. Felizmente, como foram realizados trabalhos anteriores com processadores de arquitetura ARMv4T, aproveitou-se a compatibilidade binária, pois ARMv4T é um subconjunto das instruções ARMv5TE (ARM, 2005), utilizadas na arquitetura Intel XScale.

Aproveitou-se o *run-time* C++ para se adicionar as rotinas responsáveis pela manutenção da pilha do sistema. A arquitetura ARM possui um comportamento indefinido para acessos desalinhados à memória principal. Garantiu-se então, na inicialização da pilha, um endereço alinhado para a mesma. O código gerado pelo compilador faz a manutenção desta propriedade.

Além disso, na arquitetura ARMv5TE, o processador possui pilhas diferenciadas para diversos modos de funcionamento. Isto significa que a pilha utilizada no tratamento de interrupções é diferente da pilha utilizada pela aplicação. Como o EPOS é utilizado em modo *library* neste trabalho, a aplicação e o tratador de interrupções dividem o mesmo espaço de memória, ou seja, devem ser executados em um único modo de operação do processador, com uma única pilha. Como isso não é possível pelo funcionamento padrão da arquitetura, uma adaptação técnica foi realizada no *run-time* C++ para que a existência de uma única pilha fosse transparente ao sistema EPOS.

3.1.5 Implementação de Mediadores

Os mediadores do EPOS são a interface para a acesso de componentes da plataforma de hardware (FRöHLICH, 2001). Assim como as rotinas de pré-configuração, cada mediador é um componente do sistema dependente de máquina. De fato, a implementação dos mediadores para a plataforma PXA255 é o passo final e fundamental para a criação do porte.

Para os fins deste trabalho, não necessariamente todos os mediadores do sistema precisam ser implementados. Isso acontece porque as abstrações que serão utilizadas para experimento estão restringidas a componentes de processo e tempo. Não é preciso, por exemplo, que se implemente mediadores para a interface *bluetooth*, pois sincronizadores, escalonadores, etc. não utilizam este tipo de mediador.

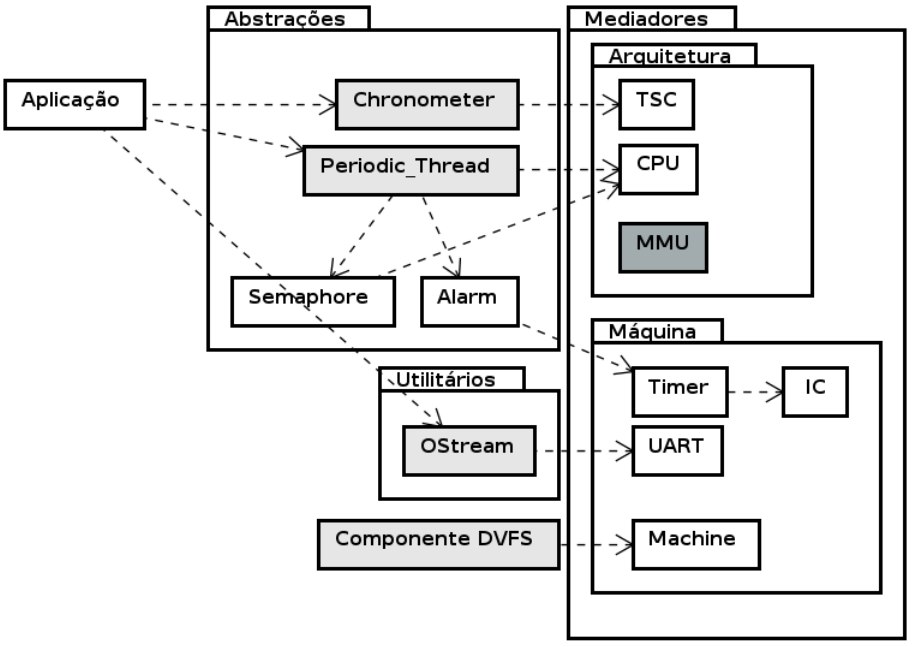

Figura 3.2: Análise de dependências entre aplicação, abstrações e mediadores para o porte

Em uma análise para definir os mediadores necessários, o diagrama de classes apresentado na figura 3.2 foi desenvolvido. A aplicação criada posteriormente para experimentos utiliza diretamente somente duas abstrações: *Periodic_Thread* (tarefas periódicas) e *Chronometer* (cronômetro virtual), além do utilitário *OStream* (saída de dados). Estas dependências exigem, indiretamente, a implementação dos mediadores: *CPU*, *Timer* (temporizador), *UART*, *TSC* (contador temporizado) e *IC* (controlador de interrupções).

O mediador de *MMU* foi também escolhido para implementação, pois foi delegado a ele o remapeamento dos vetores de interrupção (subseção 3.1.4), procedimento esclarecido mais adiante.

O mediador *Machine*, ou mediador de máquina, é responsável pelas características presentes no PXA255, mas que excedem o conceito de arquitetura (interface binária) do processador. Logo, as atividades de alteração de tensão e frequência devem ser atribuídas a este mediador.

A seguir, são apresentadas informações breves sobre a implementação de cada um destes mediadores.

CPU

Este mediador é responsável por características como a troca de contexto, entrada e saída em registradores, habilitação de interrupções, e criação de pilhas para novas *threads* no sistema. Sendo assim, é um mediador intimamente ligado com as questões de gerenciamento de tarefas no EPOS. Boa parte deste mediador pôde ser herdada do suporte às arquiteturas ARM utilizadas em projetos anteriores. Ainda assim, pequenas modificações foram necessárias, pois algumas operações de entrada e saída envolvem coprocessadores específicos da arquitetura Intel XScale.

Timer

Este mediador é requerido por várias abstrações relacionadas a processos, como por exemplo, escalonadores ativos. Como a abstração *Periodic_Thread* necessita de invocações periódicas, este mediador necessita de implementação.

UART

O EPOS utiliza este mediador muito cedo em sua inicialização, de modo que seja possível facilitar a depuração em máquinas reais, onde o uso de emuladores e JTAG pode não ser uma realidade. Utilizando a placa Connex em conjunto com sua expansão HWUART, é possível obter informações do sistema em tempo de execução.

TSC

O TSC (*Time Stamp Counter*) é um dispositivo que auxilia a medição de tempo em alta resolução.

IC

A arquitetura ARM necessita de um roteamento de interrupções em alto nível. Este mediador abstrai itens referentes aos indicadores de interrupção da máquina, para a decisão de qual tratador instalado chamar. Este mediador é necessário para o funcionamento ideal, por exemplo, do mediador *Timer*.

MMU

Por padrão, os processadores ARM (como é o caso do Intel XScale) utilizam um vetor de interrupções que encontra-se no endereço 0 emitido pela CPU. Como na máquina Connex este endereço refere-se à *flash* programável, onde já encontra-se instalado um vetor de interrupções do U-Boot, é preciso que seja configurado, através da MMU presente no PXA255, um remapeamento dos endereços emitidos (virtuais) para novos endereços físicos. Assim, é possível traduzir o endereço padrão do vetor de interrupções (endereço 0), que está em *flash*, para um endereço qualquer em RAM, sendo possível instalar nesta memória um vetor de interrupções apropriado ao EPOS.

A MMU do PXA255 segue a VMSAv5 (*Virtual Memory System Architecture version 5*), como descrita no *Arm Reference Manual* (ARM, 2005). Na inicialização do mediador de MMU, foi inserida uma rotina que preenche a tabela de descritores de memória. Esses descritores fazem o remapeamento da faixa inicial de 1MB de memória em *flash*, para a primeira faixa de 1MB em RAM. Assim, sempre que uma interrupção for chamada, o acesso ao endereço virtual 0, onde se encontra o vetor de interrupções do U-Boot, será na realidade um acesso ao endereço real em RAM, onde o vetor de interrupções preenchido pelo EPOS se encontra.

Machine

Este mediador refere-se ao PXA255 e seus componentes. É ele que deve possuir a interface para que o modo de operação do processador seja alterado. A seção 3.1.6 explica com mais detalhes a implementação do suporte a DVFS.

3.1.6 Implementação do Suporte a DVFS

Como citado na seção 3.1.1, o PXA255 possui suporte a alteração dinâmica de frequência e tensão para três dispositivos: núcleo do processador, barramento do sistema (*PXBus*) e memória SDRAM. A relação entre os principais componentes do PXA255 pode ser observada na figura 3.3. Na mesma figura, em destaque, encontram-se os dispositivos que suportam DVFS.

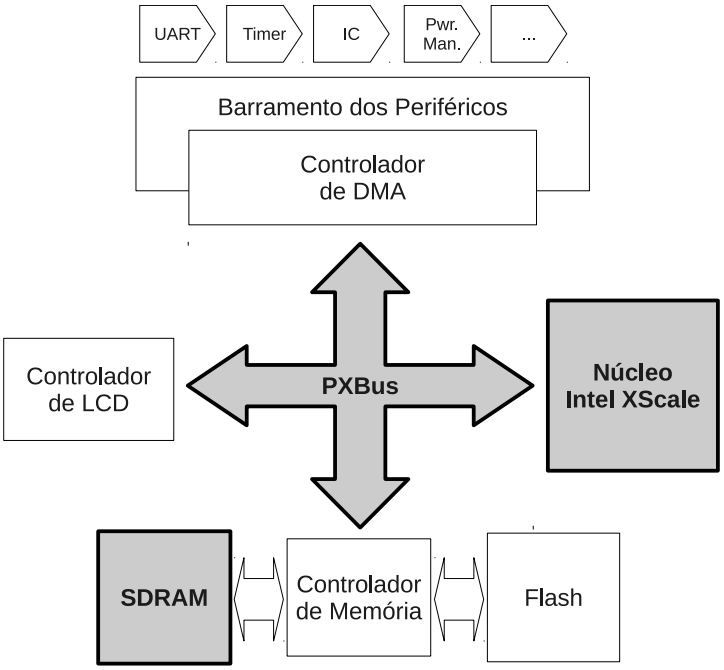

Figura 3.3: Representação dos componentes do processador PXA255

Segundo o *Intel PXA255 Developer's Manual* (INTEL, 2004), existe um conjunto de configurações de frequência válidas para estes dispositivos. Este conjunto é apresentado na tabela 3.1.

No PXA255, a alteração dinâmica de frequência e tensão é feita através da FCS (*Frequency Change Sequence*). Para qualquer alteração na configuração corrente do processador, invocando a FCS, os seguintes passos devem ser executados diretamente por software (INTEL, 2004):

1. Configurar o controlador de memória para que a frequência de atualização da SDRAM seja compatível com as frequências anterior e posterior à FCS.
2. Desabilitar o controlador de LCD.
3. Configurar os periféricos para que eles manipulem o atraso que a FCS pode ocasionar no controlador de DMA.

Tensão do processador (V)	Frequência do Núcleo (MHz)	Frequência do PXBus (MHz)	Frequência da SDRAM (MHz)
1.0	99,5	50	99,5
1.0	199,1	50	99,5
1.1	298,6	50	99,5
1.0	132,7	66	66
1.0	199,1	99,5	99,5
1.1	298,6	99,5	99,5
1.3	398,1	99,5	99,5
1.1	265,4	132,7	66
1.3	331,8	165,9	83
1.3	398,1	186	99,5

Tabela 3.1: Possíveis configurações de tensão e frequência para o PXA255

4. Desabilitar periféricos que não podem acomodar um atraso maior que 500μs no tratamento de suas interrupções. Isto se dá porque todas interrupções geradas durante a FCS são tratadas apenas ao final do processo de configuração.

5. Programar o registrador CCCR (*Core Clock Configuration Register*), de modo a representar uma das configurações apresentadas na tabela 3.1.

6. Invocar a FCS através da escrita do primeiro bit à direita do registrador CCLKCFG.

Após a execução do item 6, o fluxo de execução de instruções continua normalmente. Os itens 2, 3 e 4 foram ignorados para a implementação deste porte, pois os controladores de LCD e DMA não foram utilizados em nenhum momento. Além disso, o tratamento de interrupções não foi prejudicado conforme o especificado no item 4.

Para algumas das configurações apresentadas na tabela 3.1, não basta apenas a invocação da FCS pelo registrador CCLKCFG, mas também a invocação do modo *TURBO*, realizada pela escrita do segundo bit à direita do mesmo registrador.

Os métodos para invocação da FCS e do modo TURBO foram encapsulados no mediador de máquina, conforme apresentado no diagrama de classes da figura 3.4.

Para facilitar a interação entre heurísticas DVFS e o mediador de máquina, uma interface chamada *DVFS_Machine* foi criada. Essa interface é capaz de fornecer às abstrações do sistema as configurações disponíveis e um meio de aplicá-las. É ela a responsável pelo cumprimento dos passos descritos na enumeração anterior.

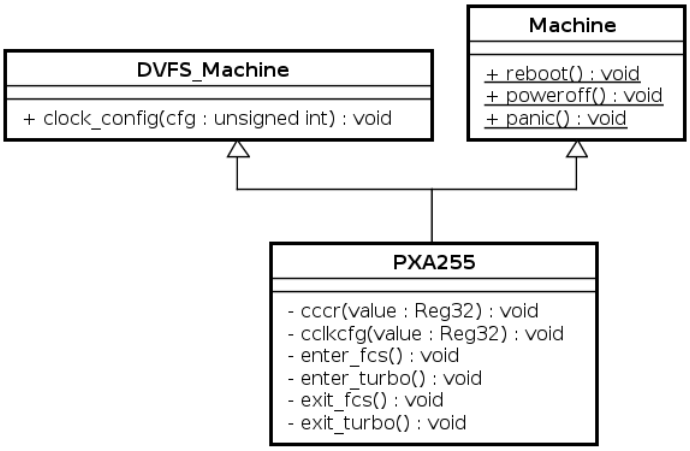

Figura 3.4: Diagrama de classes do mediador de máquina para PXA255

3.2 Criação do Ambiente RT-DVFS

Para a adequação com o modelo de sistema de tempo real apresentado na seção 2.1.1, é utilizada a abstração do EPOS chamada *Periodic_Thread*. Uma extensão para seu funcionamento foi criada, de modo a captar-se os eventos necessários para a implementação das heurísticas *Cycle-conserving DVS* para EDF e *Static Voltage Scaling* para EDF.

3.2.1 *Threads* Periódicas no EPOS

A abstração *Periodic_Thread* do EPOS oferece suporte à execução periódica de tarefas no sistema. A partir dessa abstração, é possível criar um objeto ao qual existem associados um período e uma rotina a ser executada. Este período é representado, na imagem 3.5, como sendo o atributo *period* da classe *Alarm*. A rotina associada à tarefa periódica é representada na mesma imagem pelo atributo *entry*, herdado por *Periodic_Thread* da classe *Thread*.

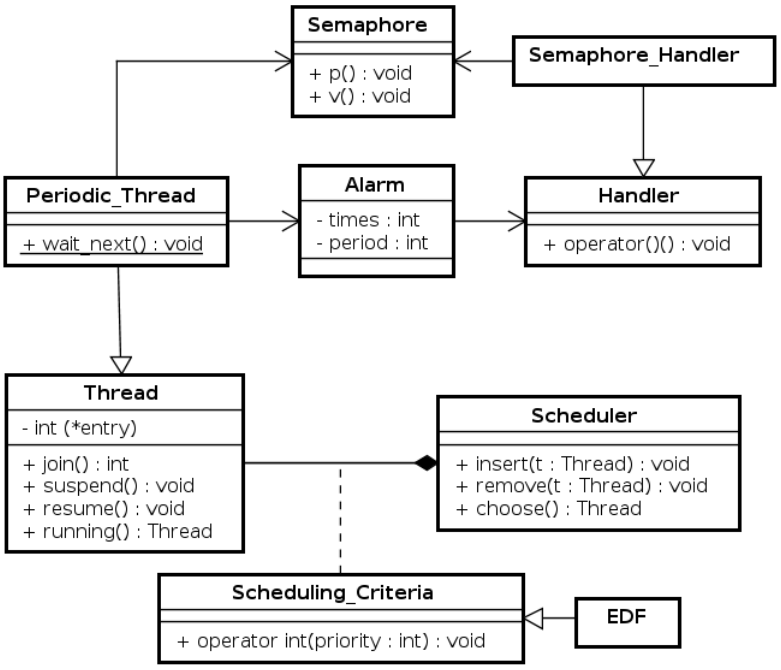

Figura 3.5: Diagrama de classes apresentando a abstração *Periodic_Thread*

A rotina a ser executada periodicamente deve ser uma função C++ não-membro (STROUSTRUP, 1997), que deve explicitamente se comportar como um laço, invocando dentro do bloco de repetição a função membro estática *Periodic_Thread::wait_next*. O exemplo a seguir (3.1) demonstra este artefato para um número *iterations* de repetições da tarefa. Observe que é possível estratificar a execução da tarefa em um ponto para sua inicialização, outro ponto com o corpo de código que será executado repetidamente, e por final, um trecho de código que representa a finalização da tarefa.

```
int periodic_task(int iterations){

    // inicialização

    for(int i = 0; i < iterations; i++){
        // corpo da tarefa
        Periodic_Thread::wait_next();
    }

    // finalização

    return 0;
}
```

Texto C++ 3.1: Exemplo de uma tarefa como uma função não-membro

A *thread* que executa a função membro *Periodic_Thread::wait_next*, através da operação "*p*" sobre semáforo associado à abstração *Periodic_Thread*, perde o processador. Esta *thread* entrará em um estado onde ela não ocupará mais a CPU até que a operação "*v*" seja realizada sobre o mesmo semáforo. Para oferecer o comportamento periódico às tarefas, a abstração *Alarm* é então utilizada.

A abstração *Alarm* invoca periodicamente um objeto função (instância da classe *Handler* apresentada na figura 3.5), neste caso especializado para executar a operação "*v*" sobre o semáforo que "impede" a repetição do corpo do laço. A *thread* pode então, voltar a ganhar a CPU segundo os critérios do escalonador. Ao recuperar a execução, ela volta ao ponto onde invocou *Periodic_Thread::wait_next*, obedecendo ao comportamento do laço escrito.

O escalonador EDF entra como critério de ordenação utilizado pela classe *Scheduler*, também apresentada na figura 3.5. Para configurar o critério utilizado no escalonamento, o EPOS utiliza o padrão de projeto *traits*, oferecendo a escolha da política como uma configuração estática do sistema (FRöHLICH, 2001).

3.2.2 Modificações no Modelo de *Threads*

Para implementação das heurísticas para RT-DVFS, algumas modificações foram necessárias no sistema EPOS. Primeiramente, as heurísticas precisam estar cientes de eventos referentes às tarefas, como inicialização, término ou até mesmo alterações no conjunto de tarefas. Ainda, o sistema deve reagir a eventos como a troca de contexto, já que o modelo utilizado é preemptivo. Para que fosse possível atribuir ações a estes eventos, algumas modificações foram necessárias no modelo de *threads* periódicas do EPOS, como apresentado no diagrama de classes da figura 3.6.

Para reagir à troca de contexto, dois métodos foram criados na interface da classe *Thread*: *pre_cs* e *post_cs*. Esses métodos são dedicados a extrair os pontos de execução antes e depois da troca de contexto entre *threads*. Em C++, estes foram criados como funções membro virtuais, passíveis de serem sobrecarregados por classes filhas de *Thread*. A interceptação destes eventos auxilia no cálculo do tempo de computação efetivamente utilizado pela tarefa.

Para reagir aos outros eventos citados, a classe *Periodic_Thread* foi especializada. Uma filha chamada *RTDVS_Thread* foi criada como uma abstração a fim de capturar estes eventos e repassá-los à heurística a ser utilizada. Ainda, a abstração

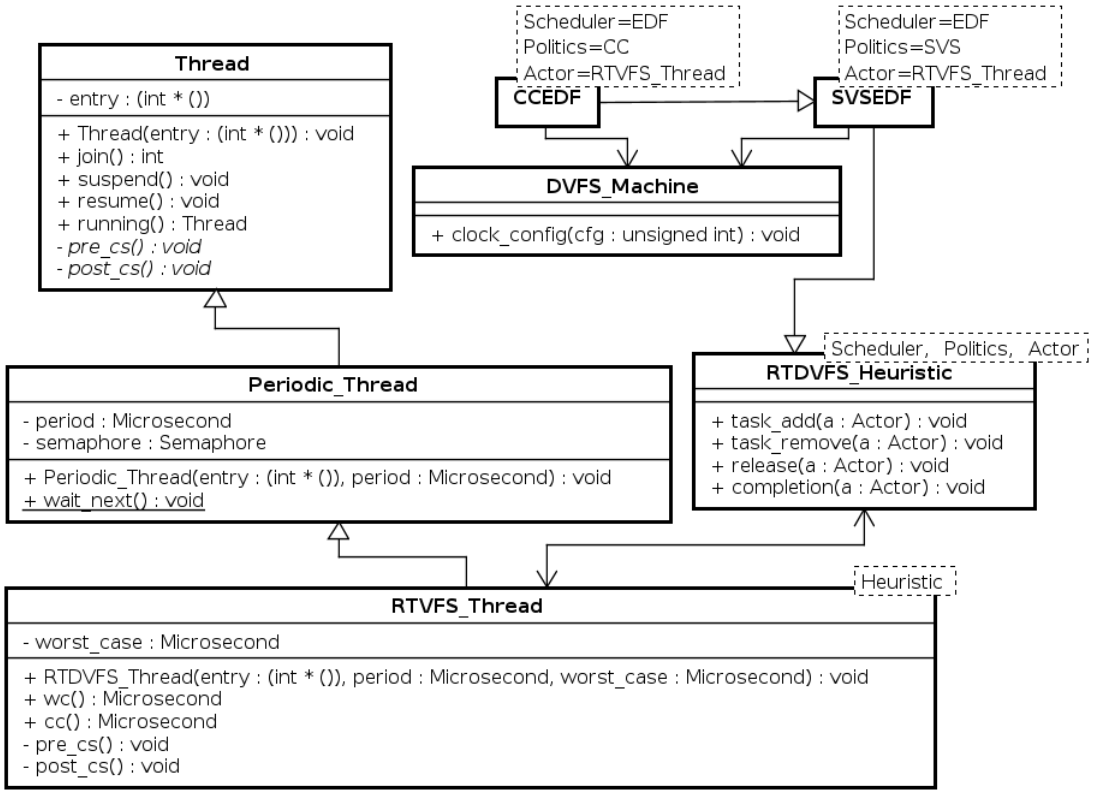

Figura 3.6: Diagrama de classes demonstrando as relações da *RTDVFS_Thread*

RTDVFS_Thread recebe como parâmetros de construção o pior tempo de computação da tarefa que ela representa.

Captura de Modificações no Conjunto de Tarefas

A reação à alteração no conjunto de tarefas é dada a partir de duas maneiras: adição ou remoção de tarefas. Para capturar o evento de adição, em toda construção de uma nova *RTDVFS_Thread*, a classe *RTDVS_Heuristic* tem o método *task_add* invocado, conforme apresentado no diagrama de sequência da figura 3.7. Analogamente, na destruição de uma *RTDVS_Thread*, o método *task_remove* é chamado.

Captura da Inicialização e Término de uma Tarefa

Os eventos de inicialização e término de tarefas são capturados através da reimplementação do método *wait_next* para a *RTDVS_Thread*. O diagrama de sequência da figura 3.8 mostra o momento em que estes eventos são reportados à heurística,

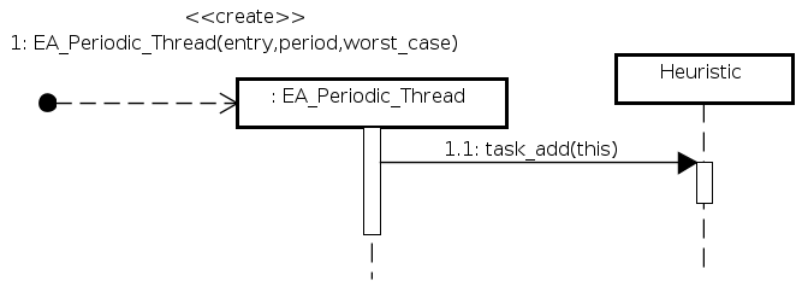

Figura 3.7: Diagrama de sequência demonstrando a criação de uma nova *RTDVS_Thread*

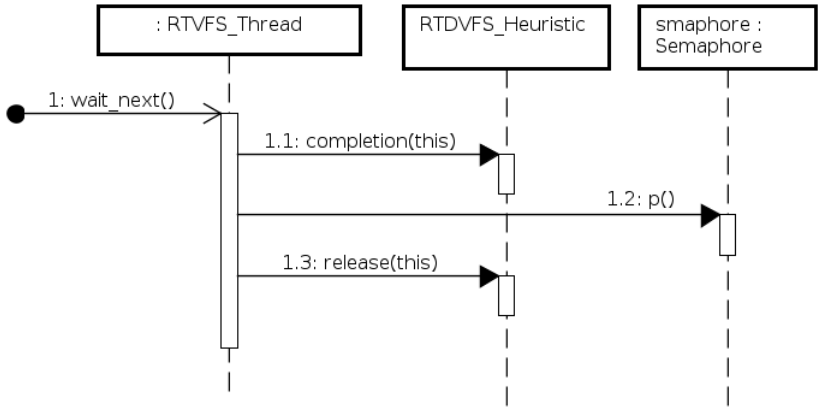

Figura 3.8: Diagrama de sequência demonstrando o comportamento do método *wait_next* da classe *RTDVFS_Thread*

anteriormente e posteriormente à aquisição de recurso no semáforo pertencente à *thread* periódica (mecanismo explicado anteriormente na seção 3.2.1).

Heurísticas

A implementação das heurísticas foram realizadas na forma de especialização de *templates* em C++. A escolha da heurística pode ser programada estaticamente, sendo que cada heurística é uma especialização de três parâmetros: *Scheduler* (o escalonador a ser usado, neste trabalho sempre EDF), *Politics* (a política a ser utilizada: *Static Voltage Scaling* ou *Cycle-conserving*) e *Actor* (a abstração que utiliza a heurística, neste trabalho sempre *RTDVFS_Thread*).

3.2.3 Implementação de *Static Voltage Scaling* para EDF

Para implementar a heurística *Static Voltage Scaling* para EDF, uma especialização *SVSEDF* foi criada a partir da classe *RTDVFS_Heuristic* (figura 3.6).

A cada chamada de *task_add* ou *task_remove*, uma lista interna (*task_set*) a *SVSEDF* contendo objetos *RTDVFS_Thread* é atualizada. Quando este evento ocorre, o método *select_frequency*, escrito nos moldes do algoritmo 1, é chamado. O trecho abaixo mostra o código em C++ do método citado:

```
        static void optmize_frequency(double usage){

            unsigned int j;
            for(j = 0; j < DVFS_Machine::n_clock_configs; j++){
                if(usage <= DVFS_Machine::clock_configs[j].norm){
                    DVFS_Machine::clock_config(j);
                    return;
                }
            }

            DVFS_Machine::clock_config(DVFS_Machine::max_clock_config);
        }

        static void select_frequency(){

            RTDVFS_Thread::List::Iterator i;
            double u = 0;

            for(i = task_set.begin(); i != task_set.end(); i++){
                RTDVFS_Thread* t = i->object();
                u += t->wc()/(double)t->period();
            }

            optmize_frequency(u);
        }
```

Texto C++ 3.2: Método SVSEDF::select_frequency

Observe que o atributo *DVFS_Machine::clock_configs[j].norm* é utilizado. Este atributo apresenta a frequência normalizada de uma suposta configuração para a *DVFS_Machine*. Caso uma configuração de frequência normalizada menor ou igual a 1 não seja encontrada, a frequência máxima é utilizada.

3.2.4 Implementação de *Cycle-conserving DVS* para EDF

Na implementação da heurística dinâmica, um método *select_frequency* também é implementado, mas neste caso, em vez do cálculo da utilização de pior caso, é feito

o cálculo da utilização real quando a tarefa encontra-se terminada.

```
        static void select_frequency(){

            RTDVFS_Thread::List::Iterator i;
            double u = 0;

            for(i = task_set.begin(); i != task_set.end(); i++){
                RTDVFS_Thread* t = i->object();
                if(t->cc() == 0)
                    u += t->wc()/(double)t->period();
                else
                    u += t->cc()/(double)t->period();
            }

            optmize_frequency(u);
        }

        static void release(Periodic_Thread* a){
            select_frequency();
        }

        static void completion(Periodic_Thread* a){
            select_frequency();
        }
```

Texto C++ 3.3: Método CCEDF::select_frequency

A heurística dinâmica também utiliza a adição e remoção de tarefas para atualizar a lista *task_set*. Deste modo, a inicialização da execução do conjunto de tarefas sempre se dá com o processador configurado para o desempenho de pior caso de utilização. O efeito dinâmico é adquirido através da invocação de *CCEDF::select_frequency* pelos eventos de término e início de execução de das *threads* periódicas.

4 Experimentos e Resultados

Este capítulo apresenta como foram conduzidos os experimentos criados para avaliar a economia de energia obtida com o ambiente RT-DVFS construído. Os resultados destes experimentos são apresentados e analisados.

4.1 Ambiente Experimental

O ambiente experimental foi criado com base em um conjunto pré-definido de tarefas periódicas. Como métrica para avaliação, utilizou-se a potência média de todo o sistema, composto de uma placa-mãe Gumstix Connex e sua expansão HWUART, utilizada para transmissão de dados sobre RS232.

4.1.1 Tarefa Canônica

Para a avaliação, uma tarefa canônica foi concebida, cujo papel é comportar-se como uma tarefa de tempo real periódica. Sua escrita em C++ dá-se como uma função não-membro que recebe como argumento dois inteiros sem sinal. O primeiro representando a ocupação da tarefa, e um segundo, representando o número de invocações que a tarefa periódica receberá, conforme mostrado no texto em C++ 4.1.

A simulação da ocupação do processador pela tarefa nada mais é do que um laço vazio, que se repete por um número constante arbitrário de vezes. Para diferentes ocupações do processador, utiliza-se diferentes constantes para a repetição do laço. Esta abordagem é a mesma utilizada no trabalho de Lin, Song e Cheng (2010), onde os autores propõem um *framework* destinado à avaliação de heurísticas para RT-DVFS. Vale salientar que a tarefa apresentada simula tarefas *CPU-bound*, ou seja, tarefas que tem seu tempo de execução limitado principalmente pela frequência da CPU, utilizando poucas instruções que fazem acesso à memória principal.

```
int cononical_task(unsgiend int busy, unsigned int iterations){

    for(unsigned int i = iterations; i > 0; i--){
        for(volatile unsigned int b = busy; b > 0; b--);
        RTDVFS_Thread::wait_next();
    }

    return 0;
}
```

Texto C++ 4.1: Exemplo de uma tarefa canonica como uma função não-membro

4.1.2 Aplicações EPOS para Experimentação

Utilizando-se então a tarefa canônica, cinco aplicações para o sistema EPOS foram criadas, cada uma composta por três tarefas periódicas (utilizando=se a abstração *RTDVFS_Thread*) baseadas na tarefa canônica apresentada. As aplicações foram elaboradas para simularem 100, 80, 60, 40 e 20% de ocupação total de CPU, sempre considerando um acréscimo de 5% na ocupação total como sobrecarga do próprio sistema operacional e tempo de alteração de tensão e frequência para o processador. Isto significa que, uma aplicação que simula 100% de uso, na realidade utiliza 95% do tempo de CPU com suas tarefas. Concluindo, para uma utilização u, cada pior tempo de computação C_i' utilizado na aplicação é na realidade $C_i' = (u - 0,05u)C_i$.

Para se conseguir o número de repetições do laço de cada tarefa canônica, constantes foram empiricamente aplicadas pelo programador. Utilizando-se o método *cc* da abstração *RTDVFS_Thread*, que por sua vez utiliza o TSC do sistema, foi calculada a utilização de cada tarefa. As constantes assumidas para experimentação passaram por 10 sessões de execução por 5 segundos em configuração de frequência máxima, onde a utilização esperada pelo programador foi atingida com um desvio padrão menor que 0.01% da média amostral. O pior tempo de computação, passado como parâmetro para a criação de cada *RTDVFS_Thread*, foi extraído como o maior tempo de computação obtido da amostragem citada.

Cada uma das aplicações cria um conjunto de três tarefas periódicas, estas com 500, 400 e 300 milisegundos de período. A aplicação indica um número de repetições para cada tarefa, de modo que elas se repitam durante pelo menos 305 segundos. A interrupção do *timer* de sistema do EPOS foi reduzido para um período de 100 milisegundos, sendo compatível com os períodos das tarefas apresentadas.

Do conjunto de configurações para tensão e frequência apresentados na tabela

Tensão do processador (V)	Frequência do Núcleo (MHz)	Frequência do PXBus (MHz)	Frequência da SDRAM (MHz)
1.0	99,5	50,0	99,5
1.0	199,1	99,5	99,5
1.1	298,6	99,5	99,5
1.3	398,1	186,0	99,5

Tabela 4.1: Configurações de tensão e frequência do PXA255 utilizadas nos experimentos

3.1, o subconjunto mostrado na tabela 4.1 foi utilizado. Os critérios para a escolha destas configurações foram:

- Não há variação na frequência da memória principal, comportamento que não será avaliado neste trabalho.
- Para todos dispositivos que variam a frequência, o subconjunto oferece uma ordem total dos elementos (hipótese necessária para o funcionamento das heurísticas implementadas).
- Uniformidade (cada frequência do núcleo é um múltiplo aproximado de 0,25, ou seja, conforme a teoria apresentada na seção 2.3.1, há α 0,25, 0,50, 0,75 e 1,00).

As caches de dados e de instruções foram desligadas para a realização dos experimentos.

4.1.3 Configuração para Medições

Para a realização das medições, a placa Connex foi devidamente ligada à sua fonte de corrente contínua. Em seu circuito de alimentação, um resistor de 1Ω com 1% de variação foi colocado em série com o cabo da fonte, assim como mostrado pela figura 4.1.

Utilizando-se os canais de um osciloscópio digital, dois pontos do circuito de alimentação foram amostrados a $5Hz$: a queda de tensão no resistor apresentado e a tensão na fonte de alimentação, conforme a figura 4.1.

A tensão medida entre as pontas do resistor, pela Lei de Ohm, pode ser convertida na corrente aproximada que passa pelo circuito. Através dessa corrente I e a tensão

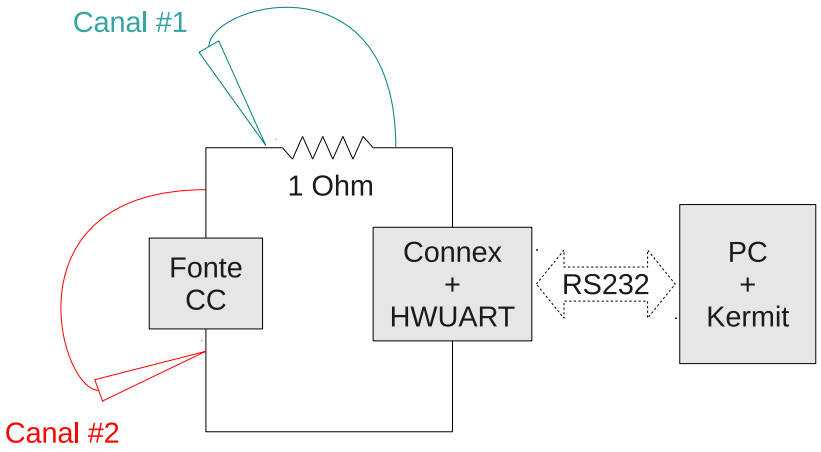

Figura 4.1: Configuração do experimento utilizado para medições

da fonte de alimentação U, pode-se então ser calculada em um instante a potência P dissipada pelo sistema, esta sendo $P = U \cdot I$.

Para a avaliação dos resultados, a potência média P_{med} foi escolhida como métrica. Sendo $U(t)$ e $I(t)$, respectivamente, a tensão e corrente do sistema no instante t, pode-se calcular a potência média de um instante 0 até T como sendo:

$$P_{med} = \frac{1}{T} \cdot \int_0^T U(t) \cdot I(t) dt.$$

A escolha da potência média como métrica deve-se ao fato de ela representar, substancialmente, o trabalho elétrico realizado por todo o dispositivo (placa-mãe e expansão). Devido à alta granularidade temporal dos eventos, a sumarização através da média não traz prejuízo à leitura do comportamento das heurísticas.

Para a transmissão das aplicações à placa Connex, essa foi ligada a um PC utilizando-se o padrão RS232.

4.1.4 Coleta e Compilação dos Dados

As aplicações de teste foram carregadas utilizando-se a facilidade Kermit do U-Boot, como já relatado no capítulo 3. A coleta das amostras adquiridas pelo osciloscópio foi feita através de um "*pendrive*", no qual é possível gravar as leituras obtidas. Um programa escrito na linguagem Python foi desenvolvido com o fim de processar as amostras contidas no pendrive, calculando então a potência média utilizada para a execução das aplicações.

Para cada aplicação, repetiu-se 5 vezes o processo de carregamento em memória e execução do sistema. A potência média foi calculada para estas 5 amostras, o desvio padrão manteve-se sempre entre 1% e 2% da potência média. Os resultados obtidos são apresentados e discutidos na seção seguinte (4.2).

4.2 Resultados

O gráfico apresentado na figura 4.2, mostra a potência média obtida para uma utilização real igual à utilização de pior caso, ou seja, o caso onde a utilização real por cada tarefa é igual à utilização referente ao pior tempo de computação passado como parâmetro para a construção do objeto *RTDVFS_Thread*.

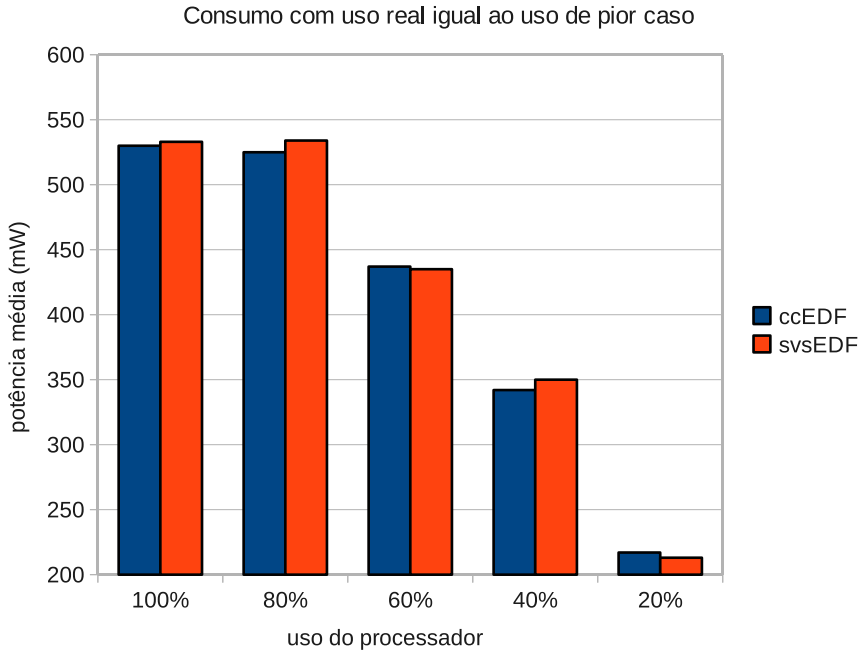

Figura 4.2: Gráfico da potência média com utilização real igual a de pior caso

Observe que para as duas heurísticas implementadas, tanto *Cycle-conserving* (ccEDF) como para *Static Voltage Scaling* (svsEDF), não existe diferencial significativo no consumo entre os dois mecanismos. Isso se dá porque a heurística ccEDF tende a comportar-se como a heurística svsEDF quando a utilização real e de pior caso estão muito próximas, já que o resultado do cálculo estático e dinâmico da utilização neste caso são equivalentes. A pequena variação entre o consumo das duas heurísticas nos mesmos casos ocupação acontece devida a erros experimentais e à pequena variação presente na utilização real das tarefas.

Ainda, o gráfico da figura 4.2 é capaz de mostrar diretamente os efeitos que as configurações de tensão e frequência causam no consumo de energia total do sistema, já que neste caso as duas heurísticas são equivalentes. A tabela 4.2 apresenta a possível relação entre o uso de processador e a configuração selecionada pelo heurística estática svsEDF, segundo o que é apresentado no gráfico. Há variação na potência média para frequência normalizada 1,00 devido a erros experimentais.

Uso	Frequência de CPU	Frequência normalizada	Tensão	Potência média
100%	$398,1MHz$	$1,00$	$1.3V$	$533mW$
80%	$398,1MHz$	$1,00$	$1.3V$	$534mW$
60%	$298,6MHz$	$0,75$	$1.1V$	$435mW$
40%	$199,1MHz$	$0,50$	$1.0V$	$350mW$
20%	$99,5MHz$	$0,25$	$1.0V$	$213mW$

Tabela 4.2: Relação entre uso de processador e configurações de tensão e frequência utilizadas

O gráfico apresentado na figura 4.3 mostra o comportamento das heurísticas com utilização de pior caso total, mas apresentando uma variação na utilização real entre 20 e 100%. Isto significa que, mesmo as tarefas tendo como parâmetro um pior tempo de computação que causa 100% de utilização, na realidade as tarefa canônicas correspondentes realizam apenas uma porcentagem fixa de 20 a 100% do pior caso de computação.

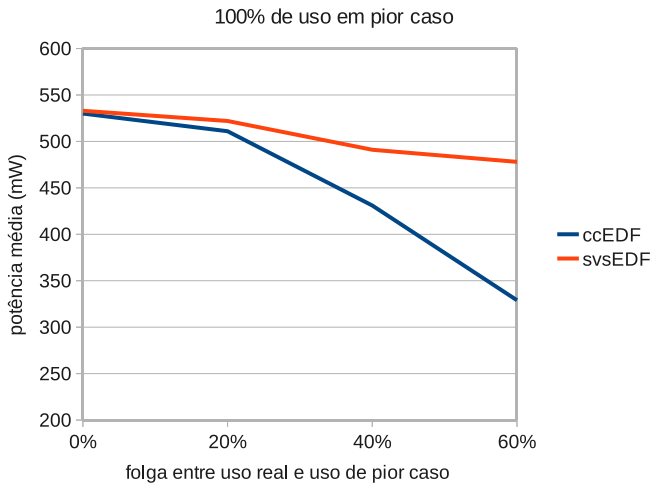

Figura 4.3: Potência média em 100% de uso de pior caso do processador com variação no uso real

A variação na utilização real apresentada demonstra a atuação da heurística dinâmica ccEDF, que leva em conta dados obtidos durante a execução do sistema. Ob-

serve que as curvas são muito próximas até a utilização real de 80%, efeito causado pela indisponibilidade de uma configuração de tensão e frequência que represente um desempenho entre 0,75 e 1,00. O ganho só se torna observável a partir de 60% de uso real, onde a heurística ccEDF calcula o uso dinâmico inferior ao de pior caso obtido estaticamente. A economia apresentada pela heurística estática é devido à redução no uso físico de componentes do circuito do processador.

Observe que, no funcionamento da heurística dinâmica, o pior tempo de computação configurado estaticamente para cada tarefa quase sempre elevará a utilização calculada dinamicamente. O caso onde isto não acontece é quando todas tarefas foram completadas, sabendo-se o tempo de computação real utilizado por cada uma. De qualquer modo, na forma como a heurística ccEDF é escrita, este valor é "sujado" com o pior caso em novas requisições de tarefas. Aqui encontra-se a principal limitação da heurística *Cycle-conserving*. É este tipo de problema que heurísticas de previsão (citadas na subseção 2.3.1) tentam resolver.

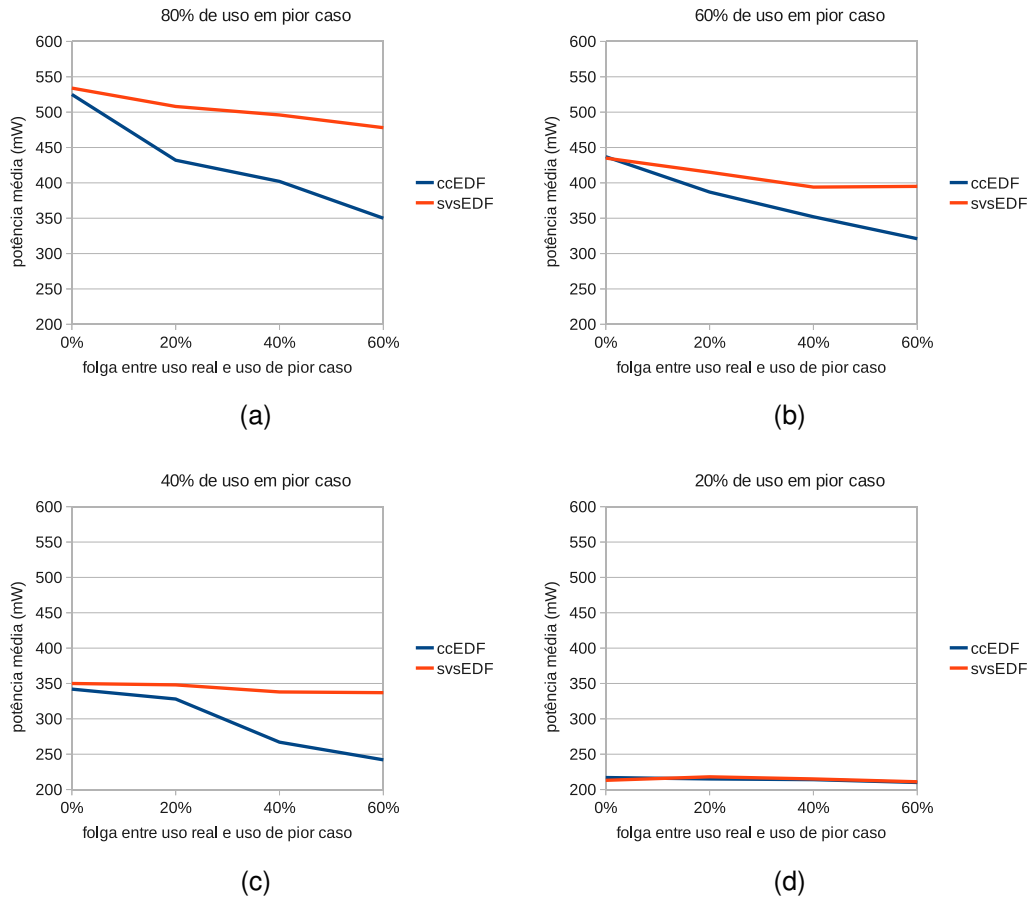

Figura 4.4: Potência média para uso de pior caso de 20 a 80% com variação no uso real

O comportamento para outras utilizações de pior caso e variações no uso real, pode ser visualizado na figura 4.4. É possível observar, novamente, uma queda no consumo utilizando-se a heurística svsEDF (itens 4.4a, 4.4b, 4.4c e 4.4d) isto se dá devido à redução no uso do processador. É interessante salientar que esta queda tende-se a estabilizar com redução do uso, como pode-se observar em 4.4d. Isto acontece porque em d, a maior influência no consumo total do sistema passa a ser a energia estática consumida pelo PXA255, ou seja, a energia consumida não referente às consecutivas trocas de estado do circuito digital.

Ainda sobre 4.4d, devido à baixa granularidade das configurações de frequência utilizadas, não há configurações de desempenho tão inferior sendo utilizadas para uso do processador inferior a 25%, a heurística ccEDF não causa nenhuma influência.

O sistema colocado em questão utilizando a heurística svsEDF pode ser entendido como um sistema de tempo real padrão com escalonamento EDF. Pode-se observar que mesmo não sendo ótima, a heurística ccEDF ainda obtém ganhos de até 25% (na figura 4.4a) neste cenário, e sem a perda de prazos. De qualquer modo, vale salientar que estes resultados sofreriam modificações caso as heurísticas fossem expostas em aplicações reais. Primeiramente, é imaginável que a folga entre uso real e de pior caso seria uma variável estocástica, como é de se esperar neste tipo de sistema. Em segundo, aplicações reais tendem a fazer um uso maior dos componentes do processador. Fatores importantes, como a influência da habilitação da memória cache do processador e aplicação de DVFS na SDRAM do PXA255 são temas para pesquisa futura.

5 Conclusões

Como resultados deste trabalho, o porte do EPOS foi realizado para uma plataforma de hardware com suporte a DVFS, além disso, obteve-se uma implementação de um ambiente RT-DVFS.

O EPOS agora possui um porte para o Intel PXA255, um processador com núcleo Intel XScale e suporte a DVFS para vários dispositivos internos. A plataforma é destinada a aplicações de alto desempenho e baixo consumo de energia. Um fator interessante é que, agora sobre o PXA255, portar novamente o sistema para um outro processador XScale torna-se uma tarefa mais simples, já que boa parte dos mediadores referentes à arquitetura podem ser reaproveitados.

As heurísticas *Cycle-conserving* e *Static Voltage Scaling* foram implementadas para o escalonador EDF, já presente no ambiente de tempo real do EPOS. As heurísticas implementadas foram avaliadas do ponto de vista do consumo de energia, e obtiveram o comportamento esperado para o nicho de aplicações presentes nos testes.

5.1 Contribuição

A maior contribuição deste trabalho encontra-se na implementação de um ambiente RT-DVFS sobre o EPOS. Grande parte das implementações de ambientes RT-DVFS utilizam o sistema operacional Linux. O EPOS é um sistema operacional nativamente desenvolvido para a produção de sistemas embarcados, além de seguir a metodologia ADESD (FRöHLICH, 2001), um cenário diferenciado das outras implementações presentes na literatura sobre o tema.

Ainda, as novas abstrações que foram criadas no sistema, permitem facilmente a criação de novas heurísticas. A classe *EA_Periodic_Thread* é capaz de fornecer metadados referentes a execução de tarefas periódicas. Utilizando-a em conjunto

com os eventos captados por *Heuristic*, basta-se estender esta classe para que uma nova política para economia de energia seja adicionada ao sistema. Vale salientar que, como proposto por Pillai e Shin (2001), as heurísticas continuam fracamente acopladas ao escalonador de tempo real do sistema operacional embarcado.

5.2 Trabalhos Futuros

Mesmo que com seus objetivos cumpridos, este trabalho ainda necessita de amadurecimento em um aspecto principal: *design* de software.

A abstração *Thread* do EPOS não permite captar os eventos de troca de contexto. O conceito utilizado neste trabalho para trabalhar com estes eventos foi a adição de polimorfismo dinâmico na classe *Thread*, o que leva a indireções no controle de fluxo da troca de contexto. Esta foi a alternativa utilizada pois o EPOS ainda utiliza *threads* não periódicas para representar a *thread* principal do sistema e a *thread* que ocupa o processador em inatividade. Procurar uma alternativa completamente estática provavelmente recairia em um reprojeto maior do sistema, o que está fora do escopo deste trabalho. Observe que a captura deste tipo de evento é interessante para adicionar flexibilidade para a criação de novas abstrações de processos no EPOS.

Além disso, como trabalhos futuros, ainda aponta-se:

- Criar experimentos com aplicações mais complexas ou reais. O estudos em RT-DVFS revelam que o comportamento para tarefas que tem um uso maior da memória é mais complexo, como apresentando por Snowdown, Ruocco e Heiser (2005).
- Realizar experimentos comparativos com as implementações existentes sobre Linux. Lin, Song e Cheng (2010) apresenta um *framework* e modelo para a comparação de heurísticas RT-DVFS. A reprodução dos experimentos realizados em ambiente Linux por essa publicação de Lin poderia revelar mais detalhes do comportamento destas heurísticas, já que o EPOS provavelmente acarretaria menos sobrecarga aos resultados.
- Já que o EPOS é orientado a aplicação, poderia-se realizar uma análise estática ou dinâmica a fim de presumir mecanismos ou parâmetros de heurísticas para RT-DVFS.

APÊNDICE A – Artigo

Este apêndice inclui o artigo sobre este trabalho de conclusão de curso segundo o artigo 17 do *Regimento Interno Para Elaboração de Trabalhos de Conclusão de Curso de Ciência da Computação*.

A criação de um ambiente RT-DVFS sobre o sistema EPOS

Gustavo Roberto Nardon Meira[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil

Resumo. *Várias implementações de sistemas RT-DVFS vem sendo estudadas na última década. Em sua maioria, estas implementações são realizadas em sistemas Linux, que exigem modificações para suporte de tempo real. O presente trabalho mostra a construção de um ambiente RT-DVFS sobre o EPOS, um sistema operacional de tempo real orientado a aplicação. Um suporte para a inserção de heurísticas RT-DVFS fracamente acopladas ao escalonador do sistema foi implementado. Os experimentos conduzidos demonstraram o ganho energético de heurísticas utilizadas em conjunto com o escalonador EDF.*

1. Introdução

Dispositivos embarcados alimentados por bateria trazem consigo restrições conflitantes: autonomia e desempenho. Ao mesmo tempo que devem economizar energia, ganham a responsabilidade sobre tarefas inteligentes que necessitam de processamento poderoso. Mesmo com a exigência de alto desempenho, o pico da demanda de computação costuma acontecer apenas em alguns momentos do funcionamento de sistemas de tempo real [Pillai e Shin 2001]. Em sistemas dedicados, onde se tem um maior conhecimento sobre o comportamento da aplicação, tem-se encontrado um palco interessante para a plicação de técnicas de DVFS em tempo real, também chamadas de RT-DVFS.

Grande parte dos estudos em RT-DVFS são realizados através de simulações, que possuem resultados restritos, devido a fatores imprevisíveis ou não modelados [Lin et al. 2010]. Dos trabalhos que desenvolvem implementações reais de suporte a RT-DVFS, comumente as demonstrações ocorrem em sistemas Linux com a adição de módulos ou extensões [Pillai e Shin 2001, Snowdown et al. 2005, Zhu e Mueller 2007, Lin et al. 2010]. O sistema operacional EPOS, onde realizamos nossa implementação, já possui suporte nativo a tarefas de tempo real [Marcondes et al. 2009]. Além disso, o EPOS é um sistema operacional que segue a metologia ADESD (*Application-Driven Embedded System Design*) [Fröhlich 2001], estratificando ainda mais o cenário utilizado neste trabalho.

Neste trabalho, utiliza-se o porte do EPOS para a plataforma de hardware PXA255, um processador Intel XScale com suporte a DVFS amplamente utilizado em trabalhos de RT-DVFS. As heurísticas para RT-DVFS *Static Voltage Scaling* e *Cycle-conserving* para o escalonador EDF [Pillai e Shin 2001] são utilizadas como estudo de caso para avaliação do ambiente criado. A próxima seção mostra as características da plataforma de hardware utilizada no trabalho. A seção 3 dá uma breve introdução ao sistema EPOS e mostra o funcionamento de seu ambiente de tempo real. Em sequência, a seção 4 descreve as modificações necessárias para a implementação do ambiente RT-DVFS. A seção 5 mostra a condução de experimentos e resultados de comparações entre as duas heurísticas. Finalmente, na seção 6, conclusões são apresentadas.

2. A Plataforma de Hardware

O processador PXA255 é uma plataforma de hardware destinada a aplicações que exigem alto desempenho e baixo consumo de energia [Intel 2004]. O processador possui suporte a alteração dinâmica de frequência e tensão para três dispositivos: núcleo do processador, barramento do sistema (*PXBus*) e memória SDRAM. A relação entre os principais componentes do PXA255 pode ser observada na figura 1. Na mesma figura, em destaque, encontram-se os dispositivos que suportam DVFS.

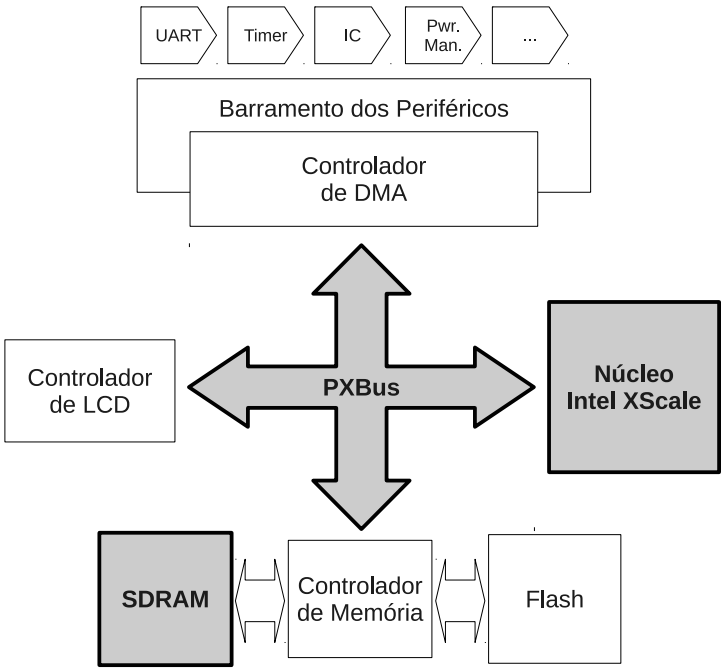

Figura 1. Relação entre os componentes do PXA255.

Segundo o *Intel PXA255 Developer's Manual* [Intel 2004], existe um conjunto de configurações de frequência válidas para estes dispositivos. Desse conjunto, para os fins deste trabalho, selecionamos apenas as configurações para as quais não há variação na frequência da SDRAM. Este subconjunto é apresentado na tabela 1.

Tensão do processador (V)	Frequência do Núcleo (MHz)	Frequência do PXBus (MHz)	Frequência da SDRAM (MHz)
1.0	99,5	50	99,5
1.0	199,1	99,5	99,5
1.1	298,6	99,5	99,5
1.3	398,1	186	99,5

Tabela 1. Subconjunto de configurações do PXA255 utilizadas neste trabalho.

3. EPOS e Tempo Real

O EPOS (*Embedded Parallel Operating System*) [Fröhlich 2001] é um sistema operacional direcionado a aplicações embarcadas de alto desempenho. Segue a metodologia ADESD (*Application-Driven Embedded System Design*), utilizando técnicas de

orientação a aspectos e programação estática em busca de se adequar às restrições presentes nas aplicações para o qual é utilizado. Em sua maior parte, o sistema é desenvolvido na linguagem C++.

De modo a manter portabilidade, entidades chamadas *mediadores de hardware* fornecem interfaces simples para acesso a funções dependentes de máquina. Estas interfaces são utilizadas por entidades mais abstratas do sistema, também chamadas de *abstrações*. Alguns exemplos de abstrações do EPOS seriam *Thread* e *Scheduler*, que utilizam mediadores de hardware como, por exemplo, o *Timer*.

A abstração *Periodic_Thread* do EPOS é responsável por oferecer suporte à execução de tarefas periódicas de tempo real no sistema. A partir dessa abstração, é possível criar um objeto ao qual existem associados um período e uma rotina a ser executada. Este período é representado, na imagem 2, como sendo o atributo *period* da classe *Alarm*. A rotina associada à tarefa periódica é representada na mesma imagem pelo atributo *entry*, herdado por *Periodic_Thread* da classe *Thread*.

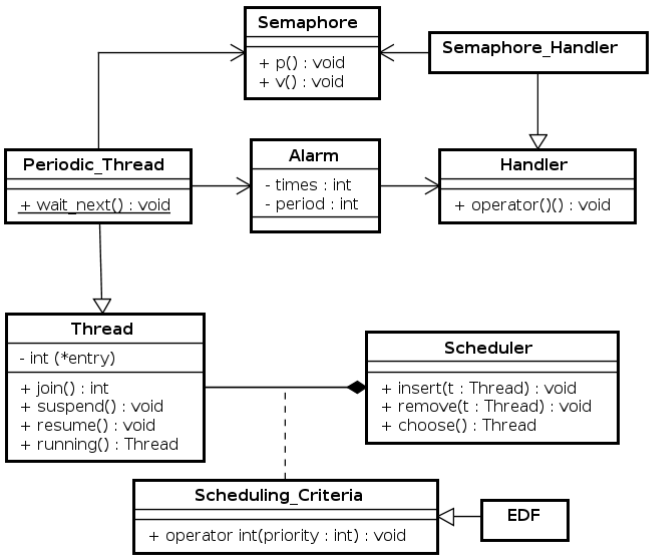

Figura 2. Diagrama de classes apresentando a abstração *Periodic_Thread*

A rotina especificada pelo atributo *entry* deve ser a tarefa exigida pela aplicação. Esta rotina deve ser explicitamente programada como um laço de repetição, e a cada iteração, a operação *Periodic_Thread::wait_next* deve ser chamada pelo programador. Esta chamada, através da operação "*p*" sobre o semáforo associado à abstração *Periodic_Thread*, faz com que a *thread* em execução perca o processador. Esta *thread* entrará em um estado onde ela não ocupará mais a CPU até que a operação "*v*" seja realizada sobre o mesmo semáforo. Para oferecer o comportamento periódico às tarefas, a abstração *Alarm* é então utilizada.

A abstração *Alarm* invoca periodicamente um objeto função (instância da classe *Handler* apresentada na figura 2), neste caso especializado para executar a operação "*v*" sobre o semáforo que "impede" a repetição do corpo do laço. A *thread* pode então, voltar a ganhar a CPU segundo os critérios do escalonador. Ao recuperar a execução, ela volta

ao ponto onde invocou *Periodic_Thread::wait_next*, obedecendo ao comportamento do laço escrito.

4. Ambiente RT-DVFS

Para implementação das heurísticas para RT-DVFS as heurísticas precisam estar cientes de eventos referentes às tarefas do sistema, como inicialização, término ou até mesmo alterações no conjunto de tarefas. Para que fosse possível atribuir ações a estes eventos, algumas modificações foram necessárias no modelo de *threads* periódicas do EPOS, como apresentado no diagrama de classes da figura 3.

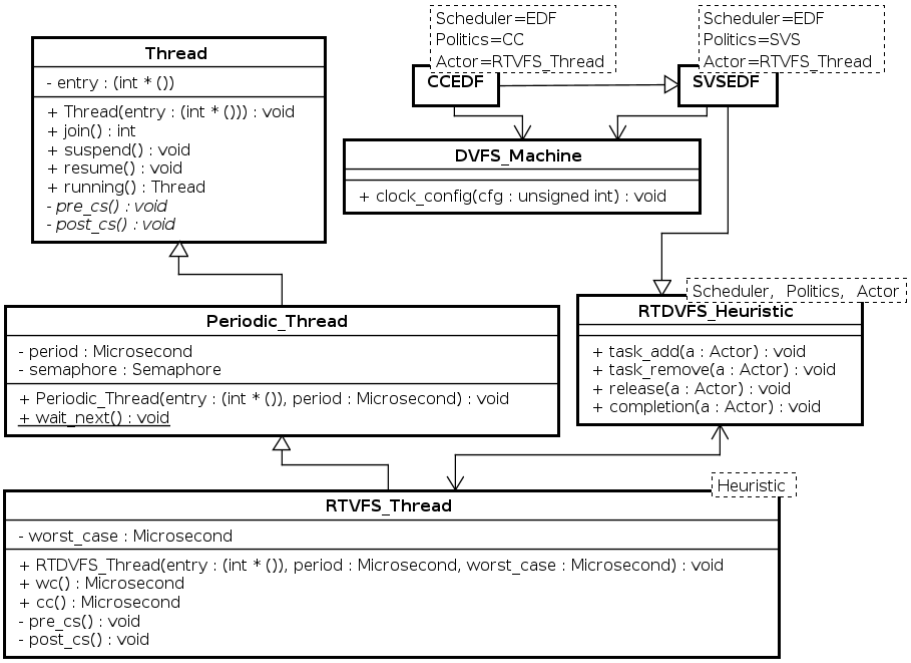

Figura 3. Diagrama de classes demonstrando as relações da *RTVFS_Thread*

4.1. Captura de Inicialização e Término de uma Tarefa

Os eventos de inicialização e término de tarefas são capturados através da reimplementação do método *wait_next* para a *RTDVS_Thread*. O diagrama de sequência da figura 4 mostra o momento em que estes eventos são reportados à heurística, anteriormente e posteriormente à aquisição de recurso no semáforo pertencente à *thread* periódica.

4.2. Captura de Modificações no Conjunto de Tarefas

A reação à alteração no conjunto de tarefas é dada a partir de duas maneiras: adição ou remoção de tarefas. Para capturar o evento de adição, em toda construção de uma nova *RTDVFS_Thread*, a classe *RTDVS_Heuristic* tem o método *task_add* invocado, sendo passada como parâmetro para sua execução, a instâcia de *RTDVFS_Thread* em ação. Analogamente, na destruição de uma *RTDVS_Thread*, o método *task_remove* é chamado.

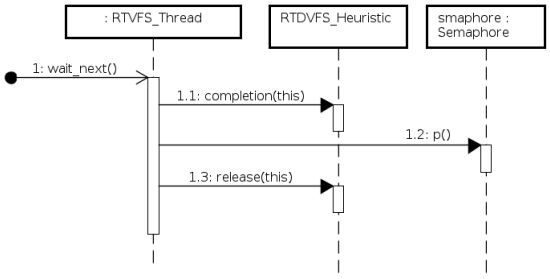

Figura 4. Diagrama de sequência demonstrando o comportamento do método *wait_next* da classe *RTDVFS_Thread*.

4.3. Adição de Heurísticas

Como estudo de caso, duas heurísticas RT-DVFS foram selecionadas para a implementação utilizando-se o suporte criado: *Static Voltage Scaling* e *Cycle-conserving* [Pillai e Shin 2001], ambas sendo utilizadas em conjunto com o escalonador EDF já presente no EPOS. A política *Static Voltage Scaling* se baseia em dados de pior caso do tempo de execução das tarefas, assim sendo, ela calcula estaticamente a configuração de frequência ideal para o pior caso de execução das tarefas. A política *Cycle-conserving* se baseia em dados capturados de forma *on-line*, ajustando a configuração de frequência e tensão do processador dinamicamente segundo o uso real que as tarefas promovem no sistema.

A implementação das heurísticas foram realizadas na forma de especialização de *templates* em C++. A escolha da heurística pode ser programada estaticamente, como uma configuração do sistema operacional, sendo que cada heurística é uma especialização de três parâmetros: *Scheduler* (neste caso EDF), *Politics* (a política a ser utilizada: *Static Voltage Scaling* ou *Cycle-conserving*) e *Actor* (a abstração que utiliza a heurística, neste trabalho *RTDVFS_Thread*).

5. Experimentos e Resultados

O ambiente experimental foi criado com base em um conjunto pré-definido de 3 tarefas periódicas, com 300, 400 e 500 milisegundos de período, de modo que estas possuíssem uma ocupação real configurável através de um laço com um número arbitrário de iterações. Como métrica para avaliação, utilizou-se a potência média de todo o sistema, composto de uma placa-mãe Gumstix Connex, que contém um PXA255 como módulo de processamento. Juntamente a esta placa, uma expansão chamada Gumstix HWU-ART foi utilizada para a comunicação com um computador pessoal para a transmissão de aplicações EPOS.

O gráfico apresentado na figura 5, mostra a potência média obtida para uma utilização real igual à utilização de pior caso, ou seja, o caso onde a utilização real por cada tarefa é igual à utilização referente ao pior tempo de computação passado como parâmetro para a construção do objeto *RTDVFS_Thread*. Observe que este gráfico é capaz de mostrar a relação entre potência média do sistema e as configurações apresentadas na tabela 1, já que neste cenário de pior caso, as heurísticas tendem a utilizar as configurações (normalizadas pela frequência mais alta possível de CPU) correspondentes

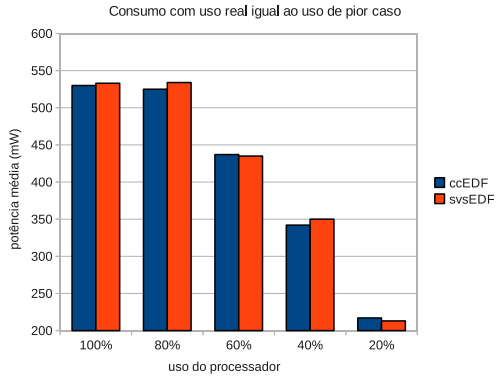

Figura 5. Gráfico da potência média com utilização real igual a de pior caso

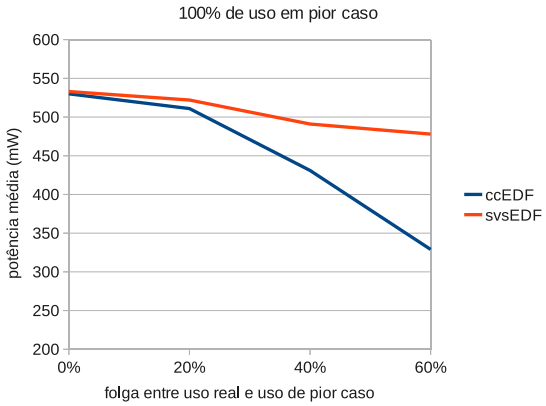

Figura 6. Gráfico da potência média com utilização real igual a de pior caso

à utilização do sistema. Por exemplo, se as tarefas promovem uma ocupação real de 0,6, o sistema possivelmente trabalhará a 298,6MHz (0,75 da maior frequência possível).

O gráfico apresentado na figura 6 mostra o comportamento das heurísticas com utilização de pior caso total, mas apresentando uma variação durante a execução na utilização real entre 20 e 100%. Isto significa que, mesmo as tarefas tendo como parâmetro um pior tempo de computação que causa 100% de utilização, na realidade as tarefa canônicas correspondentes realizam apenas uma porcentagem fixa de 20 a 100% do pior caso de computação. Como esperado, a heurística dinâmica tem maior aproveitamento energético, já que ela se beneficia de dados capturados dinamicamente.

6. Conclusões

A maior contribuição deste trabalho encontra-se na implementação de um ambiente RT-DVFS sobre o EPOS. Grande parte das implementações de ambientes RT-DVFS utilizam o sistema operacional Linux. O EPOS é um sistema operacional nativamente desenvolvido para a produção de sistemas embarcados, além de seguir a metodologia ADESD [Fröhlich 2001], um cenário diferenciado das outras implementações presentes na literatura sobre o tema.

Ainda, as novas abstrações que foram criadas no sistema, permitem facilmente a

criação de novas heurísticas. A classe *RTDVFS_Thread* é capaz de fornecer metadados referentes a execução de tarefas periódicas. Utilizando-a em conjunto com os eventos captados por *RTDVS_Heuristic*, basta-se estender esta classe para que uma nova política para economia de energia seja adicionada ao sistema. Vale salientar que as heurísticas continuam fracamente acopladas ao escalonador de tempo real do sistema operacional embarcado, como proposto na criação das mesmas [Pillai e Shin 2001].

Referências

Fröhlich, A. A. (2001). *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Intel (2004). *Intel PXA255 Processor: Developer's Manual*. Intel.

Lin, J. D., Song, W., and Cheng, A. M. (2010). Real-energy: a new framework and a case study to evaluate power-aware real-time scheduling algorithms. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '10, pages 153–158, New York, NY, USA. ACM.

Marcondes, H., Cancian, R., Stemmer, M., and Fröhlich, A. A. (2009). On design of flexible real time schedulers for embedded systems. In *International Symposium on Embedded and Pervasive Systems*, pages 382–387, Vancouver, Canada.

Pillai, P. and Shin, K. G. (2001). Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 89–102, New York, NY, USA. ACM.

Snowdown, D., Ruocco, S., and Heiser, G. (2005). Power management and dynamic voltage scaling. In *Proceedings of the 2005 Workshop on Power Aware Real-time Computing*, New Jersey, USA.

Zhu, Y. and Mueller, F. (2007). Exploiting synchronous and asynchronous dvs for feedback edf scheduling on an embedded platform. *ACM Trans. Embed. Comput. Syst.*, 7:3:1–3:26.

APÊNDICE B – Implementação em C++

Obedecendo o artigo 17 do *Regimento Interno Para Elaboração de Trabalhos de Conclusão de Curso de Ciência da Computação*, este apêndice contém o código fonte das classes implementadas em linguagem C++ para os fins deste trabalho.

B.1 Código-fonte C++ – Classe *RTDVFS_Thread*

B.1.1 Declarações

```
// EPOS RTDVFS Periodic Threads

#ifndef __rtdvfs_thread_h
#define __rtdvfs_thread_h

#include <periodic_thread.h>

__BEGIN_SYS

class RTDVFS_Thread: private Periodic_Thread{

public:

    typedef Simple_List<RTDVFS_Thread> List;
    List::Element link;

    RTDVFS_Thread(int (* entry)(),
        const Microsecond & period,
        const Microsecond & wc,
      int times = Alarm::INFINITE,
      const State & state = READY,
      unsigned int stack_size = STACK_SIZE);
```

```
    Microsecond wc();
    Microsecond cc();
    Microsecond period();

    static void wait_next() {
        self()->completed = true;
        Periodic_Thread::wait_next();
    }

    ~RTDVFS_Thread();

private:

    static Microsecond now();

    void pre_cs();
    void post_cs();

    static RTDVFS_Thread* self(){
        return reinterpret_cast<RTDVFS_Thread*>(Thread::running());
    }

private:

    Microsecond _wc;
    Microsecond _cc;
    Microsecond _last_entering;
    Microsecond _period;
    bool completed;
    bool first;
};

__END_SYS

#endif
```

B.1.2 Definições

```
#include <rtdvfs_thread.h>
#include <rtdvfs_heuristics.h>

__BEGIN_SYS
```

```
typedef RTDVFS_Heuristic<Traits<Thread>::Criterion,
                         Traits<RTDVFS_Thread>::Politics,
                         RTDVFS_Thread> Heuristic;

RTDVFS_Thread::RTDVFS_Thread(int (* entry)(),
    const Microsecond & period,
    const Microsecond & wc,
    int times,
    const State & state,
    unsigned int stack_size)
: Periodic_Thread(entry, period, times, state, stack_size),
  _wc(wc), _cc(0), _period(period), completed(false), first(false),
  link(this)
{
    Heuristic::task_add(this);
}

RTDVFS_Thread::Microsecond RTDVFS_Thread::wc(){
    return _wc;
}

RTDVFS_Thread::Microsecond RTDVFS_Thread::cc(){
    return _cc;
}

RTDVFS_Thread::Microsecond RTDVFS_Thread::period(){
    return _period;
}

RTDVFS_Thread::~RTDVFS_Thread(){
    Heuristic::task_remove(this);
}


RTDVFS_Thread::Microsecond RTDVFS_Thread::now(){
    return TSC::time_stamp()/(double)TSC::frequency()*1000000.0;
}

void RTDVFS_Thread::pre_cs(){
    _cc += now() - _last_entering;
    if(completed){
        Heuristic::completion(this);
        completed = false;
```

```
        first = true;
    }

}

void RTDVFS_Thread::post_cs(){
    _last_entering = now();
    if(first){
        _cc = 0;
        Heuristic::release(this);
        first = false;
    }

}

__END_SYS
```

B.2 Código-fonte C++ – Classe *RTDVFS_Heuristic* e derivadas

B.2.1 Declarações

```
#include <traits.h>
#include <scheduler.h>
#include <rtdvfs_thread.h>

__BEGIN_SYS

class CC;
class SVS;

template<class Scheduler, class Politics, class Actor>
class RTDVFS_Heuristic{
    public:
    static void init(){}
    static void task_add(Actor* a){}
    static void task_remove(Actor* a){}
    static void release(Actor* a){}
    static void completion(Actor* a){}
};

template<>
```

```
class RTDVFS_Heuristic<Scheduling_Criteria::EDF,SVS,RTDVFS_Thread>{

    protected:

    static RTDVFS_Thread::List task_set;

    static void optmize_frequency(double usage){

        unsigned int j;
        for(j = 0; j < DVFS_Machine::n_clock_configs; j++){
            if(usage <= DVFS_Machine::clock_configs[j].norm){
                DVFS_Machine::clock_config(j);
                return;
            }
        }

        DVFS_Machine::clock_config(DVFS_Machine::max_clock_config);
    }

    static void select_frequency(){

        RTDVFS_Thread::List::Iterator i;
        double u = 0;

        for(i = task_set.begin(); i != task_set.end(); i++){
            RTDVFS_Thread* t = i->object();
            u += t->wc()/(double)t->period();
        }

        optmize_frequency(u);
    }

    public:

    static void init(){

    }

    static void task_add(RTDVFS_Thread* a){
        task_set.insert(&a->link);
        select_frequency();
    }
```

```
    static void task_remove(RTDVFS_Thread* a){
        task_set.remove(&a->link);
        select_frequency();
    }

    static void release(RTDVFS_Thread* a){}
    static void completion(RTDVFS_Thread* a){}
};

template<>
class RTDVFS_Heuristic<Scheduling_Criteria::EDF,CC,RTDVFS_Thread>:
    public RTDVFS_Heuristic<Scheduling_Criteria::EDF,SVS,RTDVFS_Thread>{

    typedef RTDVFS_Heuristic<Scheduling_Criteria::EDF,SVS,RTDVFS_Thread>
        super;

    public:

    static void select_frequency(){

        RTDVFS_Thread::List::Iterator i;
        double u = 0;

        for(i = task_set.begin(); i != task_set.end(); i++){
            RTDVFS_Thread* t = i->object();
            if(t->cc() == 0)
                u += t->wc()/(double)t->period();
            else
                u += t->cc()/(double)t->period();
        }

        optmize_frequency(u);
    }

    static void release(Periodic_Thread* a){
        select_frequency();
    }

    static void completion(Periodic_Thread* a){
        select_frequency();
    }
};
```

__END_SYS

B.3 Código-fonte C++ – Classe *PXA255* (mediador de máquina)

B.3.1 Declarações

```
// OpenEPOS PXA255 Mediator Declarations

#ifndef __pxa255_h
#define __pxa255_h

#include <machine.h>
#include <cpu.h>
#include <mmu.h>
#include <tsc.h>
#include <system/memory_map.h>

__BEGIN_SYS

class PXA255: public Machine_Common
{
public:
    typedef CPU::Reg32 Reg32;
    typedef IO_Map<PXA255> IO;
    typedef void (int_handler)(unsigned int);
    typedef TSC::Hertz Hertz;

public:
    PXA255() {}

    static void smp_init(unsigned int n_cpus) {}
    static unsigned int n_cpus() { return 1; }
    static unsigned int cpu_id() { return 0; }
    static void smp_barrier(unsigned int n_cpus = 1) { }

    static int_handler * int_vector(unsigned int i) {
      return 0;
    }
    static void int_vector(unsigned int i, int_handler * h) {
    }
```

```
  template<typename Dev>
  static Dev * seize(const Type_Id & type, unsigned int unit) {
    return 0;
  }

  static void release(const Type_Id & type, unsigned int unit) {
  }

  static void panic() {
      CPU::int_disable();
  db<PXA255>(ERR) << "PANIC!\n";
  for(;;);
}

  static void reboot() {
      CPU::int_disable();
      db<PXA255>(ERR) << "Reboot not implemented.\n";
      panic();
  }

  static void poweroff();

  static int irq2int(int i) { return i; }
  static int int2irq(int i) { return i; }

  static void init();

  static unsigned int clock() { return Traits<Machine>::CLOCK; }

  enum{

      // CCLKCFG Register Values
      TURBO = 0x1,
      FCS   = 0x2,

      // Core Clock Configuration Register (CCCR) Fields
      CCCR_L  = 0,
      CCCR_M  = 5,
      CCCR_N  = 7,

      // Core Clock Configuration Register (CCCR) Values
      MEM_995 =   0x1,
```

```
        CPU_X1_MEM  =   0x1,
        CPU_X2_MEM  =   0x2,
        CPU_X4_MEM  =   0x3,

        TURBO_X15_CPU   =   0x3,
        TURBO_X2_CPU    =   0x4,
        TURBO_X3_CPU    =   0x6
    };

    typedef struct {
        Hertz cpu;
        Hertz pxbus;
        Hertz mem;
        Reg32 cccr_value;
        bool turbo;
        unsigned short dri; // SDRAM Refresh Interval
        bool div_mem;
        double norm;
    } Clock_Configuration;

    static const Clock_Configuration clock_configs[];
    static const unsigned int n_clock_configs;
    static const unsigned int max_clock_config;

    static void clock_config(unsigned int config);

    static void enter_turbo(){
        cclkcfg(cclkcfg() | FCS | TURBO);
    }

    static void exit_turbo(){
        cclkcfg(cclkcfg() & ~TURBO);
    }

    static void enter_fcs(){
        cclkcfg(cclkcfg() | (FCS & ~TURBO));
    }

    static void exit_fcs(){
        cclkcfg(cclkcfg() & ~FCS);
    }

private:
```

```
static void cclkcfg(Reg32 value){
    ASMV(   "mcr␣p14,␣0,␣%0,␣c6,␣c0,␣0\n"
            :: "r"(value) :
        );
}

static Reg32 cclkcfg(){
    Reg32 value = 0;
    ASMV(   "mrc␣p14,␣0,␣%0,␣c6,␣c0,␣0\n"
            : "=r"(value) ::
    );
    return value;
}

static void cccr(Reg32 value){
    CPU::out32(IO::CM_CCCR, value);
}

static Reg32 cccr(){
    return CPU::in32(IO::CM_CCCR);
}

static void mdrefr(Reg32 value){
    CPU::out32(IO::MC_MDREFR, value);
}

static Reg32 mdrefr(){
    return CPU::in32(IO::MC_MDREFR);
}

enum{
    SDRAM_REFR_TIME = 64,
    SDRAM_ROWS_NUM  = 8192,
    MDREFR_DRI_MASK = 0xFFF,
    MDREFR_DB2_MASK  = 0x24000
};

static unsigned short dri(){
    return (unsigned short)(mdrefr() & MDREFR_DRI_MASK);
}

static short dri(unsigned int membus_freq){
```

```
        return (membus_freq*SDRAM_REFR_TIME)/(SDRAM_ROWS_NUM*32);
    }

};

__END_SYS

#include "info.h"
#include "uart.h"
#include "rtc.h"
#include "timer.h"
#include "nic.h"

#endif
```

B.3.2 Definições

```
// OpenEPOS PXA255 Mediator Implementation

#include <machine.h>

__BEGIN_SYS

// Configurations MUST be ordered by the leading frequency!
const PXA255::Clock_Configuration PXA255::clock_configs[] = {

    {
        99500000,   // CPU clock
        50000000,   // PXBus clock
        99500000,   // SRAM clock
        MEM_995<<CCCR_L | CPU_X1_MEM<<CCCR_M | TURBO_X15_CPU<<CCCR_N,   // cccr values
        false,   // enter turbo mode
        24, // SDRAM refresh rate in us
        false, // use memory clock divisor
        0.25 // normalized CPU clock
    },

    {
        199100000,
        99500000,
        99500000,
        MEM_995<<CCCR_L | CPU_X2_MEM<<CCCR_M | TURBO_X15_CPU<<CCCR_N,
        false,
```

```
        24,
        false,
        0.50
    },

    {
        298500000,
        99500000,
        99500000,
        MEM_995<<CCCR_L | CPU_X2_MEM<<CCCR_M | TURBO_X15_CPU<<CCCR_N,
        true,
        24,
        false,
        0.75
    },

    {
        398100000,
        196000000,
        99500000,
        MEM_995<<CCCR_L | CPU_X4_MEM<<CCCR_M | TURBO_X15_CPU<<CCCR_N,
        false,
        24,
        false,
        1.00
    }

};

const unsigned int PXA255::n_clock_configs = 4;
const unsigned int PXA255::max_clock_config = 3;

void PXA255::clock_config(unsigned int config){
    cccr(clock_configs[config].cccr_value);
    if(!clock_configs[config].turbo){
        exit_turbo();
        enter_fcs();
    }
    else
        enter_turbo();
}

__END_SYS
```

Referências Bibliográficas

ARM. *ARM Architecture Reference Manual*. [S.l.], jul. 2005.

CHEN, J.-J.; KUO, C.-F. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In: *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 2007. (RTCSA '07), p. 28–38. ISBN 0-7695-2975-5. Disponível em: <http://dx.doi.org/10.1109/RTCSA.2007.37>.

FRöHLICH, A. A. *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. 200 p. ISBN 3-88457-400-0. Disponível em: <http://www.lisha.ufsc.br/pub/aoos.pdf>.

GNU. *GDB: The GNU Project Debbuger*. jun. 2011. Disponível em: <http://www.gnu-.org/software/gdb/>.

GUMSTIX. *Gumstix Open Source Products*. jun. 2011. Disponível em: <http://www-.gumstix.com/store/index.php>.

GUMSTIX. *Legacy Products*. jun. 2011. Disponível em: <http://www.gumstix.org-/hardware-design/legacy-products.html>.

IEEE. *IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture*. [S.l.], 10 2010. c1 -985 p.

INTEL. *Intel PXA255 Processor*: Developer's manual. [S.l.], jan. 2004.

KUMAR, G. S. A.; MANIMARAN, G. An intra-task dvs algorithm exploiting path probabilities for real-time systems. *SIGBED Rev.*, ACM, New York, NY, USA, v. 2, p. 7–10, April 2005. ISSN 1551-3688. Disponível em: <http://doi.acm.org/10.1145-/1121788.1121792>.

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

LIN, J. D.; SONG, W.; CHENG, A. M. Real-energy: a new framework and a case study to evaluate power-aware real-time scheduling algorithms. In: *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*. New York, NY, USA: ACM, 2010. (ISLPED '10), p. 153–158. ISBN 978-1-4503-0146-6. Disponível em: <http://doi.acm.org/10.1145/1840845.1840877>.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, ACM, New York, NY, USA, v. 20, p. 46–61, January 1973. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/321738.321743>.

MARCONDES, H. et al. On design of flexible real time schedulers for embedded systems. In: *International Symposium on Embedded and Pervasive Systems*. Vancouver, Canada: [s.n.], 2009. p. 382–387. ISBN 978-0-7695-3823-5.

PILLAI, P.; SHIN, K. G. Real-time dynamic voltage scaling for low-power embedded operating systems. In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2001. (SOSP '01), p. 89–102. ISBN 1-58113-389-8. Disponível em: <http://doi.acm.org/10.1145/502034.502044>.

QEMU PROJECT. *About - QEMU*. jun. 2011. Disponível em: <http://wiki.qemu.org-/Main_Page>.

RANGANATHAN, P. Recipe for efficiency: principles of power-aware computing. *Commun. ACM*, ACM, New York, NY, USA, v. 53, p. 60–67, April 2010. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1721654.1721673>.

SNOWDOWN, D.; RUOCCO, S.; HEISER, G. Power management and dynamic voltage scaling: Myths and facts. In: *Proceedings of the 2005 Workshop on Power Aware Real-time Computing*. New Jersey, USA: [s.n.], 2005.

STROUSTRUP, B. *The C++ Programming Language*. 3. ed. [S.l.]: Addison-Wesley, 1997.

TSAI, Y.-H.; WANG, K.; CHEN, J.-M. A deferred-workload-based inter-task dynamic voltage scaling algorithm for portable multimedia devices. In: *Proceedings of the 2007 international conference on Wireless communications and mobile computing*. New York, NY, USA: ACM, 2007. (IWCMC '07), p. 677–682. ISBN 978-1-59593-695-0. Disponível em: <http://doi.acm.org/10.1145/1280940.1281084>.

ZHU, Y.; MUELLER, F. Exploiting synchronous and asynchronous dvs for feedback edf scheduling on an embedded platform. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 7, p. 3:1–3:26, December 2007. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/1324969.1324972>.