Universidade Federal do Amazonas
Instituto de Computação

Inserção de Código DVFS-*Aware* em Sistemas de Tempo Real Críticos

Diego Quintana Pinheiro

Manaus - Amazonas
Setembro de 2015

Diego Quintana Pinheiro

Inserção de Código DVFS-*Aware* em Sistemas de Tempo Real Críticos

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas, como requisito parcial para obtenção do Título de Mestre em Informática.

Orientador: Prof. Dr. Raimundo da Silva Barreto.

Manaus - Amazonas

Fevereiro de 2015

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Pinheiro, Diego Quintana
P654i Inserção de código DVFS-Aware em sistemas de tempo real críticos / Diego Quintana Pinheiro. 2015
123 f.: il. color; 31 cm.

Orientador: Raimundo da Silva Barreto
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Sistemas de Tempo Real. 2. Baixo Consumo de Energia. 3. DVFS. 4. DVFS Inter-Tarefa. 5. Transformação de Código. I. Barreto, Raimundo da Silva II. Universidade Federal do Amazonas III. Título

Agradecimentos

Primeiramente, agradeço a Deus pela força que me deu e continua dando para enfrentar os desafios diários. Agradeço também por todas as dificuldades que vivenciei e serviram para o meu crescimento.

Aos meus pais, minha irmã Thaís, a todos os meus tios, em especial Irapuã e Lúcia, e aos meus primos, Leninha, Raul, e toda a minha família, só tenho a agradecer pela força e incentivo. Pelas conversas tranquilizadoras e também de cobrança. Pelos ensinamentos que me deram desde a infância. Sei que nunca faltará lições novas para aprender, assim como compartilhar alegrias com vocês.

Agradeço a minha namorada, Gethe, pela alegria que sempre me deu. Compreensão, apoio e carinho também foram alguns dos principais pontos que tive com ela ao longo do meu mestrado.

Aos grandes amigos da escola, Douglas, Alison, Paulo, Sabrina, Rodrigo, Gabriel, da faculdade, Letícia, Fábio, Manderson e Felipe, e do trabalho, especialmente Marcelo, Emory, Guilherme e Rodolfo. Por mais que eu estivesse distante em muitos momentos, vocês compreenderam e me apoiaram. Sempre vi nossas conversas, encontros, brincadeiras como uma forma incentivadora para construção desta dissertação.

Quero agradecer também aos meus amigos do grupo de pesquisa, Herberth, Rawlinson, Valentin e Daniela. Nunca fui presente no laboratório, mas sempre tive de vocês apoio, esclarecimento e ótimas sugestões para o crescimento do meu trabalho. Principalmente o Herberth que muitas vezes interrompi para conversar e trocar ideias sobre as dificuldades da minha pesquisa.

Por fim, dedico meu principal agradecimento ao professor Barreto. Esta dissertação só foi possível graças a sua paciência e compreensão. O senhor acreditou em mim mesmo com as ausências, e sempre me incentivou a seguir em frente e continuar a pesquisa. Seu exemplo como pessoa também foi um dos fatores diferenciais nesta minha caminhada. Obrigado pelas conversas esclarecedoras.

A todos os que listei e também aqueles que contribuíram direta ou indiretamente para este trabalho: muito obrigado!

"*Not all those who wander are lost*"

(J. R. R. Tolkien)

Resumo

Desempenho e consumo de energia são variáveis diretamente proporcionais. Para aumentar o desempenho, é necessário também aumentar o número de instruções por segundo a serem executadas, ou seja, alterar a frequência do processador. Quanto maior for este valor, também será o consumo de energia. Do mesmo modo, reduzir o consumo de energia implica diminuir o número de instruções a serem executadas e, logo, o desempenho. Explorar a relação entre desempenho e consumo de energia é a ideia base da técnica de escalonamento dinâmico de tensão e frequência DVFS (do inglês *Dynamic Voltage and Frequency Scaling*).

Em sistemas de tempo real críticos, aplicar a técnica DVFS não é uma tarefa trivial. Estes sistemas associam a execução de uma tarefa a um limite temporal, de modo que, se este valor não for respeitado, devido à redução do desempenho, falhas graves podem ocorrer ao sistema. Assim, esta dissertação tem como objetivo unir duas abordagens da técnica DVFS em sistemas de tempo real críticos: uma intra e outra inter-tarefas.

A abordagem intra-tarefa procura analisar o fluxo de execução de uma tarefa e identificar pontos onde é possível inserir instruções para troca de frequência e tensão, quando a execução de uma tarefa se distanciar do pior caso.

Já a abordagem inter-tarefas, é responsável por: analisar o tempo de espera na execução de uma tarefa devido às interferências (preempções, compartilhamento de recursos), verificar a escalonabilidade do sistema e determinar um conjunto de frequências iniciais ótimas em ambientes de múltiplas tarefas.

O resultado deste estudo é a geração de um novo código com funcionalidade igual ao de entrada, porém com instruções de troca de frequência e tensão, consideradas as interferências que uma tarefa possa sofrer. Além disso, resultados experimentais mostram como não só foi possível reduzir o consumo de energia, mas também respeitar os limites temporais das tarefas em questão.

Palavras-Chave: Sistemas Embarcados, Sistemas de Tempo Real, Baixo Consumo de Energia, DVFS Intra-Tarefa, DVFS Inter-Tarefa, Transformação de Código.

Abstract

Performance and energy consumption are directly related. To increase performance, the number of instructions per second to be executed must also be increased, in other words, processor frequency must be changed. The higher this value is, higher energy consumption also has to be. Likewise, by decreasing the number of instructions to be executed, energy consumption and performance are also reduced. So, exploring performance and energy relation is the key idea behind Dynamic Voltage and Frequency Scaling – DVFS, technique.

Applying DVFS in real time systems is not a trivial task. These system's tasks are bounded to timing constraints in such a way that, if decreasing performance does not guarantee constraints, the system may totally fail. Thus, this work aims to gather two DVFS approaches in real time systems: intra and inter-tasks.

The intra-task analyzes execution flow of a task and identify where the new instructions can be inserted to change supply voltage and frequency when the worst case path is not followed. On the other hand, the inter-task approach analyzes how long a task will wait due to interferences (e.g. preemption, shared resources), verifies system schedulability and defines a set of initial optimum frequencies in multi-task environment.

The result is a new code with the same functionality as the original one, however with instructions to change voltage and frequency when taking into account a task interferences. Moreover, the experimental results show not only energy consumption was reduced, but also timing constraints were satisfied.

Keywords: Embedded Systems, Real Time Systems, Low Power Consumption, DVFS Intra-Task, DVFS Inter-Task, Code Generation.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

CFG – *Control Flow Graph*
DM – *Deadline Monotonic*
DVFS – *Dynamic Voltage and Frequency Scaling*
EDF – *Earliest Deadline First*
PCP – *Priority Ceiling Protocol*
PHP – *Priority Heritage Protocol*
RM – *Rate Monotonic*
RWCEC – *Remaining Worst-Case Execution Cycles*
SEC – *Saved Execution Cycles*
WCEC – *Worst-Case Execution Cycles*
WCEP – *Worst-Case Execution Path*
WCET – *Worst-Case Execution Time*

Capítulo 1

Introdução

Relógios inteligentes, aparelhos para monitoramento de pacientes em hospitais e controle de tráfego aéreo são exemplos de sistemas voltados a tarefas específicas. Por terem a aplicação restrita a alguns cenários, esses sistemas permitem otimizações tanto em nível de *hardware* quanto *software*, diferentemente de computadores de propósito geral.

Nestes tipos de sistemas, é possível, por exemplo, reduzir o tamanho de um dispositivo, utilizar baixos recursos como memória e processamento, dispensar o uso de um sistema operacional, entre outros. Estas são algumas características que definem os sistemas embarcados.

Um fato interessante relacionado aos sistemas supracitados é quanto a eficiência energética, pois trabalhar com baixo consumo de energia pode influenciar diretamente na arquitetura dos dispositivos. Por exemplo, redução do tamanho, menor exigência quanto ao resfriamento dos componentes, aumento do tempo de vida e autonomia. Porém, aumentar o desempenho nos sistemas embarcados, mesmo com as limitações intrínsecas a eles, é uma das principais exigências do mercado.

Desempenho e consumo de energia são variáveis diretamente proporcionais. Para aumentar o desempenho, é necessário também aumentar o número de instruções por segundo a serem executadas, ou seja, consiste em alterar a frequência do processador. Quanto maior for este valor, também será o consumo de energia

Do mesmo modo, reduzir o consumo de energia implica diminuir o número de instruções a serem executadas e, logo, o desempenho. Manter o equilíbrio entre estes dois elementos não é uma tarefa trivial. Ao analisar estas informações nos sistemas embarcados, diversos componentes podem ser os responsáveis pelo alto consumo de energia, todavia se tem os processadores como os principais determinantes desta análise (Pering *et al.*, 1998).

É responsabilidade dos processadores executar cada instrução das tarefas de um

sistema. Se houver a aplicação de uma alta frequência, menor será a redução de energia. Segundo Shin *et al.* (2001), a energia consumida nos processadores é proporcional ao desempenho, conforme a seguinte equação reduzida:

$$E \propto C * N_{ciclos} * V_{dd}^2 \tag{1.1}$$

onde C é a capacitância do circuito, N_{ciclos} corresponde ao número de ciclos executados, e V_{dd} representa a tensão de alimentação. Portanto, quanto maior for a tensão de alimentação, maior será o consumo de energia.

A relação quadrática entre energia e tensão proporciona otimizações a partir da utilização de tensões e frequências menores, uma vez que um conjunto discreto destes valores está disponível aos usuários nos processadores atuais. Estas informações podem ser utilizadas para alterar o desempenho conforme a necessidade do sistema. Logo, quando uma tarefa estiver em execução e demandar um desempenho menor que o atual, otimizações no consumo de energia serão obtidas se for possível diminuir a frequência e tensão. Esta é a ideia base da técnica de escalonamento dinâmico de tensão e frequência DVFS – do inglês *Dynamic Voltage and Frequency Scaling* (Weiser *et al.*, 1994).

Um exemplo da aplicação da DVFS consiste na execução de um processo com restrição temporal de $25ms$ à $50MHz$ e $5V$. Se uma tarefa exigir o consumo de $5 * 10^5$ ciclos, o tempo de execução será de $10ms$ já que, por segundo, são executados $50 * 10^6$ ciclos, então haverá uma ociosidade de $15ms$. Todavia, se a tensão e a frequência forem alteradas, respectivamente, para o valor ideal de $2V$ e $20MHz$, o término da tarefa ocorrerá exatamente na restrição temporal imposta de $25ms$. Desta forma, não só a restrição temporal foi cumprida, como também há uma redução de energia de 84% (Shin *et al.*, 2001).

1.1 Descrição do Problema

A utilização da técnica DVFS nos sistemas embarcados apresenta uma alternativa para a otimização do consumo de energia. Em contrapartida, se deve avaliar cautelosamente o quanto as tarefas do sistema serão afetadas.

Se o sistema for monotarefa, no qual ocorre apenas a execução de uma tarefa por vez, reduzir o desempenho não será um problema crítico já que o processador está dedicado à

tarefa em execução.

Entretanto, nos ambientes de múltiplas tarefas, os processadores estão encarregados da execução de diversas delas simultaneamente. A mudança no desempenho proposto pela técnica DVFS pode influenciar negativamente no tempo de término de uma tarefa e prejudicar outras partes do sistema. Por exemplo, em sistemas onde há compartilhamento de recursos, quanto maior for o tempo de bloqueio de uma região crítica, maior também será o tempo para a finalização de outras tarefas que dependam do mesmo recurso. Ao considerar ainda a aplicação da DVFS em sistemas embarcados com restrições temporais, também conhecidos como *sistemas de tempo real*, o tempo de término de uma tarefa é essencial para a confiabilidade do sistema (Farines *et al.*, 2000).

Em sistemas de tempo real críticos, considera-se uma falha grave à execução se as restrições temporais definidas em tempo de projeto não forem respeitadas. Contudo, um tempo ocioso é registrado pelo processador caso o término de uma tarefa ocorra antes da meta temporal. Além disso, ainda se registra um consumo elevado de energia no tempo ocioso, pois a frequência e tensão permanecem as mesmas utilizadas desde o início da execução da tarefa.

O estudo com base no tempo ocioso de uma tarefa é um dos principais desafios da aplicação da técnica DVFS em sistemas de tempo real críticos, devido às restrições temporais de cada tarefa. Ao considerar ambientes de múltiplas tarefas, a utilização da DVFS se torna ainda mais complicada, pois preempções, relações de precedências e compartilhamento de recursos devem ser considerados antes de alterar o desempenho do sistema.

As seguintes perguntas definem o problema abordado por esta dissertação: é possível identificar pontos onde a troca dinâmica de tensão e frequência pode ser realizada? Há como transformar o código de uma tarefa sem restrições quanto ao consumo de energia (DVFS-*Unaware*) em outro onde pontos para troca de tensão e frequência (DVFS-*Aware*) são identificados, e a restrição temporal respeitada – sem influenciar na execução das demais tarefas?

1.2 Contexto

O contexto deste trabalho é o desenvolvimento de uma abordagem para aplicação do escalonamento dinâmico de tensão e frequência em sistemas de tempo real críticos, a partir da análise estática das tarefas de um sistema. Este contexto é estendido ainda aos sistemas cuja execução de múltiplas tarefas é possível.

1.3 Motivação

A crescente demanda por dispositivos embarcados como *smartphones*, *tablets*, *wearable computers*, e a concorrência neste mercado apresentaram um modelo de negócio, no qual o balanço entre desempenho e autonomia é cada vez mais essencial na compra de um novo produto. Mesmo com a popularização destes sistemas, ainda é baixa a preocupação em se desenvolver soluções com eficiência energética.

Quando restrições temporais são impostas às aplicações de um sistema, como nos sistemas de tempo real críticos, qualquer redução no desempenho para otimizar o consumo de energia se torna uma tarefa complexa.

Assim, apresentar um estudo onde haja preocupação com: os limites temporais das tarefas; preempções; variáveis intrínsecas a cada ambiente (como tensões e frequências disponíveis); e otimizações no consumo de energia, como uma forma de não só contribuir para o crescimento do desenvolvimento de pesquisas e soluções nesta área, como também de automatizar a etapa de verificação e melhoria do consumo de energia a fim de reduzir o tempo de codificação de novos programas.

1.4 Objetivos

O objetivo principal desta dissertação consiste em propor uma metodologia de transformação de código, onde um código sem otimizações de energia possa ser transformado em outro com eficiência energética.

Tem-se ainda como objetivos específicos:

- Analisar um código estaticamente e abstrair o comportamento de uma tarefa em um grafo de fluxo de controle;

- Identificar pontos onde é possível alterar a frequência e tensão do processador durante a execução de uma tarefa;
- Verificar experimentalmente se a metodologia proposta é capaz de otimizar o consumo de energia de tarefas no contexto de sistemas de tempo real críticos;

1.5 Metodologia

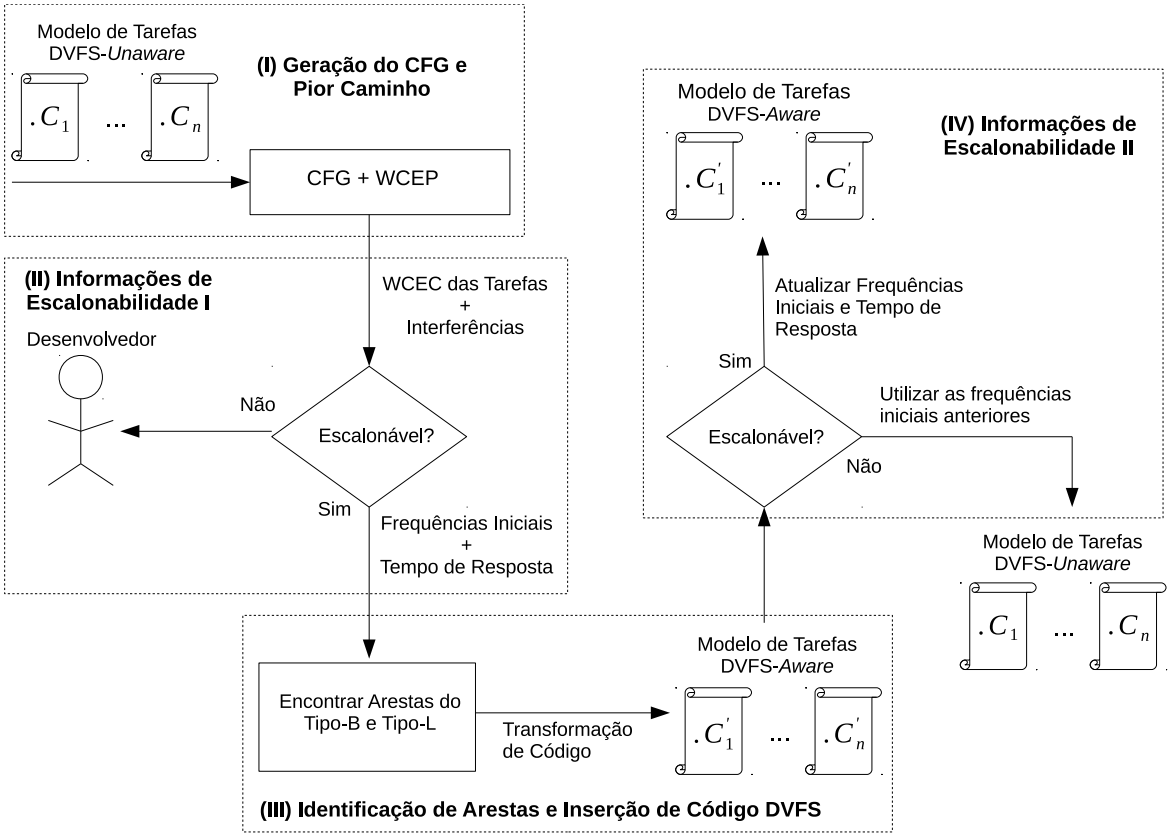


Figura 1.1: Visão geral da metodologia proposta.

A Figura 1.1 apresenta a visão geral da metodologia proposta, a qual pode ser dividida em quatro etapas:

- **Geração do CFG e Pior Caminho:** esta etapa visa à obtenção do grafo de fluxo de controle a partir da análise estática do comportamento de uma tarefa.

Além disso, utiliza informações do código intermediário para calcular o custo computacional de cada nó do grafo e extrair o pior caminho do fluxo de execução de uma tarefa;

- **Informações de Escalonabilidade I:** consiste em utilizar o trabalho proposto por (Valentin, 2010) para verificar a escalonabilidade do modelo de tarefas, e determinar as frequências iniciais e tempos de resposta das tarefas utilizadas na primeira etapa;

- **Identificação de Arestas e Inserção de Código DVFS:** a terceira etapa relaciona-se à identificação de arestas no grafo de fluxo de controle no qual é possível trocar a tensão e frequência do processador, além de calcular qual valor deveria ser aplicado, de acordo com o trabalho apresentado em (Shin *et al.*, 2001). Em seguida, com as arestas identificadas, transforma-se o código original DVFS-*Unaware* em DVFS-*Aware* a partir da inserção de novas instruções em alto nível, as quais se encarregam de calcular e aplicar a nova frequência;

- **Informações de Escalonabilidade II:** a última etapa procura garantir que a inserção do código DVFS-*Aware* não alterou a escalonabilidade do modelo de tarefas, assim como atualizar as frequências iniciais e tempos de resposta, caso tenham sido alterados pela terceira etapa.

1.6 Organização do Trabalho

Este trabalho está dividido em mais cinco capítulos. O segundo apresenta conceitos fundamentais para a compreensão dos assuntos abordados. No terceiro, abordam-se estudos relacionados à utilização da técnica DVFS com três diferentes abordagens. O quarto capítulo contém o método proposto desta dissertação. Por sua vez, o quinto expõe a descrição detalhada do ambiente de simulação e dos resultados experimentais. Por fim, o sexto apresenta as conclusões gerais e contribuições, além dos pontos limitantes e trabalhos futuros.

Capítulo 2

Conceitos e Definições

Os sistemas embarcados possuem um processamento mais restrito, ao contrário dos computadores de propósito geral, em razão de normalmente estarem direcionados a tarefas específicas. Diante disso, durante o projeto, pontos como tamanho, desempenho e consumo de energia devem ser analisados a fim de se ter o melhor custo/benefício.

Máquinas de lavar, geladeiras, portão elétrico, são exemplos simples de sistemas embarcados. Em sentido diverso, outros casos que envolvem maior complexidade vão desde sistemas militares de defesa ao controle do tráfego aéreo. Por sua vez, algumas aplicações como o monitoramento de pacientes em hospitais e o controle em plantas industriais exigem ainda restrições mais rigorosas, como o tempo (Farines *et al.*, 2000), já que uma falha pode resultar em consequências graves para as pessoas ou ambientes em questão.

Assim, é importante assegurar não só o correto funcionamento de um sistema, como também o cumprimento de todas as restrições temporais eventualmente impostas – no caso de sistemas de tempo real.

Com base nas características dos sistemas embarcados e de tempo real, este capítulo apresenta os principais conceitos necessários para a compreensão desta dissertação.

Primeiramente, abordam-se as definições de sistemas de tempo real e os diversos aspectos temporais intrínsecos a eles; em seguida, apresentam-se testes de como garantir a escalonabilidade de um modelo de tarefas com base nas respectivas metas temporais; por fim, conceitos da técnica de escalonamento dinâmico de tensão e frequência, e estudos sobre o comportamento de uma tarefa, a partir da análise estática de código, são discutidos nas duas últimas seções.

2.1 Sistemas de Tempo Real

Sistemas de tempo real podem ser definidos como sendo "*um sistema computacional que deve reagir a estímulos oriundos do seu ambiente em prazos específicos*" (Farines *et al.*, 2000).

Os estímulos podem ser representados por alguma ação externa de entrada ou saída, ou mesmo algum tempo pré-definido para a ativação de uma tarefa. As reações aos estímulos devem fornecer garantias da integridade dos resultados obtidos e também da correção temporal.

Classificam-se os sistemas de tempo real de duas formas:

- **Sistemas de Tempo Real Não Críticos** (*Soft Real Time Systems*): são sistemas que fazem o melhor possível para atingir as restrições temporais, porém podem continuar a execução normalmente caso uma tarefa não cumpra o tempo pré-definido, ou seja, a falha temporal não resulta em uma consequência grave. Exemplos: sistemas bancários, reserva de passagens aéreas e aplicações multimídias;

- **Sistemas de Tempo Real Críticos** (*Hard Real Time Systems*): são sistemas que devem cumprir as metas temporais estipuladas. Por exemplo, o controle de plantas industriais nucleares, o sistema depende de uma resposta dentro de um tempo específico. Caso contrário consequências graves podem acontecer aos funcionários da indústria ou mesmo a eventuais moradores de locais próximos. Outro agravante nos sistemas críticos é o desempenho, pois a sua redução influenciará no tempo de término de uma tarefa, visto que a relação entre tempo e desempenho é inversamente proporcional.

Tanto em sistemas de tempo real não críticos quanto críticos, as tarefas são as menores unidades de processamento sequencial que concorrem sobre um ou mais recursos computacionais (Farines *et al.*, 2000). Elas podem ser de três tipos de acordo, com o período de ativação:

- **Periódicas**: a ativação de uma tarefa ocorre sempre a um intervalo de tempo fixo, denominado período;

- **Aperiódicas ou Assíncronas**: a ativação de uma tarefa é dependente de um evento interno ou externo, por exemplo, alarme do carro. Elas podem ainda ocorrer em rajadas caso o evento que as ative ocorra consecutivamente;

- **Esporádicas**: similares às aperiódicas, exceto por uma restrição a mais: para ocorrer a ativação de uma tarefa esporádica, é necessário esperar um intervalo mínimo de tempo entre duas execuções da mesma tarefa. Elas também podem estar associadas a períodos.

A classificação das tarefas está diretamente relacionada à variável tempo. De fato, em sistemas de tempo real críticos, o cumprimento da restrição temporal de cada tarefa é o principal fator para garantir a correção temporal do sistema. Ao analisar uma tarefa quanto ao comportamento temporal, ressaltam-se alguns conceitos (veja Figura 2.1):

- **Tempo de Execução**: tempo necessário para execução completa de uma tarefa;

- **Tempo de Chegada**: momento em que o escalonador toma conhecimento de uma ativação da tarefa. O tempo de chegada é igual ao período de ativação em tarefas periódicas. No caso das não periódicas, o tempo de chegada é igual ao tempo em que uma tarefa ocupa o processador pela primeira vez durante a mesma ativação;

- **Tempo de Início**: instante de início do processamento de uma tarefa em uma ativação;

- **Tempo de Término**: momento em que a tarefa completa a execução;

- **Meta Temporal** (*Deadline*): valor máximo que o tempo de execução pode assumir;

- **Tempo Ocioso**: tempo não utilizado pela tarefa dentro da restrição temporal definida. Portanto, é igual à diferença entre a meta temporal e o tempo de execução.

- **Tempo de Liberação** (*release jitter* ou *jitter*): diferença entre o tempo de chegada e o de início da execução, ou seja, é o tempo necessário ao escalonador para liberar a execução inicial de uma tarefa em uma ativação. Ressalta-se que o *jitter* é uma medida de tempo, não uma tarefa, de modo que não pode interromper a execução de uma tarefa tampouco ser interrompido.

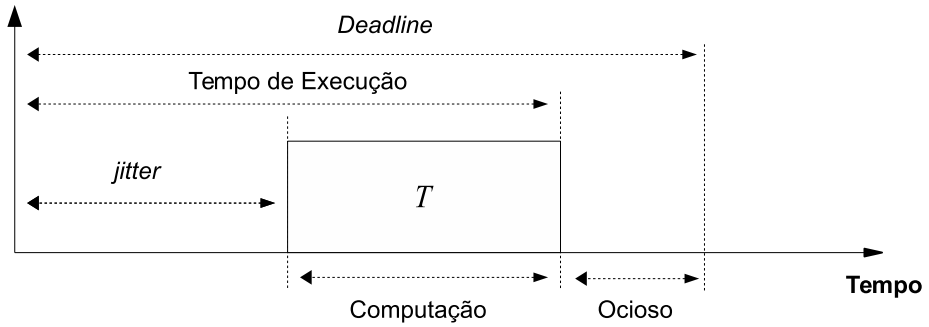


Figura 2.1: Comportamento temporal de uma tarefa.

Outro conceito ainda empregado às tarefas é o **fator de utilização**. Em outras palavras, o quanto uma tarefa i utiliza do processador. Esta métrica é definida por:

$$U_i = \frac{C_i}{P_i}$$

onde U_i é o fator de utilização da tarefa T_i no processador, C_i é o tempo de computação e P_i é o período da tarefa i.

As tarefas ainda podem respeitar uma ordem de execução, a qual é definida por um escalonador. Este é dito ser **preemptivo** se aceitar interrupções de uma tarefa por outras de maior prioridade, como na Figura 2.2 onde T_2 tem prioridade maior que T_1; ou **não-preemptivo**, caso contrário.

As prioridades, contudo, são determinadas em tempo de projeto – análise estática, ou em tempo de execução – análise dinâmica. Caso os valores calculados sejam aplicados em tempo de projeto, tem-se um escalonador *off-line*, senão *on-line*.

2.2 Teste de Escalonabilidade

Escalonador é o componente do sistema operacional responsável por decidir qual tarefa executará quando houver mais de uma pronta para execução.

A decisão da próxima tarefa a ser enviada ao processador se baseia em algoritmos de escalonamento, os quais associam e ordenam a lista de tarefas por prioridades.

Em sistemas de tempo real, o *deadline* de uma tarefa é outra variável a ser considerada, pois é necessário verificar se a ordem de execução definida é escalonável ou não.

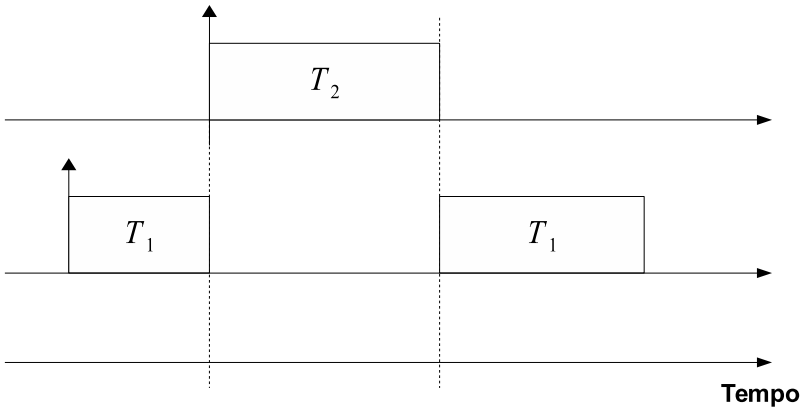


Figura 2.2: Preempção entre tarefas.

O teste de escalonabilidade consiste de uma prova de correção do conjunto de tarefas dado como entrada para um escalonador, o qual propõe verificar se este conjunto terá todos os *deadlines* satisfeitos quando ordenado por um algoritmo de escalonamento.

Kopetz (1992) apresenta três tipos de testes: exatos, suficientes e necessários. Testes **exatos** identificam conjuntos escalonáveis e não escalonáveis, porém em vários problemas são impraticáveis. Já os **suficientes**, descartam conjuntos que possam ser escalonáveis, todavia, se um conjunto de tarefas for aceito neste teste, então ele é escalonável. Os **necessários** garantem apenas que, se um conjunto não for aceito neste teste, então ele certamente não é escalonável. No teste de escalonabilidade ainda é necessário verificar a relação de dependência entre as tarefas.

2.2.1 Tarefas independentes

Um dos algoritmos bastante comuns na literatura referente ao tratamento de tarefas independentes em tempo de projeto é o *Rate Monotonic* – RM (Farines *et al.*, 2000).

Este algoritmo utiliza o conceito de prioridades fixas baseadas na periodicidade de uma tarefa. Assim, quanto menor for o período da tarefa, maior é a prioridade. As principais características que definem este modelo são:

- As tarefas necessitam ser periódicas e independentes;
- O *deadline* de cada tarefa coincide com o período da mesma;

- O tempo de computação de uma tarefa é conhecido e constante;
- O tempo de troca de execução entre tarefas é zero.

Com base nos dados acima, o teste de escalonabilidade do RM pode ser definido a partir da utilização de uma tarefa, ou seja, a taxa de ocupação da mesma no processador. Assim, para um conjunto de n tarefas, define-se um **teste suficiente** de escalonabilidade a partir da Equação (2.1).

$$U = \sum_{i=0}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1) \tag{2.1}$$

RM é considerado ótimo na classe de problemas a qual é aplicado. Em outras palavras, considerando-se um modelo de tarefas[1] similar, se outro algoritmo da mesma classe afirmar que o conjunto de tarefas é escalonável, então este conjunto também o é pelo RM. No entanto, considerar um conjunto de tarefas independentes e com o *deadline* igual ao período não representa bem os problemas encontrados na prática, visto que precedência entre tarefas e compartilhamento de recursos não são considerados. Assim, este teste de escalonabilidade não poderá informar com precisão se um conjunto de tarefas é escalonável ou não na prática.

2.2.2 Tarefas Dependentes

Relações de Precedência

Farines *et al.* (2000) comentam que, em muitos modelos de tarefa nos sistemas de tempo real, definem-se relações de precedência entre as tarefas.

Por exemplo, seja uma tarefa T_j precedida por T_i, T_j só poderá executar após o término de T_i – representado por (T_i -> T_j). Por oportuno, vale mencionar que a precedência não necessariamente está ligada apenas ao término de uma tarefa, uma vez que ela pode ser também expressa como a dependência de informações entre tarefas, a exemplo da troca de mensagens em uma rede.

O principal problema relacionado a precedência entre tarefas está na interferência que a execução de uma influenciará na outra, haja vista que as tarefas podem ser preemptadas

[1] Modelos de tarefas é a entrada para um problema de escalonamento. Nele definem-se as restrições temporais, relações de precedência e exclusão mútua entre as tarefas.

por outras mais prioritárias. Assim, tem-se um atraso no término da tarefa preemptada e inicialização da tarefa dependente.

No caso da escalonabilidade de um conjunto de tarefas, necessita-se avaliar qual a **interferência** máxima sofrida por uma tarefa devido às relações de precedência. Esta análise utiliza o conceito de **tempo de resposta** máximo de uma tarefa – diferença entre o tempo de término da execução e o tempo de chegada de uma tarefa.

Assume-se um conjunto de tarefas periódicas cujas ativações sempre ocorrerão no início de cada período, primeiramente para simplificar o teste de escalonabilidade. Deste modo, toda tarefa deve respeitar a condição (2.2).

$$R_i \leq D_i, \forall i \in T \tag{2.2}$$

onde T corresponde ao conjunto de tarefas, R_i ao tempo de resposta máximo e D_i deadline da tarefa i (Farines *et al.*, 2000). A Equação (2.3) define o tempo de resposta.

$$R_i = C_i + \sum_{j \in hp(i)} I_j \tag{2.3}$$

Para C_i é o tempo de computação da tarefa i e o $\sum_{j \in hp(i)} I_j$ como a interferência máxima que a tarefa i pode sofrer das tarefas j de maior prioridade no tempo de resposta R_i. Já a interferência I_j é definida na Equação (2.4).

$$I_j = \left\lceil \frac{R_i}{P_j} \right\rceil * C_j \tag{2.4}$$

onde P_j é o período da tarefa j, $\lceil \frac{R_i}{P_j} \rceil$ representa o número de liberações de j em R_i e C_j o tempo de computação da tarefa j. Isto permite reescrever a Equação (2.3).

$$R_i = C_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i}{P_j} \right\rceil * C_j \right) \tag{2.5}$$

Como R_i aparece em ambos os lados da equação, define-se um método iterativo[2]para se obter o valor do tempo de resposta R_i onde n é o número de iterações. Definido na Equação (2.6).

[2]Métodos iterativos geram uma sequência de valores aproximados a convergirem para um valor melhor conforme o número de iterações. O término deste método é definido por um critério de parada o

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i^n}{P_j} \right\rceil * C_j \right) \tag{2.6}$$

A Equação (2.6) encerra o cálculo do teste de escalonabilidade de um determinado conjunto de tarefas periódicas, onde o *jitter* corresponde sempre ao início do período.

Todavia, o *jitter* pode ser variável se a inicialização de uma tarefa depender do término de outra, de acordo com o modelo de tarefas, ou o sistema possuir um escalonador ativo por tempo, o qual só verificará a liberação de uma tarefa quando estiver em estado de ativação.

período ocupado é outro conceito na análise de relações de precedência, o qual considera a variação do *jitter*. *"Um período ocupado corresponde a uma janela de tempo W_i onde ocorre a execução contínua de tarefas com prioridade maior ou igual à da tarefa i...com o pressuposto que todas as tarefas de prioridades maiores que i estão na fila de prontos"* (Farines *et al.*, 2000). O número de ocorrências de uma tarefa j nesse intervalo de tempo, então, é definido por $\lceil \frac{W_i}{P_j} \rceil$. Entretanto, a tarefa j está sujeita a um *jitter* J_j – considera-se o valor máximo das variações possíveis na liberação de uma tarefa, logo o número de ativações de j que interferem em i é dado por $\lceil \frac{W_i + J_i}{P_j} \rceil$, e o cálculo de W_i pela Equação (2.7).

$$W_i^{n+1} = C_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{W_i^n + J_i}{P_j} \right\rceil * C_j \right) \tag{2.7}$$

A Equação (2.7) define o tempo que uma tarefa i precisa para terminar a execução a partir da liberação pelo escalonador com base nas interferências sofridas, devido a relação de precedência entre as tarefas. Porém, o valor de W_i não deve ser considerado como o tempo de resposta R_i da tarefa i, já que ele é determinado a partir do instante da liberação da tarefa. É necessário adicionar ainda o atraso máximo – J_i, que i pode sofrer na liberação. Define-se, então, o tempo de resposta máximo de uma tarefa i e a restrição de escalonabilidade como:

---

qual afirma que a partir de um certo ponto, a solução não convergirá para um valor aceitável. No caso do cálculo de R_i, define-se $R_i^0 = C_i$ como ponto inicial e $R_i^{n+1} = R_i^n$ a aproximação ideal. O método não é válido caso a utilização do conjunto de tarefas for maior que 100%.

$$\begin{aligned} R_i &= W_i + J_i \\ R_i &\leq D_i, \forall i \in T \end{aligned} \tag{2.8}$$

Exclusão Mútua

O compartilhamento de recursos é outra forma de relação entre tarefas em um ambiente de múltiplas tarefas. Basicamente, o recurso pode ser uma variável global ou mesmo uma estrutura de dados localizado em *seções críticas*.

A seção crítica é onde se encontram recursos a serem compartilhados entre duas ou mais tarefas. É preciso que o acesso desta seção seja exclusivo de uma tarefa por vez, pois manipulações de dados compartilhados simultaneamente podem levar a leitura suja de informações (*e.g.* mesma variável com valores diferentes armazenados na execução de cada tarefa) ou mesmo erros a prejudicar o progresso do sistema – este acesso exclusivo é conhecido como *exclusão mútua*.

Logo, caso uma tarefa esteja em uma seção crítica, qualquer outra tarefa que tentar executar a mesma seção deverá esperar até a primeira terminar de utilizar esses recursos para as demais terem acesso.

Um problema característico do compartilhamento de recursos e exclusão mútua é a *inversão de prioridades* – quando uma tarefa de menor prioridade bloqueia uma de maior prioridade. Por exemplo, sejam as tarefas i e j onde a prioridade de i é menor que a de j; a tarefa i acessa e bloqueia a seção crítica S antes da inicialização de j; após j inicializar, i ainda contém S, assim j não poderá utilizar os recursos de S até a liberação por i. j está, portanto, bloqueada pela tarefa menos prioritária i. As duas principais soluções para este problema consistem no *Protocolo de Herança de Prioridade* – PHP, e *Protocolo de Prioridade Teto* – PCP (do inglês, *Priority Ceiling Protocol*).

Sha *et al.* (1990) definem o PHP como um protocolo no qual a prioridade das tarefas é dinâmica. Nesses termos, se uma tarefa de menor prioridade utiliza um recurso compartilhado e outra mais prioritária tentar acessá-lo, esta será bloqueada. A tarefa menos prioritária assume a prioridade da tarefa bloqueada e continua a utilizar o recurso compartilhado até liberá-lo, e voltar a assumir a prioridade original.

Dado o exemplo descrito anteriormente, a tarefa i de menor prioridade herdará a prioridade de j ao executar S. A prioridade de i voltará ao valor original logo após o

término da execução da seção crítica e j poderá enfim utilizar S.

Contudo, a determinação do número máximo de bloqueios que uma tarefa experimenta com a utilização do PHP não é trivial, uma vez que tarefas mais prioritárias estão sujeitas a dois tipos de bloqueios: tentar acessar um recurso compartilhado já bloqueado por outra tarefa de menor prioridade, ou uma tarefa de prioridade intermediária ser bloqueada por outra que tenha herdado a prioridade de uma mais prioritária. A equação definida por Burns & Wellings (1997) procura estimar o número de bloqueios B_i em uma tarefa i a partir do PHP (veja Equação (2.9)).

$$B_i = \sum_{s=1}^{S} \textit{utilização}(s, i) * C(s) \tag{2.9}$$

onde S é o número de seções críticas do conjunto de tarefas; *utilização*(s, i) será 1 se a tarefa i sofrer bloqueio da seção crítica s, ou seja, por já ter uma tarefa de menor prioridade com a manipulação dos recursos de s. *utilização*(s,i) será 0 se não houver uma tarefa já em execução. Já $C(s)$ corresponde ao tempo de execução no pior caso de s. Mesmo com a determinação do número máximo de bloqueios devido ao PHP, não há como prevenir a ocorrência de *deadlocks* durante a execução de um conjunto de tarefas.

Deadlock é um problema de exclusão mútua onde uma tarefa i bloqueia uma seção crítica A e em seguida, sem liberar A, tenta manipular os recursos de outra seção B. No entanto, uma tarefa j já bloqueava os recursos de B, assim, i não pode acessar os recursos em B. j tenta, então, bloquear A já em utilização por i. Desta forma, i espera j liberar B, enquanto j espera i liberar A. Esta situação é conhecida por *deadlock*.

Um exemplo de *deadlock* na utilização do PHP ocorre quando uma tarefa i de menor prioridade bloqueia os recursos de uma seção crítica B. A execução de uma tarefa j de maior prioridade preempta i. j bloqueia outra seção crítica A durante a execução e, sem liberar este recurso, tenta acessar B, porém i já manipula os recursos de B. Então, i herda a prioridade de j de acordo com o PHP e retoma a execução. Porém, i tenta acessar A, ainda sobre o bloqueio de j, sem liberar os recursos em B. Como as prioridades agora são iguais devido ao PHP, ambas as tarefas esperam pela liberação dos recursos bloqueados por cada uma, assim, caracteriza-se um *deadlock*. A utilização do PCP em sistemas de tempo real críticos previne as ocorrências de *deadlocks* no acesso às seções críticas.

O PCP é uma extensão do PHP já que, se uma tarefa menos prioritária bloquear outra de maior prioridade, a menor assumirá a prioridade da maior. Contudo, todas as seções críticas a serem executadas possuem um valor de *prioridade teto* igual à prioridade da tarefa mais prioritária que acessa esses recursos – o valor da prioridade teto é definido em tempo de projeto. Uma tarefa só acessa uma seção crítica se a sua prioridade ativa (original ou herdada) for maior que a prioridade teto das seções críticas previamente bloqueadas, com a exclusão das seções onde a própria tarefa já adquiriu acesso.

De acordo com o exemplo do *deadlock*, a tarefa i acessa as seções críticas B e A, respectivamente; já a tarefa j acessa na ordem A e B; como A e B são acessados por i e j, a prioridade teto de ambos é igual à de j, pois possui a maior prioridade; quando i inicia a execução e tenta bloquear A, verifica-se se as demais seções críticas estão bloqueadas, no caso, nem A, nem B estão, então i executa A; em seguida, ocorre a ativação de j e, consequentemente, a preempção de i; j tenta acessar o recurso B, mas dentre as duas seções críticas existentes, i já bloqueia A cuja prioridade teto é igual à de j.

Como as prioridades são iguais, a execução de B por j ainda não acontece e i herda a prioridade de j, assim i volta a executar; ainda no bloqueio de A, i tenta acessar B e consegue uma vez que a única seção crítica bloqueada está também por i; ao terminar de executar B e em seguida A, i libera os bloqueios e j assume a execução por ter uma prioridade maior. A vantagem da utilização do PCP, além de evitar *deadlocks*, é permitir uma tarefa mais prioritária ser bloqueada por uma de menor prioridade apenas uma vez por execução (Sha *et al.*, 1990).

O bloqueio máximo B_i de uma tarefa i no PCP corresponde à duração da maior seção crítica de tarefas menos prioritárias que podem bloquear i. Segundo Burns & Wellings (1997), o valor máximo de bloqueio B_i em i é calculado a partir da Equação (2.10).

$$B_i = max\{\sum_{s=1}^{S} utilização(s,i) * C(s)\} \tag{2.10}$$

O valor de B_i influencia diretamente na janela de tempo da tarefa i. Como B_i refere-se ao tempo que uma tarefa está bloqueada devido à exclusão mútua, W_i deve refletir esse valor em ambientes com compartilhamento de recursos. Assim, a Equação (2.8) pode ser modificada para incluir interferências de exclusão mútua entre tarefas:

$$W_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} (\lceil \frac{W_i^n + J_i}{P_j} \rceil * C_j) \tag{2.11}$$

2.3 DVFS

A presença de circuitos CMOS (do inglês, *Complementary Metal-Oxide-Semiconductor*) é cada vez mais constante no projeto de sistemas embarcados, principalmente no desenvolvimento de processadores.

Nestes circuitos, observa-se o consumo dinâmico de energia como o principal responsável pela dissipação (Baums & Zaznova, 2008). Shin *et al.* (2001) definem o consumo de energia proporcional ao desempenho conforme a Equação 2.12.

$$E \propto C * N_{ciclos} * V_{dd}^2 \tag{2.12}$$

onde E é a energia consumida, C é a capacitância do circuito, N_{ciclos} é o número de ciclos necessários para a execução de uma tarefa e V_{dd}^2 é a tensão de alimentação. Verifica-se ainda que, com base na redução da tensão, o consumo de energia pode ser otimizado uma vez que a dissipação de energia tem relação quadrática com a tensão aplicada. Porém, reduzir a tensão pode implicar também diminuir a frequência e consequentemente o desempenho do processador.

Uma exemplificação onde a redução da tensão e frequência resulta na otimização do consumo de energia seria: de posse de um processador cuja troca de frequência em tempo de execução é possível, ocorre o processamento de uma tarefa que necessita de $100ms$ para ser concluída; a execução desta tarefa com a frequência máxima F e tensão V disponíveis duraria aproximadamente $50ms$; a diferença entre o *deadline* de $100ms$ e o tempo de execução com a maior frequência resulta também em $50ms$ de ociosidade pelo processador – estado em que ainda ocorre consumo de energia.

Ocorre que, caso a frequência e tensão pudessem ser reduzidas pela metade em tempo de execução, a mesma tarefa cumpriria o *deadline*, pois $T = 50ms, F = \frac{1}{T}, F' = \frac{F}{2} = \frac{1}{2T} \therefore T' = \frac{1}{F'} = 2T = 100ms$, e haveria redução de 1/4 da energia consumida já que a relação entre energia consumida e tensão é quadrática. A troca destes elementos em tempo de execução é a ideia base da técnica de escalonamento dinâmico de tensão e

frequência (DVFS – *Dynamic Voltage and Frequency Scaling*) (Shin *et al.*, 2001).

Há duas abordagens para a utilização desta técnica:

- **Escalonamento inter-tarefa**: a determinação da tensão de uma tarefa ocorre a partir da análise da execução *entre* tarefas. Estudos relacionados, como Lee & Krishna (1999), Kim *et al.* (2002), e Valentin (2010), normalmente consideram relações de precedência e exclusão mútua para calcular a frequência de cada tarefa, a partir da otimização do tempo de folga entre elas;

- **Escalonamento intra-tarefa**: a determinação da tensão e frequência de uma tarefa ocorre com base na análise do fluxo da sua execução. Estudos relacionados, como Tatematsu *et al.* (2011), Azevedo *et al.* (2002) e Shin *et al.* (2001), normalmente analisam o comportamento da execução de apenas uma tarefa. Análises como o pior caso do fluxo de execução, instrumentação de código ou geração de perfis de execução são metodologias empregadas para tentar determinar as novas tensões e frequências que uma tarefa pode assumir.

2.4 Análise Estática de Código

Analisar o comportamento de uma tarefa consiste em duas partes: nível de *hardware* e *software*.

Em nível de *hardware*, quando as instruções de máquina são executadas, elas estão sujeitas a diversas limitações físicas intrínsecas às arquiteturas dos computadores, como *pipeline*, barramento de comunicação entre memória e processador, dissipação de calor, entre outros. Destaca-se que apenas os projetistas da arquitetura conseguem ter maior controle sobre tais limitações

Entretanto, otimizações no consumo de energia de uma tarefa tornam-se uma alternativa em nível de *software*, em virtude dos seus peculiares componentes determinantes para o controle e definição das tarefas (ex: modelo de tarefas, políticas de escalonamento, escalonador). Assim, a partir da análise do comportamento de uma tarefa, pode-se otimizar o consumo de energia com base no fluxo atual de execução, se este não exigir a capacidade máxima de processamento. Porém, como é possível analisar o comportamento de uma tarefa?

A representação de tarefas normalmente divide-se em três partes: código em alto nível (linguagens como *ANSI-C*, *Python*, *Java*), código intermediário (*Assembly*) e o código de máquina (binário). A análise do comportamento de uma tarefa usualmente restringe-se às primeiras duas partes.

Em Ball & Laurus (2000) verifica-se o uso constante da análise do fluxo de execução de tarefas a partir do código em alto nível, por exemplo: um programa pode ser representado como um grafo de fluxo de controle – CFG (do inglês *Control Flow Graph*), onde cada nó representa um bloco sequencial de execução do código e as arestas, possíveis desvios. Uma aplicação do CFG está na previsão de desvios – determinar qual caminho será percorrido a partir de um desvio condicional para, assim, recuperar uma instrução da memória antes do momento e evitar o estado de *stall* no *pipeline* do processador.

Outro exemplo está na otimização de código pelos compiladores já que podem identificar possíveis caminhos redundantes e alterá-los para um fluxo mais eficiente. Outras aplicações ainda podem ser derivadas a partir desta análise como a corretude de um código, onde se verifica se, independente do estado das variáveis, o programa em questão sempre apresentará um resultado válido no fluxo de execução.

Os exemplos anteriores apresentam como a análise do comportamento de uma tarefa, a partir de um grafo de fluxo de controle, fornece informações importantes em tempo de projeto. Logo, a análise estática pode auxiliar não só na redução de riscos, como também no aumento da eficiência durante a execução. A análise do grafo de fluxo de controle de uma tarefa em sistemas de tempo real normalmente está associado a identificação do pior caso do fluxo de execução deste grafo.

2.5 Pior Caso do Fluxo de Execução de uma Tarefa

Determinar o pior caso do fluxo de execução de uma tarefa, não é trivial. Isto porque algumas tarefas podem ter o comportamento variável de acordo com as informações de entrada. Estas variações e a complexidade de uma tarefa podem tornar o problema de encontrar o pior fluxo de execução algo impraticável.

Alguns estudos na literatura como Wilhelm *et al.* (2008), Shin *et al.* (2001) e Cohen (2011) tratam este problema. Estes trabalhos permitem destacar duas abordagens comumente utilizadas: determinação do tempo necessário para o término de uma tarefa

durante o pior caso de execução – WCET (*Worst-Case Execution Time*), e identificação do pior caminho no CFG – WCEP (*Worst-Case Execution Path*).

O WCET procura utilizar o maior número possível de configurações do fluxo de execução de uma tarefa, a fim de estimar com maior precisão o tempo de resposta dela no pior caso, sem necessariamente obter o CFG. Wilhelm *et al.* (2008) definem dois métodos como principais na aplicação do WCET: estático e baseado em medições.

O método estático realiza uma análise individual das informações pertinentes a uma tarefa. Ele ainda se baseia nos dados extraídos em tempo de compilação, como o fluxo de controle e as instruções de máquina geradas. Já os métodos baseados em medições, restringem-se ao estudo do fluxo de controle de uma tarefa a partir da determinação do tempo necessário para executar cada bloco da mesma.

Por outro lado, Shin *et al.* (2001) e Cohen (2011) apresentam estudos para a redução do consumo de energia com base no CFG para se obter o WCEP. Primeiramente, em ambos os trabalhos, deriva-se o grafo de fluxo de controle de uma tarefa e calcula-se, para cada nó, quantos ciclos do processador são necessários para a execução de cada um com base nas instruções do código intermediário. Em seguida, determina-se o caminho de maior custo no grafo.

É importante ainda destacar o número de ciclos a serem executados pelos nós do CFG no estudo do WCEP, pois são eles a representar o custo de cada elemento do grafo. O CFG normalmente é gerado a partir do código de alto nível. Em seguida, realiza-se o mapeamento de cada expressão para as instruções no código intermediário. Esta relação, contudo, não é trivial ao ponto de ser 1:1, pois a geração do código intermediário é dependente de cada compilador, o qual ainda aplica otimizações durante o processo de compilação a fim de aumentar a eficiência do código final e evitar conflitos (como memória e dados) durante a execução. Estas otimizações também podem excluir códigos sem influência no comportamento de uma tarefa. Wilhelm *et al.* (2008) comentam sobre isto uma vez que também está relacionado ao WCET, e denominam análise de anomalias no tempo de cada instrução.

Nesta anomalia, tem-se a dificuldade de se determinar o tempo de execução de uma tarefa quando poucas informações são fornecidas, tanto desta quanto do processador, a exemplo do tempo necessário para executar uma instrução, o qual é variável de acordo

com o estado atual do processador.

Por exemplo, considera-se o acesso à memória *cache* para recuperar um determinado dado. Em um momento, esta informação pode estar presente na memória, em outro não, logo, tem-se como consequência uma falta na memória *cache*, a qual deverá recuperar o dado da memória principal ou secundária. O tempo necessário para recuperar o dado aumentará o tempo de término da instrução. Assim, é necessário conhecer também o comportamento e a arquitetura de cada processador para poder fazer uma análise temporal de cada instrução de máquina (Heckmann *et al.*, 2003) e, assim, determinar o pior fluxo de execução.

Capítulo 3

Trabalhos Correlatos

O uso da técnica DVFS está diretamente associado a relação quadrática entre tensão e consumo de energia em dispositivos com tecnologia CMOS. Na literatura há diversos estudos que exploram esta relação em sistemas de tempo real críticos, destacando-se duas principais abordagens do uso da DVFS: inter-tarefa e intra-tarefa.

3.1 DVFS: Inter-Tarefa

Em Kim *et al.* (2002), propõe-se um novo algoritmo para estimar o tempo de ociosidade em tarefas periódicas sujeitas à política de escalonamento EDF – *Earliest Deadline First*. O estudo avalia a aplicação da DVFS inter-tarefa durante a execução das tarefas.

O algoritmo consiste em, dada uma lista de tarefas em espera e prontas para ativação, explorar o tempo ocioso da tarefa anterior. Desta forma, como a ociosidade é um fator acumulativo, se a execução de uma tarefa terminou e ainda há algum tempo ocioso, a próxima tarefa poderá aproveitá-lo.

A política EDF define a restrição temporal de uma tarefa de acordo com o período desta. Kim *et al.* (2002), então, propõem que, se uma tarefa for preemptada por outra de maior prioridade, considerando-se o tempo restante e o comportamento da tarefa de menor prioridade, será possível estimar o tempo ocioso a ser gerado e o valor da frequência para a tarefa atual. Já a tarefa preemptada, ao retomar a execução, terá a frequência calculada com base no tempo disponível pelo processador e também pela restrição temporal.

Nesse caso, os autores obtiveram uma otimização no consumo de energia de 40% ao comparar com a aplicação da DVFS de outros dois estudos. Entretanto, a proposta de Kim *et al.* (2002) não determina se as frequências utilizadas durante a execução resultam em um ganho ótimo no tempo ocioso, diferentemente da proposta em Valentin (2010),

onde se determina um conjunto inicial de frequências cujo tempo ocioso total é o menor possível.

Por sua vez, o estudo em Lee & Krishna (1999) também mantém o uso do escalonamento com prioridade fixa, todavia de acordo com a política RM. Neste estudo, a solução é dividida em duas fases: estática e dinâmica. Em ambas as fases, consideram-se dois conjuntos de frequências, o primeiro representa a execução das tarefas com a maior frequência disponível, enquanto o segundo, com a menor.

A abordagem estática calcula o fator de utilização do processador para as duas frequências, assim, tem-se um conjunto do custo de cada tarefa. A atribuição da menor ou maior frequência é tratada como um problema de decisão da soma de subconjuntos, onde se deseja encontrar o subconjunto mínimo entre os fatores de utilização, a fim de cumprir com a restrição temporal de cada tarefa. Portanto, os elementos do conjunto solução deste problema correspondem às frequências a serem atribuídas a cada tarefa.

Já na abordagem dinâmica, após a atribuição inicial, qualquer tarefa pode assumir uma das duas frequências. Se uma tarefa for preemptada por outra de maior prioridade, determina-se o tempo de espera até a tarefa de menor prioridade retomar a execução. Desta forma, quando houver uma tarefa de menor prioridade ou nenhuma em espera, altera-se a frequência para a menor possível e esta é mantida – mesmo com uma tarefa de maior prioridade, até o tempo de espera acumulado ser zero. Em seguida, adota-se novamente a maior frequência.

Observa-se tanto no trabalho de Lee & Krishna (1999), quanto no de Kim *et al.* (2002) limitações no uso da frequência. Kim *et al.* (2002), ao calcularem o valor da frequência para tarefa atual, consideram aceitável qualquer valor contínuo no intervalo entre a mínima e máxima frequência disponível. Contudo, nos processadores atuais, este intervalo é discreto e limitado.

Entretanto, Lee & Krishna (1999) utilizam apenas as frequências máxima e mínima disponíveis. Ocorre que, se tratados em conjunto, os demais valores discretos podem proporcionar uma redução maior de energia, como o presente trabalho que analisa todas as frequências disponíveis pelo processador.

Em Lee *et al.* (2009), os autores consideram a utilização da DVFS em processadores *multi-core* para dispositivos móveis. O principal objetivo é determinar uma frequência de

modo a cumprir a restrição temporal com o menor número possível de núcleos alocados para uma dada tarefa, enquanto os demais núcleos são desligados. Neste estudo, observa-se ainda um tempo ocioso na execução de uma tarefa se o tempo de execução com uma frequência e núcleos previamente calculados for menor do que a restrição temporal. Os autores definem o início do tempo ocioso como um ponto de transição, no qual uma nova frequência pode ser utilizada para otimizar o consumo de energia com o mesmo número de núcleos.

Ressalta-se ainda a opção dos autores por não desligar ou alocar novos núcleos durante a execução de uma tarefa, pois o tempo necessário para realizar qualquer um dos procedimentos é elevado ao ponto de reduzir a eficiência da proposta. Então, se uma nova frequência fosse adotada após o ponto de transição, bem como o tempo para terminar a execução da tarefa permanecesse inferior à restrição temporal, os núcleos entrariam em estado ocioso. O principal diferencial de Lee *et al.* (2009) está na utilização da DVFS em um cenário *multi-core*, contudo o estudo não considerou a presença de preempção ou dependência entre tarefas.

O trabalho proposto por Valentin (2010), utilizado como base para o cálculo da frequência inicial desta dissertação, apresenta um algoritmo *branch-and-bound* para determinação de um conjunto ótimo de frequências iniciais a serem atribuídas para cada tarefa, com a análise do pior caso de execução de um modelo de tarefas. Como resultado, Valentin (2010) analisou cenários onde há não só precedência entre tarefas, mas também compartilhamento de recursos, ao verificar se um modelo de tarefas é escalonável a partir de um teste de escalonabilidade.

Uma desvantagem comum da técnica DVFS inter-tarefa consiste em não analisar o fluxo de execução específico de uma tarefa. A abordagem intra-tarefa, contudo, preocupa-se em identificar possíveis caminhos que desviam do pior caso do fluxo de execução de uma tarefa, além de alterar a frequência e tensão quando possível.

3.2 DVFS: Intra-Tarefa

A abordagem DVFS intra-tarefa consiste na análise do fluxo de execução das tarefas a fim de determinar como e qual valor adotar ao alterar-se a frequência e tensão. Os estudos apresentados a seguir utilizam como base a análise estática de código.

Em Tatematsu *et al.* (2011), apresenta-se um estudo a partir da análise do grafo de fluxo de controle de uma tarefa. Os autores propõem um algoritmo para inserção de novas instruções de código a fim de verificar se a troca de frequência e tensão é ou não possível. Cada inserção no código original da tarefa recebe o nome de *checkpoints*.

O passo inicial do algoritmo consiste em executar um conjunto de casos de teste para percorrer os caminhos do CFG de uma tarefa. Desta forma, armazenam-se os dados resultantes de cada execução, como o número de ciclos necessários para os nós percorridos.

Em seguida, comparam-se os resultados dos caminhos e determina-se em qual deles a execução exige o maior número de ciclos. Assim, define-se para cada desvio percorrido durante os casos de teste, um possível candidato para representar um *checkpoint*. Porém, quanto maior for o número destes *checkpoints* no código, maior será o *overhead* – custo adicional em ciclos devido às novas instruções. Por tal razão, os autores aconselham a adoção de apenas alguns *checkpoints*.

Ademais, destaca-se que cada caminho percorrido inicialmente é mais uma vez executado e analisa-se quais dos desvios de fluxo encontrados apresentam um menor custo de execução. Os *ckeckpoints* a demandarem menos processamento estarão presentes no código final e fora do pior caminho.

O resultado de Tatematsu *et al.* (2011) corresponde a um novo código com a inserção de alguns *ckeckpoints* que, quando percorridos, analisam o estado atual do fluxo de execução e, se possível, alteram frequência. O problema mais grave relacionado a inserção dos *checkpoints* refere-se à restrição temporal das tarefas, pois não há como determinar quantos *checkpoints* devem ser inseridos no código original a fim de se otimizar a energia sem infringir o *deadline*.

Os trabalhos propostos por Azevedo *et al.* (2002) e Shin *et al.* (2001) também consistem em identificar pontos onde a frequência e tensão podem ser alteradas. O primeiro divide o código em seções e determina uma restrição temporal para cada parte. Já o segundo determina a frequência ideal para cada desvio do fluxo de execução.

O algoritmo proposto em Azevedo *et al.* (2002) identifica os possíveis pontos do CFG para troca de tensão. Estes pontos são determinados como qualquer desvio de fluxo. Em seguida, define-se uma restrição temporal para cada desvio a partir das transições para

os demais nós. Por fim, adota-se a transição a demandar maior tempo de execução e calcula-se a frequência ideal para o bloco em questão.

Em Shin *et al.* (2001), o segundo trabalho base desta dissertação, analisa-se estaticamente o comportamento de uma tarefa pelo CFG. Com base neste dado, eles determinam o pior caminho no fluxo de execução do grafo a partir do número de ciclos a serem executados em cada nó. Se houver um desvio do pior caminho durante a execução da tarefa, calcula-se a frequência ótima para executar os ciclos restantes, respeitar a restrição temporal e reduzir ao máximo o tempo ocioso. Todavia, os estudos realizados por Azevedo *et al.* (2002) e Shin *et al.* (2001) têm um problema em comum, o uso da frequência ideal que está presente em um conjunto contínuo de valores, o que não é real.

Os processadores possuem um conjunto discreto e limitado de frequências disponíveis, de modo que não há como considerar a frequência como contínua e ainda garantir o cumprimento da restrição temporal de cada tarefa. Além disso, estes estudos não abordam cenários onde ocorrem preempção e compartilhamento de recursos entre tarefas.

Cohen (2011), entretanto, propõe a utilização da DVFS intra-tarefa com base nas frequências e tensões disponíveis em um processador. O autor apresenta uma analogia da aplicação da técnica DVFS em uma corrida de carros. Cada carro representa uma frequência disponível e todos percorrem o fluxo de execução de uma tarefa – de forma abstrata já que apenas uma frequência por vez é adotada pelo processador. Se em algum momento da execução um carro ultrapassar ou for ultrapassado, então a frequência e tensão devem ser alterados.

A principal contribuição de Cohen (2011) foi proporcionar redução do consumo de energia inclusive no pior caso do fluxo de execução de uma tarefa. Porém, a extensão deste trabalho em ambiente multitarefas não garante a escalonabilidade do modelo de tarefas em sistemas de tempo real críticos.

3.3 DVFS: Inter-Tarefa e Intra-Tarefa

AbouGhazaleh *et al.* (2003) propõem a utilização de ambas as abordagens da DVFS em um ambiente colaborativo entre o sistema operacional e o compilador. Os autores dividem a proposta em duas fases: *off-line*, onde ocorre a identificação dos pontos para troca de frequência e tensão; e *on-line*, na qual a troca de frequência ocorre durante a

execução de um processo.

A parte *off-line* utiliza o compilador para extrair informações sobre o possível comportamento temporal de uma tarefa. Então, divide-se o fluxo de execução em segmentos e impõe-se a chamada de uma interrupção em um período de tempo estimado. Instruções também são inseridas no código fonte de uma tarefa para manter o controle de quantos ciclos já foram realizados. A parte *on-line* já fica a cargo do sistema operacional, o qual manterá o controle das chamadas de interrupção.

Uma vez alcançado o tempo da interrupção definido na parte *off-line*, interrompe-se a tarefa e avalia se o número de ciclos realizados até o momento permite a alteração da tensão com o cumprimento da restrição temporal. A adição dos *overheads* nesta abordagem é um ponto a ser destacado já que o cálculo da nova tensão e o tempo gasto para adotá-la influenciam na energia consumida e no desempenho das tarefas em execução. Além disso, os autores ainda obtiveram uma economia de 57% no consumo da energia.

A implementação do trabalho relatado por AbouGhazaleh *et al.* (2003) não é trivial, pois não só o compilador deve sofrer alterações, como também o sistema operacional, onde é necessário a implementação de novas chamadas de sistema e alterações no escalonador.

Outro estudo sobre a utilização da DVFS inter-tarefa e intra-tarefa encontra-se em Seo *et al.* (2005). A primeira parte deste trabalho consiste em analisar estaticamente o conjunto de tarefas do sistema e produzir o grafo de fluxo de controle.

Ainda neste processo, determina-se o número de ciclos para cada bloco e atribui-se uma probabilidade do fluxo de execução em adotar um desvio. Com base nas frequências disponíveis pelo processador, os autores determinam o tempo de computação da tarefa e, em seguida, os tempos inicial e final da execução, de modo a otimizar o consumo de energia. A frequência utilizada para calcular o tempo de computação de uma tarefa corresponde ao limite inferior das possíveis frequências a serem adotadas.

Deste modo, os autores garantem que qualquer valor de frequência abaixo deste limite não respeitará a restrição temporal. Ainda na análise intra-tarefa, cada desvio no fluxo de execução é um possível candidato para troca de frequência e tensão.

A parte inter-tarefa do estudo de Seo *et al.* (2005) necessita de três informações essenciais: o tempo de chegada, o *deadline* e o número de ciclos restantes para o término

da execução da tarefa.

O algoritmo proposto por Seo *et al.* (2005) identifica um intervalo crítico onde um conjunto de tarefas deve ser completado sem sofrer alteração de outras tarefas fora deste conjunto, a fim de minimizar o consumo de energia. As tarefas neste intervalo crítico executam com a maior frequência disponível e são ordenadas com base na política de escalonamento EDF. A desvantagem da proposta de Seo *et al.* (2005) está em não considerar cenários onde as tarefas possam ser dependentes ou mesmo compartilhar recursos.

Capítulo 4

Método Proposto

Este capítulo define a metodologia proposta para a análise estática do comportamento de uma tarefa, a partir da geração do grafo de fluxo de controle de um programa em C. Ainda, propõe-se uma estratégia para determinar o número de ciclos a serem consumidos no pior caso do fluxo de execução de uma tarefa. O cálculo da nova frequência a ser utilizada durante a execução de uma tarefa e qual a frequência inicial também são tópicos abordados neste capítulo.

4.1 Grafo de Fluxo de Controle

O projeto de compiladores normalmente está sujeito a três principais fases: *front-end*, otimização e *back-end*.

O *front-end* é responsável pela análise léxica, sintática, semântica e geração do código intermediário.

A análise léxica encarrega-se de reconhecer os símbolos aceitos pela linguagem. Já a sintática, verifica se a estrutura gramatical está correta, enquanto a semântica analisa o contexto das operações (*e.g.* operação de soma com dados de tipos diferentes). A última etapa do *front-end*, é responsável por gerar o código intermediário a ser utilizado na fase de otimização do compilador.

As otimizações, no código intermediário, têm como objetivo melhorar o desempenho do código final a partir da aplicação de heurísticas ou reescrita de código. Já a última fase, o *back-end*, é o responsável por transformar o código intermediário em código de máquina (Cooper & Torczon, 2012).

Dentre as três principais fases de compilação, a do *front-end* é a que melhor permite identificar a estrutura de um código. Isto ocorre devido a etapa de análise sintática em que os símbolos de uma linguagem, já reconhecidos, são comparados com a estrutura

gramatical. Assim, se uma determina linha de código corresponder a uma regra válida da gramática, então esta linha pertence a linguagem. Uma das formas de se produzir o CFG é justamente pela análise da gramática de uma linguagem, mais especificamente as gramáticas livres de contexto.

Gramáticas livres de contexto são aquelas cujas regras de produção possuem um ou mais símbolos não-terminais, os quais serão decompostos em um elemento vazio ou um ou mais símbolos terminais ou outros não-terminais. Porém, a forma como essas regras são decompostas influencia diretamente na eficiência da análise estática do código. Mais detalhes sobre o desenvolvimento de uma gramática pode ser visto em Levine (2009).

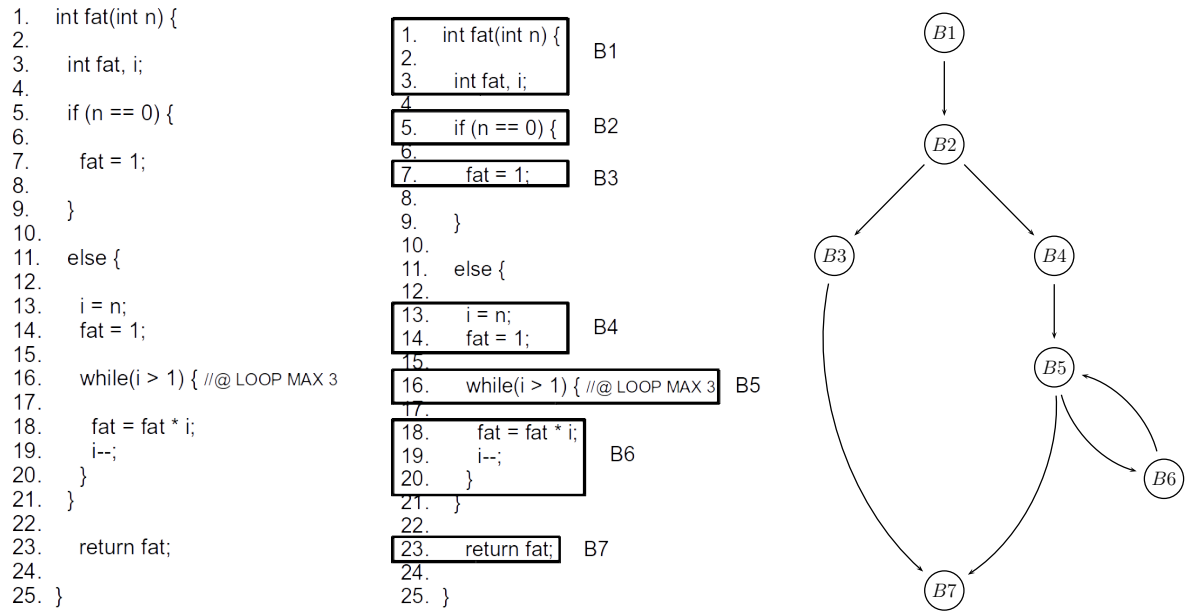
O presente estudo assume a utilização de gramática genérica para a linguagem C. O termo genérico, neste caso, faz referência ao reconhecimento de blocos – conjunto de instruções, estruturas comuns – iteração e seleção, e das definições das funções, pois são estes elementos que alteram o fluxo de execução de uma tarefa.

Ao mapear essas características para um CFG, têm-se as seguintes definições:

- Um bloco corresponde a um nó;
- Os desvios condicionais são as arestas;
- Qualquer condição dos desvios corresponde a um único nó. Assim, se a condição tiver como origem uma estrutura de seleção (ex: *if*), então ela será um nó pai com dois filhos de destino: o *then* e o *else*;
- Se a condição tiver como origem uma estrutura de iteração (ex: *while*), ela também será um nó pai com outros dois filhos, o *then* – que se refere à execução do laço, e o nó de saída do laço – que indica a continuação do fluxo;
- Como um código em C pode conter a definição de uma ou mais funções, cada função é tratada como um subgrafo do CFG da tarefa.

É possível obter o grafo de fluxo de controle, como uma abstração da análise estática do comportamento de uma tarefa, a partir das definições listadas acima. A Figura 4.1 apresenta graficamente esses pontos.

O grafo de fluxo de controle, contudo, carece de informações, pois não há como determinar qual dos possíveis caminhos do CFG demandará maior processamento. Para



(a) Código exemplo. (b) Divisão em blocos. (c) Grafo de fluxo de controle.

Figura 4.1: Visualização do código em blocos e no CFG.

isto, utiliza-se o número de ciclos necessários para execução de cada nó do grafo. Uma vez que cada nó corresponde a um conjunto de instruções, caso se conheça o custo de execução de cada instrução, a soma delas corresponderá ao custo total do nó.

Por fim, com o custo dos nós associados, é possível determinar o custo total dos caminhos no CFG, inclusive o pior caso.

4.2 Número de Ciclos por Bloco

Um bloco em C é representado por uma ou mais linhas de códigos nas quais cada instrução necessita executar um número específico de ciclos do processador. Na fase *back-end* do compilador, converte-se um código intermediário em linguagem de montagem – *Assembly*. Esta linguagem é a última representação legível de um código, visto que depois ele será convertido para linguagem de máquina.

O interessante da representação em *Assembly* é que cada processador disponibiliza uma documentação, conhecida como *datasheet*, onde informações como os ciclos necessários

para executar instruções em linguagem de montagem estão disponíveis. Assim, para determinar o custo de uma instrução de alto nível, realiza-se a tradução de um código para *Assembly* e consulta-se o *datasheeet*.

O código *Assembly* apresenta um conjunto de instruções referentes a todas de alto nível. Na linguagem C, uma instrução corresponde a uma ou mais instruções em linguagem de montagem. Com as informações de depuração – *debug*, do compilador *gcc* em *Assembly*, é possível identificar a qual linha do código em C as instruções em *Assembly* correspondem.

Por exemplo, uma das informações de *debug* é a expressão *.loc*, que informa qual o número da linha em C correspondem as instruções logo abaixo desta expressão. A Figura 4.2 apresenta um exemplo mais prático das informações de *debug* com a expressão *.loc*. Neste caso, abaixo da expressão *.loc 1 7 0*, estão todas as instruções *Assembly* correspondentes à instrução da linha 7 no código C.

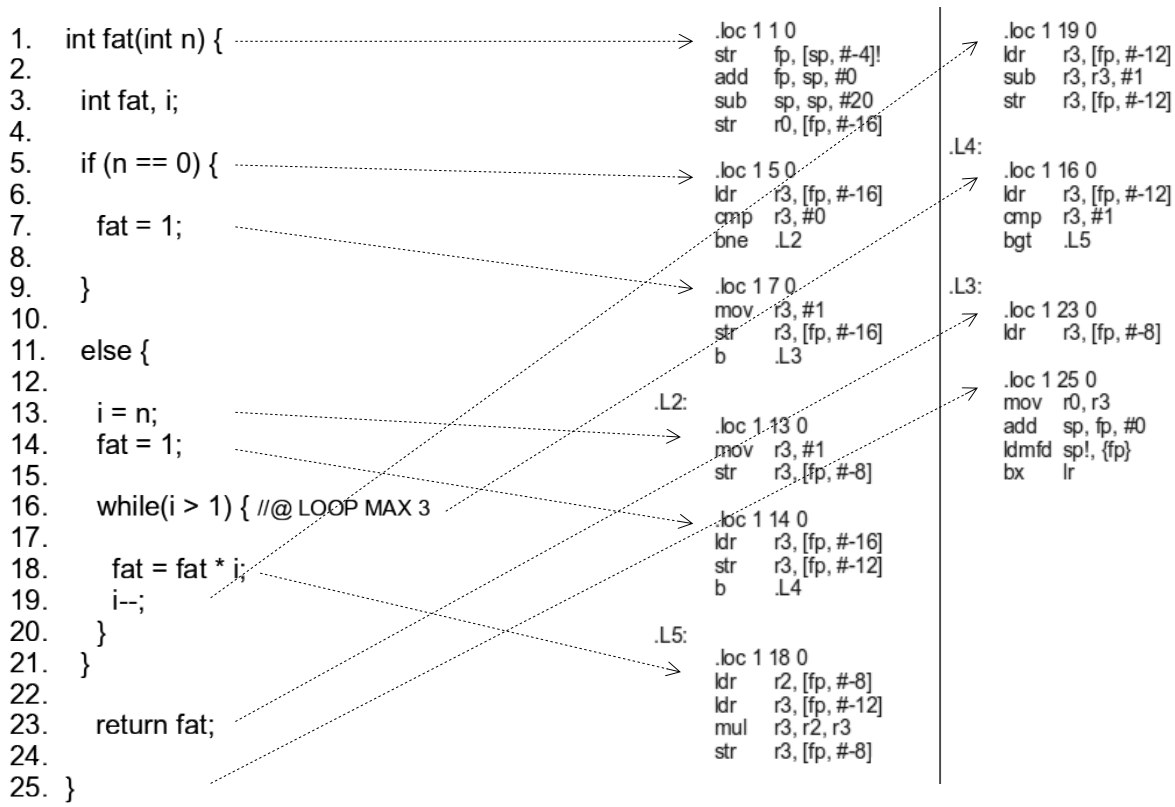


Figura 4.2: Conversão do código C para *Assembly*.

Basicamente, a expressão *.loc fileno lineno column* consiste em informações de localização para *debug*. A primeira coluna, *fileno*, indica com números inteiros a qual arquivo as instruções abaixo da expressão *.loc* fazem parte. Por exemplo, se o arquivo *main.c* incluir *lib.c* no cabeçalho, *main.c* é referenciado pelo número 1, enquanto *lib.c* por 2. *lineno* informa a qual linha do código em C as instruções abaixo de *.loc* referem-se. Já *colum* é uma informação utilizada pela estrutura interna do *gcc*.

Uma forma de se obter o código intermediário é realizar a compilação de uma tarefa com os seguintes parâmetros no *gcc*:

gcc -march=armv4t -g -S task.c -o task

A opção *-march=armv4t* faz com que a compilação do código dado como entrada tenha como alvo a arquitetura da família *armv4t*. A geração do arquivo *task.s* com o código intermediário desta arquitetura e informações de *debug*, ocorre a partir da utilização das opções *-g* e *-S*.

Para a presente dissertação, é essencial usar o *datasheet* da *armv4t* como um dicionário para relacionar o custo, em termos de ciclos de *clock*, necessário para execução de cada instrução do código intermediário. Assim, é possível obter o CFG com as informações de custo para executar cada nó, quando os ciclos de *clock* de todas as instruções de um bloco forem somadas.

O grafo de fluxo de controle, então, armazena os mesmos números das linhas de início e fim dos blocos do código de alto nível, para se ter controle do conjunto de instruções em C que compõe um determinado nó.

Em seguida, são localizadas as instruções pertencentes ao intervalo destas linhas no código *Assembly*, determina-se a quantidade de ciclos por instrução com base no *datasheet* e, assim, somam-se estes valores para atribuir o custo computacional de cada nó. Ao término deste processo, tem-se um grafo ponderado que possibilitará analisar o pior caminho.

4.3 Pior Caminho do Grafo de Fluxo de Controle

A ponderação de cada nó no grafo de fluxo de controle corresponde ao número de ciclos do processador a serem consumidos no pior caso de execução – WCEC (*Worst-Case*

Execution Cycles). Atribui-se esta definição tanto ao custo de execução de um nó, de uma função e ao custo total de uma tarefa. Desta forma, considerar a execução do número de ciclos no pior caso de um CFG significa percorrer o caminho do grafo cujo custo total de execução é maior a qualquer outro caminho possível.

Nesta dissertação, uma função, em nível de CFG, é um subgrafo composto por um ou mais nós. O custo total desta função no pior caso é representado pelo WCEC. Caso esta função seja chamada em outra, a instrução encarregada de chamá-la pode ser representada por outro nó, cujo custo em ciclos é igual ao WCEC da função.

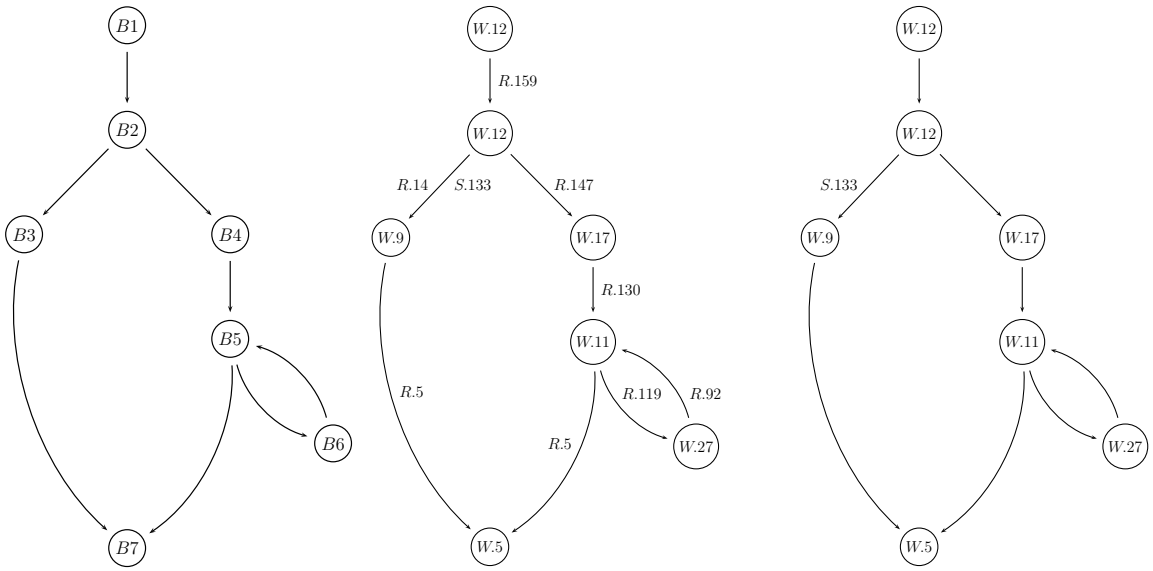
Definem-se ainda outros conceitos importantes com base na execução do número de ciclos do pior caso:

- **RWCEC** (*Remaining Worst-Case Execution Cycles*): representa quantos ciclos ainda restam ser executados até o término do pior caso ao sair de um bloco de instruções. Por exemplo, a aresta $B1(W.12) \rightarrow B2(W.12)$ da Figura 4.3a é ponderada com $RWCEC = 159$, como na Figura 4.3b. Por outro lado, a aresta $B2(W.12) \rightarrow B3(W.9)$ possui $RWCEC = 14$, ou seja, se o fluxo de execução, ao sair do nó B2, seguir pela aresta $B2 \rightarrow B3$, será necessário executar 14 ciclos para encerrar a execução da tarefa;

- **WCEP** (*Worst-Case Execution Path*): informação extraída da união do WCEC do nó e RWCEC. Este conceito define quais nós do CFG pertencem de fato ao pior caso. Na Figura 4.3a, por exemplo, o WCEC do pior caso é igual a 171 e o WCEP = {B1, B2, B4, B5, B6, B5, B6, B5, B6, B5, B7}. Os nós {B5, B6} aparecem três vezes no WCEP devido ao laço conter no máximo três iterações. Já a quarta aparição de B5 é a última vez que a condição do laço é executada, cuja resposta é falsa, assim, encerra-se o laço;

- **SEC** (*Saved Execution Cycles*): refere-se à quantidade de ciclos que deixaram de ser executados ao percorrerem um caminho diferente do pior caso. Em um nó de desvio no fluxo de execução – por exemplo, nó B2 (W.12) na Figura 4.3b, se a aresta $B2 \rightarrow B4$ for percorrida, tem-se o pior caso, com a execução restante de 147 ciclos (RWCEC). Contudo, se $B2 \rightarrow B3$ for a aresta utilizada, há um desvio no fluxo de execução do CFG, onde apenas 14 ciclos devem ser consumidos para o término da

tarefa. A diferença entre o RWCEC da aresta $B2 \rightarrow B4$ e $B2 \rightarrow B3$, corresponde a quantos ciclos deixaram de ser executados, uma vez que o pior caminho não foi percorrido. Para este caso, $SEC = RWCEC_{B2 \rightarrow B4} - RWCEC_{B2 \rightarrow B3} = 133$ (veja Figura 4.3c). Assim, um desvio no fluxo de execução no pior caminho proporcionou um consumo menor de 133 ciclos. Na Figura 4.3b, é interessante observar ainda como apenas três iterações do laço influenciaram neste exemplo.



(a) CFG. (b) WCEC(W.) e RWCEC(R.). (c) *Saved Execution Cycles.*

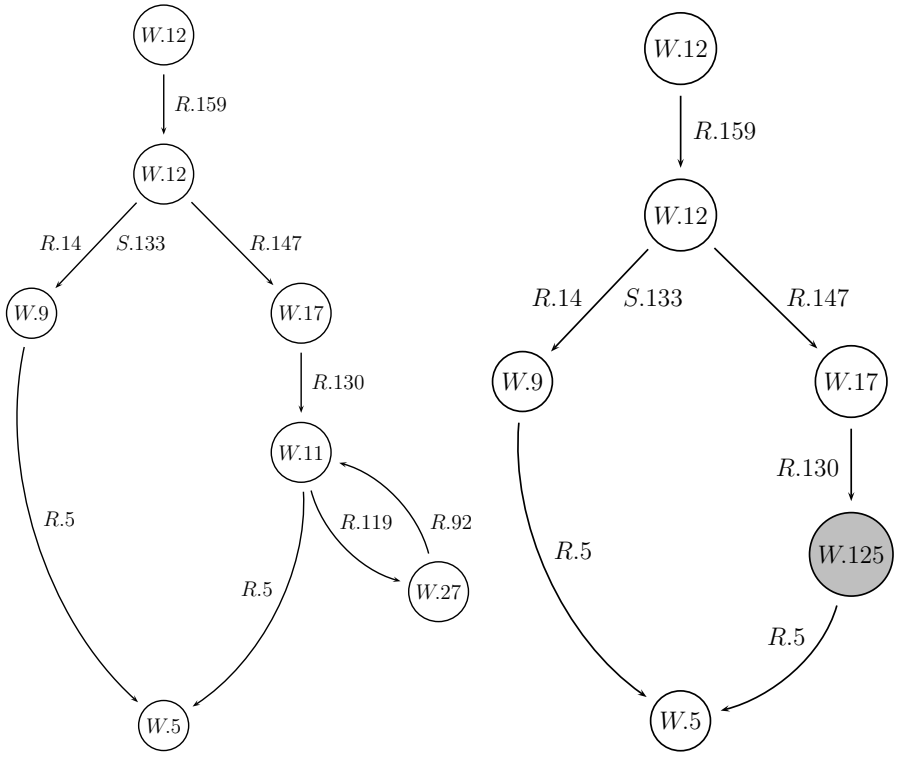
Figura 4.3: Visualização do Número de Ciclos por Bloco e do Pior Caminho.

Na presente dissertação, o número de iterações de um laço não é calculado, pois, como este valor não é determinístico, não há como saber com precisão este dado. Logo, assume-se que o usuário informa no código em C qual o número máximo de iterações do laço. Conforme a Figura 4.2 (linha 16), esta informação é apresentada com o padrão "*//@LOOP MAX número*" ao lado de um laço.

Por fim, ressalta-se o cálculo do número de ciclos do caminho de execução do pior caso. Este pode ser visto como o dual do problema de caminho mínimo em grafos. Por ele estar inserido na classe NP-Difícil, determinar o WCEP de uma tarefa não será computacionalmente viável, o que limita a análise estática de qualquer método proposto.

Contudo, para este problema, deve-se observar as características únicas: o grafo é direcionado, conhece-se a origem e o(s) possível(is) final(is), e há laços presentes no fluxo.

Estes laços podem ser reduzidos a um subgrafo abstraído no grafo original por um único nó, de maneira que ele conterá apenas o valor do WCEC do respectivo laço. Esta mesma abordagem pode ser utilizada para chamadas de função, ou seja, para evitar a inserção de um subgrafo dentro de outro subgrafo (uma chamada de função dentro de outra), limitando-se o subgrafo a apenas um nó. Assim, tem-se um grafo acíclico e direcionado (DAG – *Directed Acyclic Graph*), o qual pode ser representado por uma árvore cuja raiz e folhas são conhecidas. A Figura 4.4 ilustra as alterações.



(a) Grafo cíclico. (b) DAG.

Figura 4.4: Representação do CFG com *Directed Acyclic Graph*.

A transformação do CFG para um DAG reduz as restrições impostas e parte da complexidade do problema de encontrar o pior caminho em um grafo. Assim, qualquer algoritmo elementar de grafos pode ser aplicado, BFS – *Breadth-First Search*, ou DFS –

Depth-First Search, pois os resultados serão os mesmos.

4.4 Geração do CFG

Os diversos fluxos no comportamento de uma tarefa, assim como o pior caso de execução podem ser obtidos por meio do CFG, uma vez que todos os dados pertinentes a uma tarefa podem ser extraídos dele. Porém, considerando que o CFG contém informações importantes para a análise do comportamento de uma tarefa, como é possível obtê-lo?

Como visto na Seção 4.1, a análise estática de uma tarefa corresponde apenas à fase *front-end* de um compilador. Há algumas ferramentas disponíveis para realizar essa análise inicial, por exemplo, *PyCParser* (Bendersky, 2015) é uma ferramenta desenvolvida em *Python* que implementa a gramática do C99.

PyCParser é uma ferramenta de análise léxica e sintática da linguagem C. Ela verifica se os símbolos utilizados em um código são aceitos pela linguagem e se a estrutura sintática está de acordo com a definição da gramática do C99. Ao final, é gerada uma estrutura de dados conhecida como Árvore Sintática Abstrata – AST (do inglês *Abstract Syntax Tree*).

O termo *abstrato* da AST refere-se ao fato da árvore representar em poucos detalhes a árvore de sintaxe formada. Em outras palavras, dentre as possíveis derivações definidas da gramática, a AST corresponde à árvore resultante a partir da análise do código de entrada. Todavia, informações como parênteses, pontuação, delimitadores, entre outros, são omitidas. Já informações como a linha em que um elemento está inserido no código fonte, podem ser extraídas. A Figura 4.5 apresenta uma AST simplificada onde apenas se identificam os elementos resultantes de cada derivação da gramática do C99.

A AST não corresponde ao CFG de fato. Como visto na Seção 4.1, um CFG é composto por nós – blocos ou conjunto de instruções em alto nível, e arestas – desvios no fluxo de execução de uma tarefa. Um bloco em C é formado por uma ou mais instruções de alto nível, já um nó na AST corresponde a um elemento derivado de uma regra gramatical. Desta forma, é necessário visitar os nós da AST e verificar quando um nó corresponde ao início da regra gramatical que define um *if* ou *while*. Por exemplo, ao se visitar um nó cuja regra refere-se ao condicional *if*, sabe-se que, a partir da condição verdadeira, há um filho composto por uma nova instrução em C a ser derivada. De forma

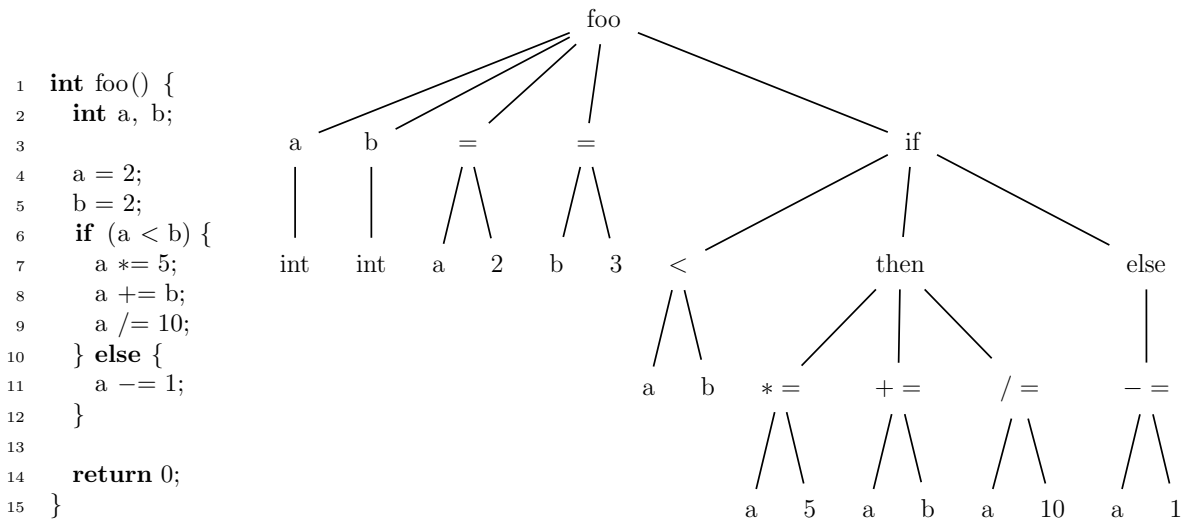


Figura 4.5: Código em C e a respectiva AST simplificada.

recíproca, há um outro filho composto por outra instrução em C a partir da condição falsa também a ser derivada.

Se durante a visita aos nós da AST forem identificadas as definições da estrutura de um *if* ou *while*, pode-se unir os elementos da AST a partir de cada condição verdadeira ou falsa.

Ainda no exemplo em que um nó da AST corresponde ao início da definição de um *if* na gramática, destaca-se que, enquanto os nós continuarem a ser visitados sem retornar a declaração que os originou, eles podem ser unidos em um mesmo nó do CFG. Portanto, cada nó do CFG é formado por um ou mais nós da AST, além disso a geração de um novo nó no CFG ocorre apenas quando houver a existência de uma regra condicional *if* ou *while* durante a visita da árvore. Entretanto, se não houver a presença de algum *if* ou *while* na AST, todos os nós dela corresponderão a apenas um nó do CFG (veja Figura 4.6).

Quanto a geração do CFG, destaca-se o último nó do grafo. Ao analisar a Figura 4.5, o último nó do grafo corresponde apenas à instrução de retorno da função. Porém, de acordo com a Seção 4.2, o custo do CFG baseia-se nas informações de *debug* do compilador, o qual sempre apresenta como último dado as instruções para liberar alguns registradores necessários para função. Esta liberação corresponde a mais um custo adicional em cada função, o que também é representado no CFG pelo nó 5, no caso da

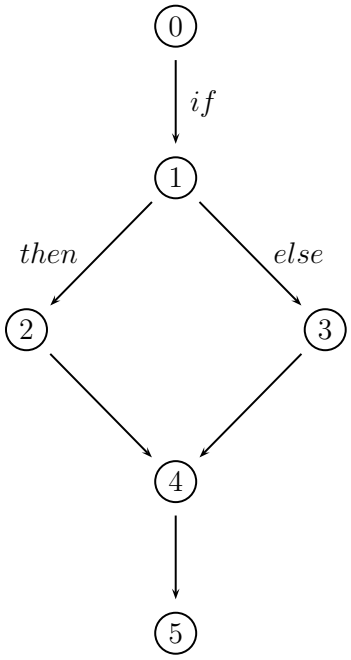


Figura 4.6: CFG da Figura 4.5.

Figura 4.6.

É importante destacar que o CFG resultante da visita aos nós da AST está armazenado em memória. Neste caso, optou-se pelo uso de um formato de arquivo baseado em *XML*, chamado de *GraphML*[1], para armazenar o CFG em disco.

O *GraphML*, assim como o *XML*, define *tags* para separar os elementos. As *tags* de um CFG referem-se aos nós e às arestas que estão relacionados a identificadores únicos. A principal diferença entre as *tags* de nós e de arestas está no fato destas terem como propriedade o nó de origem e destino.

Além disso, ambas as *tags* podem armazenar valores além do identificador do nó de origem e destino, como o WCEC do nó ou mesmo a linha inicial de um bloco (veja Figura 4.7).

A vantagem do CFG no formato *GraphML* é a possibilidade de integração com outras ferramentas para novos estudos, por exemplo, a *yEd*[2], ferramenta que tem como uma funcionalidade auxiliar a visualização dos dados.

[1]Para mais informações, acesse `http://graphml.graphdrawing.org/`

[2]*yEd Graph Editor:* `https://www.yworks.com/en/products/yfiles/yed/`

```
<graphml>
  <key attr.name=“start_line” attr.type=“int” for=“node” id=“k1”>
    <default>0</default>
  </key>
  <key attr.name=“wcec” attr.type=“int” for=“node” id=“k7”>
    <default>0</default>
  </key>
  <key attr.name=“rwcec” attr.type=“int” for=“node” id=“k8”>
    <default>0</default>
  </key>
  <graph>
    <node id=“g1n0”>
      <data key=“k1”>2</data>
      <data key=“k7”>21</data>
      <data key=“k8”>115</data>
    </node>
    <node id=“g1n1”>
      <data key=“k1”>7</data>
      <data key=“k7”>16</data>
      <data key=“k8”>94</data>
    </node>
    <node id=“g1n2”>
      <data key=“k1”>8</data>
      <data key=“k7”>62</data>
      <data key=“k8”>78</data>
    </node>
    <node id=“g1n3”>
      <data key=“k1”>15</data>
      <data key=“k7”>3</data>
      <data key=“k8”>16</data>
    </node>
    <node id=“g1n4”>
      <data key=“k1”>0</data>
      <data key=“k7”>13</data>
      <data key=“k8”>13</data>
    </node>
    <node id=“g1n5”>
      <data key=“k1”>12</data>
      <data key=“k7”>11</data>
      <data key=“k8”>27</data>
    </node>
    <edge id=“g1e0” source=“g1n3” target=“g1n4”>
      <data key=“k8”>13</data>
    </edge>
    <edge id=“g1e1” source=“g1n2” target=“g1n3”>
      <data key=“k8”>16</data>
    </edge>
    <edge id=“g1e2” source=“g1n1” target=“g1n2”>
      <data key=“k8”>78</data>
    </edge>
    <edge id=“g1e3” source=“g1n5” target=“g1n3”>
      <data key=“k8”>16</data>
    </edge>
    <edge id=“g1e4” source=“g1n1” target=“g1n5”>
      <data key=“k8”>27</data>
    </edge>
    <edge id=“g1e5” source=“g1n0” target=“g1n1”>
      <data key=“k8”>94</data>
    </edge>
  </graph>
</graphml>
```

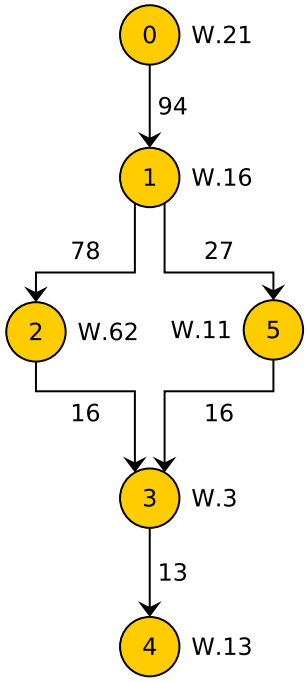


Figura 4.7: Visualização do *GraphML* e *yEd* do código C na Figura 4.5.

4.5 Cálculo da nova frequência

A transformação de um código para outro com maior eficiência energética consiste na identificação de pontos onde, durante a execução de uma tarefa, alteram-se a frequência e tensão do processador, a fim de reduzir o consumo de energia.

Porém, quais pontos são considerados candidatos para alterar a frequência e tensão?

Como determinar estes novos valores sem infringir a restrição temporal definida?

Nas seções anteriores, foi definida uma metodologia que permite abstrair o comportamento de uma tarefa em um grafo de fluxo de controle, e também determinar o pior caso do fluxo de execução.

Nesse sentido, se, durante a execução de uma tarefa, o fluxo desviar do pior caminho, sabe-se, então, que o processador executará menos ciclos. Estes desvios do fluxo de execução determinam as arestas (pontos) com SEC do CFG onde a frequência e tensão podem ser alteradas. Há dois tipos de arestas onde é possível alterar a frequência: tipo-B e tipo-L, conforme o estudo apresentado em Shin *et al.* (2001).

Definição Dado um $CFG = (N, E)$, onde N é o conjunto de nós e E de arestas, tem-se $u, v, w \in N$ e $(u, v), (u, w) \in E$. Seja u o nó condicional de uma estrutura de seleção, v o fluxo do *then* e w o *else*, define-se uma aresta do tipo-B como:

$$\textit{tipo-B}((u,v),(u,w)) = \begin{cases} (u,v), & RWCEC(u,v) \leq RWCEC(u,w) \\ (u,w), & RWCEC(u,v) > RWCEC(u,w) \end{cases}$$

Em outras palavras, as arestas do tipo-B (*branch*) são identificadas a partir dos nós condicionais do CFG. Cada nó condicional possui uma parte *then* – quando a condição é satisfeita, e *else* – caso contrário.

A aresta do tipo-B corresponde àquela a definir um fluxo onde há um SEC_B (SEC do tipo-B). Por exemplo, na Figura 4.3, B3 (*then*) apresenta um RWCEC menor que o B4 (*else*), neste caso obtém-se um SEC_B de 133 ciclos e classifica-se a aresta $B2 \rightarrow B3$ como do tipo-B (veja Figura 4.8). Este é um ponto importante, pois, ao analisar o CFG da Figura 4.8, verifica-se que, em qualquer execução do grafo, o SEC_B entre os nós B3 e B4 é sempre de 133 ciclos. Se este valor é igual em qualquer execução, por que não definir uma única frequência cuja adoção será sempre ao executar B3?

O problema de adotar uma frequência fixa no nó de destino cuja aresta tem SEC constante, é que a execução do nó de origem pode ocorrer com diferentes frequências, as quais podem ser maior ou menor à definida no nó de destino. Se o nó de origem executar a uma frequência maior do que a adotada no nó de destino, então o consumo de energia será menor. Contudo, se o nó de origem executar com uma frequência menor do que a adotada no de destino, então o consumo será maior.

Por exemplo, considera-se um cenário com as frequências 100MHz, 80MHz e 50MHz. Assume-se, ainda, que os 133 ciclos da aresta $B2 \rightarrow B3$ na Figura 4.3 resulte em adotar sempre 80MHz. Se o nó B2 for executado a 100MHz, a tarefa trocará a frequência para 80MHz ao direcionar-se a B3, o que reduzirá o consumo de energia. Porém, se o nó B2 for executado a 50MH, a frequência será de 80MHz ao executar B3 e, logo, o consumo de energia será maior.

Entretanto, se em vez de utilizar uma frequência fixa, adotar-se a proporção de quantas vezes o RWCEC da aresta do tipo-B, $B2 \rightarrow B3$, é menor ao RWCEC de $B2 \rightarrow B4$, é possível evitar o problema do nó de origem executar a uma frequência menor e o de destino com uma maior.

De acordo com o exemplo anterior, a aresta $B2 \rightarrow B3$ possui um RWCEC igual a 14 ciclos, enquanto $B2 \rightarrow B4$ um RWCEC igual a 147 ciclos. O RWCEC da aresta $B2 \rightarrow B3$ é aproximadamente 9.52% menor ao de $B2 \rightarrow B4$. Em um caso ideal, a frequência de B3 deveria ser 9.52% menor à adotada em B4 e o consumo de energia sempre seria menor, independente da frequência em B2.

Por exemplo, se B2 estiver em execução a 100MHz, ao adotar B3 e aplicar a proporção na frequência, B3 deverá executar com 9.52MHz. Por outro lado, se B2 estiver com 50MHz, a execução de B3 será com 4.76MHz.

Shin *et al.* (2001) definem quantas vezes o RWCEC de um nó é menor ao de outro com uma **taxa de atualização da velocidade** da frequência. No caso, esta taxa corresponde a: $RWCEC(B3) \,/\, RWCEC(B4)$, e representada por $speedRate(B2, B3)$, ou seja, o valor da taxa de atualização da frequência atual quando o fluxo de execução direciona-se de B2 para B3.

Com base no RWCEC e na identificação dos nós condicionais, Shin *et al.* (2001) observam também um custo adicional de ciclos a serem consumidos a fim de alterar a frequência e tensão, chamado de *overhead*. É necessária a representação deste valor, pois o cálculo da nova frequência e as instruções adicionais para trocá-la são informações adicionadas ao código original de uma tarefa, ou seja, representam um custo a mais na execução. Assim, generaliza-se o cálculo da taxa de atualização e da nova frequência das arestas do tipo-B conforme a Equação (4.1).

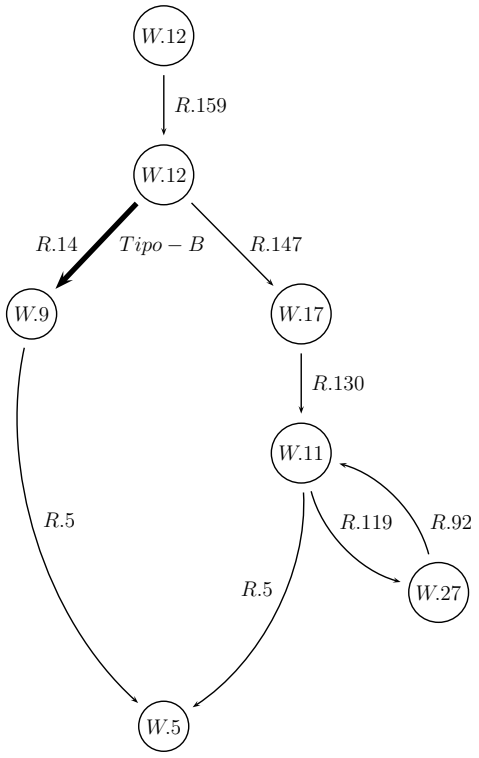


Figura 4.8: Identificação de uma aresta tipo-B.

$$
\begin{aligned}
speedRate_B(B_{cond}, B_i) &= \frac{RWCEC(B_i)}{RWCEC(succ_{worst}(B_{cond})) \; - \; overhead(B)} \\
newFreq(B_{cond}, B_i) &= curFreq(B_{cond}) \; * \; speedRate_B(B_{cond}, B_i)
\end{aligned}
\tag{4.1}
$$

onde $speedRate_B(B_{cond}, B_i)$ refere-se a taxa de atualização das arestas B_{cond} (bloco condicional) para B_i do tipo-B, $i \in \{then, else\}$; $RWCEC(B_i)$ representa quantos ciclos ainda são necessários para o término da execução da tarefa a partir do bloco B_i; $succ_{worst}(B_{cond})$ corresponde ao sucessor de B_{cond} com o maior RWCEC, ou seja, B_{then} ou B_{else}; $overhead(B)$ é o valor necessário em ciclos para a computação e troca da nova frequência e tensão a partir de uma aresta do tipo-B; $curFreq(B_{cond})$ a frequência atual utilizada na execução de B_{cond}, e $newFreq(B_{cond}, B_i)$, ao valor da nova frequência. Um exemplo prático da Equação (4.1) pode ser verificado com os dados da Figura 4.3.

Calcula-se a taxa de atualização para os nós B2 e B3, ou seja, $speedRate_B(B2, B3)$.

O sucessor de B2 com maior RWCEC corresponde ao nó B4. Assim, $RWCEC_{B3} = 14$ e $RWCEC_{B4} = 147$. Com *overhead* igual a zero, tem-se $speedRate_B(B2, B3) = 14/147 \approx 0.095$. Dada a proporção de quantas vezes o $RWCEC_{B3}$ é menor ao de $RWCEC_{B4}$, qualquer caminho no CFG que utilizar a aresta $B2 \rightarrow B3$, calculará a nova frequência como: $newFreq(B2, B3) = curFreq(B2) * 0.095$. Uma ideia semelhante aplica-se às arestas do tipo-L.

Definição Dado um $CFG = (N, E)$, onde N é o conjunto de nós e E de arestas, tem-se $u, v, w \in N$ e $(u, v), (v, u), (u, w) \in E$. Seja u o nó condicional do laço e $L = \{(u, v), (v, u)\}$ o conjunto de arestas do laço, (u, w) é uma aresta do tipo-L se:

$$\textit{tipo-L}((u, w)) = \begin{cases} 1, & (u, w) \notin L \\ 0, & (u, w) \in L \end{cases}$$

Em outras palavras, as arestas do tipo-L são identificadas sempre como o nó de saída de um laço, ao contrário daquelas do tipo-B, nas quais tanto o *then* quanto o *else* são possíveis candidatos a ter inserção de código (Shin *et al.*, 2001). A utilização da aresta de saída de um laço deve-se ao fato de só haver reduções no número de ciclos consumidos por um laço quando o mesmo, durante a execução, possui um número de iterações menor do que o pior caso.

Por exemplo, na Figura 4.9, a estrutura de repetição definida possui no máximo três iterações. Como a quantidade de ciclos consumidos em uma única iteração é igual a 38, admite-se que, no pior caso, 137 ciclos serão executados (o nó condicional é executado outra vez para encerrar o laço). Contudo, se, durante a execução da tarefa, apenas uma iteração for realizada, um SEC_L (SEC do tipo-L) é identificado. Assim, algumas informações devem ser definidas antes do cálculo da taxa de atualização da nova frequência ao sair de um laço.

O valor de SEC_L é o principal fator para determinar quantos ciclos deixaram de ser consumidos na execução de um laço. Este valor é calculado segundo a Equação (4.2).

$$SEC_L = WCEC_{once}(loop) * (N_{worst}(loop) - N_{exec}(loop)) \tag{4.2}$$

onde $WCEC_{once}(loop)$ corresponde ao número de ciclos necessários para executar o laço apenas uma vez no pior caso; $N_{worst}(loop)$ refere-se ao número de iterações do pior caso,

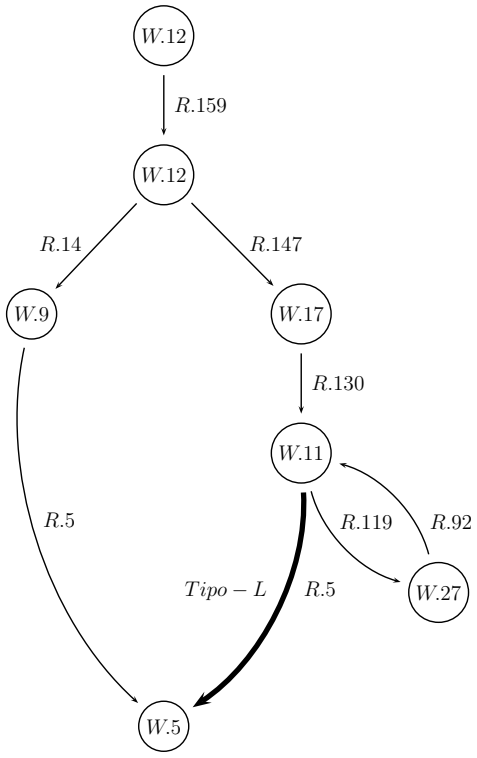


Figura 4.9: Identificação de uma aresta tipo-L.

sendo que este valor deve ser informado pelo usuário, pois não há como determinar este número máximo de execuções – por este motivo a definição do padrão *//@LOOP MAX 3* descrito anteriormente. Já $N_{exec}(loop)$ representa o número de iterações do laço durante a execução da tarefa. Assim, se o valor de $N_{exec}(loop)$ for igual a $N_{worst}(loop)$, o pior caso do laço foi executado. E, da mesma forma que as arestas do tipo-B, há um *overhead* associado aos cálculos e troca de frequência das arestas do tipo-L. Logo, Shin *et al.* (2001) definem a taxa de atualização conforme a Equação (4.3).

$$\begin{aligned} speedRate_L(B_{cond}, B_j) &= \frac{RWCEC(B_j)}{RWCEC(B_j) \ + \ SEC_L \ - \ overhead(L)} \\ newFreq(B_{cond}, B_j) &= curFreq(B_{cond}) \ * \ speedRate_L(B_{cond}, B_j) \end{aligned} \tag{4.3}$$

Na Figura 4.3, por exemplo, o laço formado pelos nós B5 e B6 possui no máximo três iterações, com um custo de 49 ciclos em uma única iteração (uma execução do

laço mais outra do nó condicional B5). Em um cenário onde apenas uma iteração foi realizada e não há *overheads*, tem-se $SEC_L = 38 * (3 - 1) = 76$ ciclos que deixaram de ser executados. Com o nó de saída B7, cujo RWCEC é igual a 5 ciclos, $speedRate_L(B5, B7) = 5/(5 + 76 - 0) = 5/81 \approx 0.062$. Assim, para este em que onde ocorreu a execução de apenas uma iteração do laço, a nova frequência é dada por: $newFreq(B5, B7) = curFreq(B5) * speedRate_L(B5, B7) = curFreq(B5) * 0.062$.

Vale ressaltar que o cálculo da nova frequência, tanto das arestas do tipo-L quanto do tipo-B, utiliza informações extraídas a partir do CFG. Além disso, só ocorrerá efetivamente a troca de frequência se $speedRate_B(B_{cond}, B_i) < 1$ ou $speedRate_L(B_{cond}, B_j) < 1$. Do contrário, significa que, devido ao *overhead* ou ao baixo valor de SEC_L, a proporção entre o RWCEC dos nós em questão manteve-se igual ou superior, o que não resulta em otimizações energéticas.

O cálculo definido por Shin *et al.* (2001) assume a utilização da frequência ideal, porém esta nem sempre estará disponível no conjunto discreto de frequências do processador. Por isto, esta dissertação adota sempre como a nova frequência o valor no conjunto de frequências do processador que for igual ou maior a frequência ideal calculada. A adoção de um valor menor ao ideal implicaria o descumprimento do *deadline*.

Além disso, a identificação das arestas tipo-B e tipo-L, assim como os cálculos das novas frequências, é a principal informação para a transformação de um código DVFS-*Unaware* para DVFS-*Aware*. O nó de destino destas arestas corresponde ao ponto onde os cálculos da nova frequência serão inseridos, assim, a produção do novo código estará concluída. Apesar da apresentação do cálculo das novas frequências, o valor inicial utilizado pelo processador ainda não foi definido.

4.6 Determinação da frequência inicial

Shin *et al.* (2001) propõe o cálculo da frequência inicial a partir da razão do RWCEC inicial da tarefa pelo *deadline*. Este valor, embora válido, já que o resultado apresentará o valor ideal para a execução da tarefa ser finalizada sem infringir a restrição temporal, não considera as limitações de frequências disponíveis pelos processadores. Além delas serem discretas na prática, poucas são as que realmente estão acessíveis para os usuários.

O contexto do trabalho proposto por Shin *et al.* (2001) considera sistemas mono

tarefas. Ao estender este trabalho para sistemas de múltiplas tarefas, verifica-se que os autores não consideraram alguns pontos relevantes no cálculo da nova frequência, como a interferência de uma tarefa nas demais.

Por exemplo, relação de dependência, compartilhamento de recursos, prioridades, são situações que Shin *et al.* (2001) não analisam. Assim, aplicar esta abordagem em um contexto de múltiplas tarefas nos sistemas de tempo real não é viável, pois pode levar ao descumprimento das restrições temporais de um conjunto de tarefas.

Valentin (2010), entretanto, apresenta um estudo de como determinar as frequências iniciais ótimas para um conjunto de tarefas, a fim de explorar ao máximo o tempo ocioso com otimizações de energia e cumprimento das metas temporais. O trabalho proposto por Valentin (2010) será resumidamente descrito, pois é a base do cálculo da frequência inicial a ser adotada durante a execução de uma tarefa DVFS-*Aware*.

Considere um processador com suporte à DVFS e X frequências disponíveis. O conjunto de todas as frequências possíveis é dado por $F = \{f_i \mid f_i$ *é uma frequência em Hz e* $1 \leq i \leq X\}$. Além disso, assuma M como um modelo de N tarefas com prioridade fixa.

Seja β um conjunto cujos elementos são N-tuplas, $K \in \beta$, com $K = (f_1, f_2, f_3, \ldots, f_n)$, onde $f_i \in F$. K representa uma possível configuração das frequências do processador a serem atribuídas às tarefas do sistema, assim, f_1 é definido para τ_1, f_2, para τ_2 e, assim, sucessivamente.

Em outras palavras, K é uma possível combinação das frequências disponíveis às tarefas. Logo, β corresponde a uma enumeração de todas as combinações para executar o modelo de tarefas M. Portanto, existem X^N combinações possíveis em β, um problema com custo exponencial.

Seja $C_i(f)$ o tempo de computação da tarefa i em função da frequência f: $C_i(f) = RWCEC(\tau_i) \,/\, f$, e D_i o *deadline* da tarefa i, o objetivo principal de Valentin (2010) é determinar um $K \in \beta$ tal que proporcione ao sistema o menor consumo de energia:

$$Minimizar\{o(M) = \sum_{i=0}^{N}(D_i - R_i(f_i))\} \tag{4.4}$$

Por tratar-se de sistemas de tempo real, o subconjunto K ótimo só será válido caso o modelo de tarefas seja escalonável. Esta validação ocorre a partir da seguinte equação:

$$R_i(K) \leq D_i, \ \forall i < N \tag{4.5}$$

onde $R_i(K)$ é o tempo de resposta da tarefa i em função do subconjunto de frequências em K. Este valor ainda é definido como: $R_i(K) = I_i(K) + J_i$, tal que $I_i(K)$ é a interferência sofrida pela tarefa i em função de K e J_i o *release jitter* da tarefa i – o pior tempo de liberação da tarefa i.

A Equação (2.11) da interferência sofrida por uma tarefa, pode ser reescrita em função do conjunto de frequências:

$$I_i(K)^{n+1} = C_i(k_i) + B_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{I_i^n(K) + J_i}{P_j} \right\rceil * C_j(k_i) \right) \tag{4.6}$$

Com base nos dados apresentados, Valentin (2010) propõe um algoritmo para gerar uma árvore de busca a partir da enumeração de todas as possíveis combinações de frequências entre as tarefas do sistema. Esta árvore é, então, percorrida utilizando-se busca em profundidade. Como cada nó corresponde a uma frequência a ser adotada por uma tarefa, a ordem de visitação destes nós ocorre de modo decrescente onde as maiores frequências são primeiramente analisadas e, em seguida, as de menor valor. Assim, conjuntos a gerarem um tempo de computação maior são construídos por último.

Na abordagem de Valentin (2010), nem todas as enumerações são feitas, pois reduções (podas na árvore) são realizadas a fim de diminuir o número de combinações a serem construídas. Foram definidos três tipos de podas: o primeiro exclui possíveis valores de K que não satisfazem o teste de escalonabilidade; o segundo determina um ponto de partida onde todos os processos executam com a mesma frequência; já o terceiro, elimina combinações de frequências considerando a carga de trabalho do sistema e as interferências (preempções, bloqueio de recursos) sujeitas a cada tarefa.

O algoritmo apresentado produz como resultado as frequências iniciais ótimas para utilização no conjunto de tarefas dado como entrada. Outra informação que ele também fornece como saída é o tempo de resposta das tarefas utilizadas como entrada.

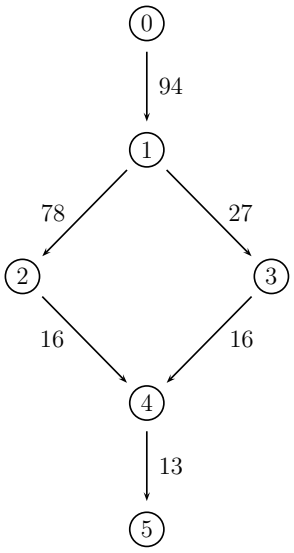
4.7 Transformação de código: DVFS-*Unaware* para DVFS-*Aware*

A transformação de um código DVFS-*Unaware* em DVFS-*Aware* consiste em inserir instruções de código encarregadas de calcular e aplicar a nova frequência. Esta seção visa a descrição deste processo.

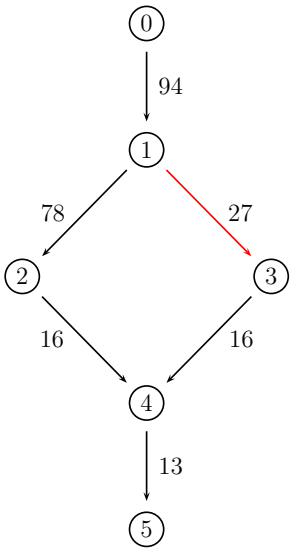
Como apresentado anteriormente, extrai-se o CFG da tarefa em estudo. Depois, identifica-se qual o pior caminho neste mesmo grafo. A partir desta informação, ao percorrer o CFG, procura-se identificar as arestas do tipo-B e tipo-L, conforme definido na Seção 4.5. Quando qualquer uma dessas arestas for identificada, insere-se o código para calcular a nova frequência, o que já representa um *overheads* ao custo total da tarefa.

```
1  int foo() {
2    int a, b;
3
4    a = 2;
5    b = 2;
6    if (a < b) {
7      a *= 5;
8      a += b;
9      a /= 10;
10   } else {
11     a -= 1;
12   }
13
14   return 0;
15 }
```

(a) DVFS-*Unaware*.



(b) CFG.



(c) Aresta tipo-B.

Figura 4.10: Identificação da aresta tipo-B.

A principal vantagem de armazenar informações de alto nível no CFG é que cada nó pode conter dados importantes como as linhas iniciais e finais do bloco de instruções que compõem o próprio nó.

Portanto, ao encontrar uma aresta tipo-B ou tipo-L, sabe-se que na linha inicial do

```
1  /*** auto generate DVFS code ***/
2  #include "cfg_wcec.h"
3  __cfg_edge_type __cfg_type;
4  float __cfg_rwcec_bi;
5  float __cfg_rwcec_bj;
6  int __cfg_loop_max_iter;
7
8  int foo() {
9    int a, b;
10
11   a = 2;
12   b = 2;
13
14   if (a < b) {
15     a *= 5;
16     a += b;
17     a /= 10;
18   } else {
19     /*** auto generate DVFS code ***/
20     __cfg_type = __CFG_TYPE_B;
21     __cfg_rwcec_bi = 78;
22     __cfg_rwcec_bj = 27;
23     __cfg_change_freq(&__cfg_type, __cfg_rwcec_bi, __cfg_rwcec_bj, 0, 0);
24     a -= 1;
25   }
26
27   return 0;
28 }
```

Figura 4.11: Código DVFS-*Aware*.

nó de destino, deve-se inserir o código para o cálculo da nova frequência. Este processo de adicionar um código com informações sobre possíveis otimizações de energia é o que se considera nesta dissertação como de transformação de um código DVFS-*Unaware* para DVFS-*Aware*.

A Figura 4.10 contém um código em C e o respectivo CFG. Ao analisar este grafo, verifica-se que a aresta $1 \rightarrow 3$ é do tipo-B, logo, adiciona-se no respectivo nó de destino informações do cálculo da nova frequência e aplicação desta, se possível. A Figura 4.11 visa a exemplificação desta etapa.

Ressaltam-se que informações como o custo de execução de cada nó e quantos ciclos faltam serem executados até o término do caminho, foram todas extraídas do próprio CFG da tarefa. Já a adição da biblioteca *cfg_wcec.h* faz parte da transformação de código.

Com o fim de deixar o processo do cálculo da nova frequência e possível troca mais

claro e didático, criou-se uma biblioteca com os cálculos da Seção 4.5, além da função que deve realizar a chamada ao processador para a troca da frequência. Comentários também foram adicionados para melhorar a compreensão do código. Veja o Apêndice A para mais detalhes.

A inserção de novas instruções ao código original é o que permite ser possível a troca da frequência e tensão de uma tarefa. Ao identificar as arestas do tipo-B e tipo-L, insere-se o novo código na linha inicial dos respectivos nós de destino. Esta etapa de identificação e geração automática do código é útil por afastar dos desenvolvedores a preocupação em se ter um conhecimento prévio sobre baixo consumo de energia.

4.8 Novos Valores de *Deadline* e WCEC

A Figura 4.12 apresenta como é possível relacionar cada tópico discutido neste capítulo.

(i) Na **Geração do CFG e Pior Caminho**, utiliza-se o modelo de tarefas, dado na forma de códigos DVFS-*Unaware*, como entrada. Em seguida, identificam-se as estruturas definidas na Seção 4.1, a fim de construir o CFG. Depois, calcula-se o custo computacional de cada nó com base no código intermediário, e determina-se o pior caminho no fluxo de execução do grafo.

(ii) Com as **Informações de Escalonabilidade I**, procura-se verificar se o modelo de tarefas é válido. Se sim, obtém-se o conjunto de frequências iniciais ótimas para aplicar no modelo de tarefas com o objetivo de reduzir o consumo de energia e o tempo ocioso (Valentin, 2010). Para isto, é necessário utilizar como entrada o WCEC do CFG. A saída desta fase informa primeiro se o modelo de tarefas é escalonável. Se não for, não há o porquê de continuar a metodologia. Se for, então se conhece o conjunto ótimo de frequências iniciais.

(iii) A próxima etapa, **Identificação de Arestas e Inserção de Código DVFS**, consiste em identificar os desvios no fluxo de execução do pior caminho, por meio das arestas do tipo-B e tipo-L. Os nós de destino destas arestas correspondem ao nó que deve conter o cálculo e troca da nova frequência. Com a identificação dos nós, inserem-se as novas instruções, cujo código resultante será DVFS-*Aware*. Contudo, este novo código ainda está sujeito às definições (*deadline* e WCEC da tarefa) do modelo de tarefas

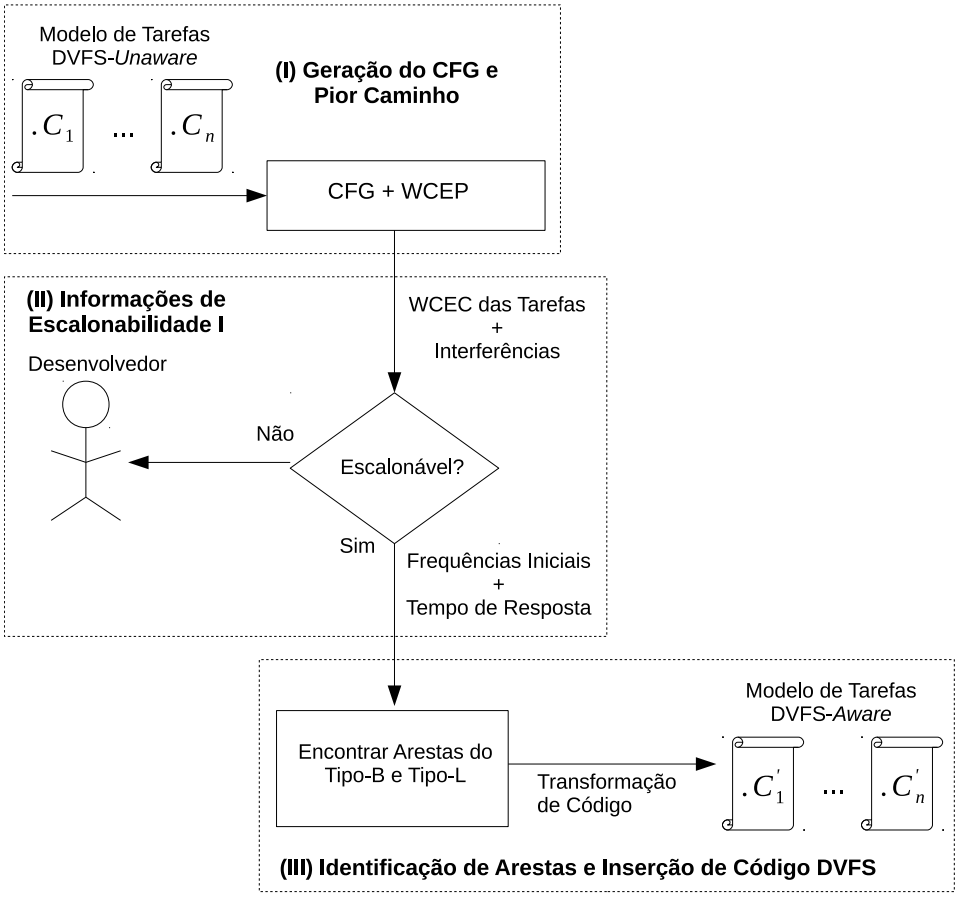


Figura 4.12: Diagrama do método proposto.

utilizado como entrada? Os códigos DVFS-*Aware* ainda são escalonáveis?

A troca de frequência, segundo os cálculos definidos por Shin *et al.* (2001), procura utilizar a frequência ideal para a execução da tarefa terminar exatamente no *deadline*. Por outro lado, Valentin (2010) apresenta, no cálculo das frequências iniciais, o tempo de resposta que cada tarefa deve ter para cumprir a meta temporal mesmo com as interferências de um ambiente de múltiplas tarefas. Um modelo de tarefas pode deixar de ser escalonável caso as trocas de frequência durante a execução de uma tarefa preocupem-se apenas com o *deadline* desta e não com o tempo de resposta calculado, pois o valor do tempo de resposta já inclui todas as interferências externas possíveis.

Assim, para garantir que o sistema continue escalonável e trocas de frequência possam ocorrer durante a execução, define-se um novo valor para o *deadline*, que corresponde

exatamente ao tempo de resposta.

Com o *deadline* **igual ao tempo de resposta**, inicialmente se mantém o sistema escalonável, pois os valores calculados por Valentin (2010) continuam válidos. Segundo, ainda é possível haver trocas de frequência, porém, desta vez, a adoção de frequências menores visa o término da execução exatamente no tempo de resposta e não no *deadline*.

Vale ressaltar que mesmo as equações definidas na Seção 4.5 calculem a frequência ideal, a proposta desta dissertação assume a utilização de uma frequência maior ou igual ao valor calculado no conjunto discreto de frequências disponíveis pelo processador. Todavia, o tempo não é a única variável com uma condição especial de uso, o WCEC também deve ser revisto.

Shin *et al.* (2001) identificam arestas do tipo-B e tipo-L para inserir um novo código responsável pelo cálculo da nova frequência. Arestas do tipo-B correspondem a desvios no fluxo de execução do pior caminho. As do tipo-L, por outro lado, são aquelas correspondentes à saída de um laço. A inserção de novas instruções devido a qualquer um destes dois tipos de arestas, acarreta um *overhead* ao caminho do CFG em execução.

Se o pior caminho do CFG incluir uma aresta do tipo-L, o que deve ser feito é inserir novas instruções de código para calcular a nova frequência, caso o laço tenha executado um número de iterações menor que o máximo. Ora, como esta aresta está no pior caminho e instruções referentes à troca de frequência são inseridas, tem-se um *overhead* no pior caminho. Logo, com a aplicação desta dissertação, o custo de execução real do pior caminho de uma tarefa será o WCEC do caminho mais o *overhead*. A Figura 4.13 procura exemplificar esta situação.

O algoritmo na Figura 4.13 determina os 10 elementos na sequência de Fibonacci e os armazena em um vetor. Este é um código simples com apenas um único laço. Com a proposta desta dissertação, a aresta de saída do laço se refere à última linha da função. Este é o ponto a ser inserido o cálculo da nova frequência.

Inicialmente, o algoritmo tinha um WCEC igual a x. Após a inserção do novo código – ou seja, um *overhead*, o custo final de execução em termos de ciclos do processador corresponde ao $x + overhead$. Estes custos adicionais refletem também no consumo de energia, pois agora ele será maior, inclusive no pior caso. Mesmo assim, as trocas de frequência para valores menores durante a execução de um caminho do CFG, podem

```
1  void fib(int *arr) {
2    int i, len;
3
4    len = 10;
5    arr[0] = 1;
6    arr[1] = 1;
7    i = 2;
8    while (i < len) {
9      arr[i] = arr[i - 1] + arr[i - 2];
10     i++;
11   }
12   // type-L edge
13 }
```

Figura 4.13: Código com aresta do tipo-L.

reduzir o consumo de energia de tal modo a compensar os custos adicionais dos *overheads*.

A Figura 4.13 apresentou que a escalonabilidade do modelo de tarefas, com os códigos DVFS-*Unaware*, deve ser verificada a fim de se obter as frequências iniciais ótimas e o tempo de resposta. Este teste se baseia no WCEC das tarefas.

Entretanto, com a transformação de código para DVFS-*Aware*, o WCEC utilizado antes pode ter tido a adição de um ou mais *overheads*. Isto influencia diretamente na escalonabilidade do modelo de tarefas, nas frequências iniciais e no tempo de resposta de cada tarefa já que o WCEC pode ter um valor superior ao inicial.

Desta forma, com o fim de evitar que o modelo de tarefas, antes escalonável, deixe de ser válido devido à aplicação desta dissertação, propõe-se uma nova verificação da escalonabilidade das tarefas, mas agora com código DVFS-Aware. Assim, a etapa final (iv) da Figura 4.13, **Informações de Escalonabilidade II**, agora é a entrada para uma nova verificação da escalonabilidade do sistema. O resultado, se escalonável, será o mesmo modelo de tarefas com a transformação de código, porém com os valores de frequências iniciais e tempos de resposta atualizados, conforme a Figura 4.14.

É interessante destacar os dois testes de escalonabilidade na Figura 4.14. Por que não utilizar apenas o teste final já que o importante é verificar se o modelo de tarefas com códigos DVFS-*Aware* é escalonável?

Não só realizar a transformação de código e manter o modelo de tarefas escalonável são os pontos principais desta dissertação, mas também otimizar o consumo de energia.

Se houvesse apenas o teste de escalonabilidade final da Figura 4.14, e o modelo de

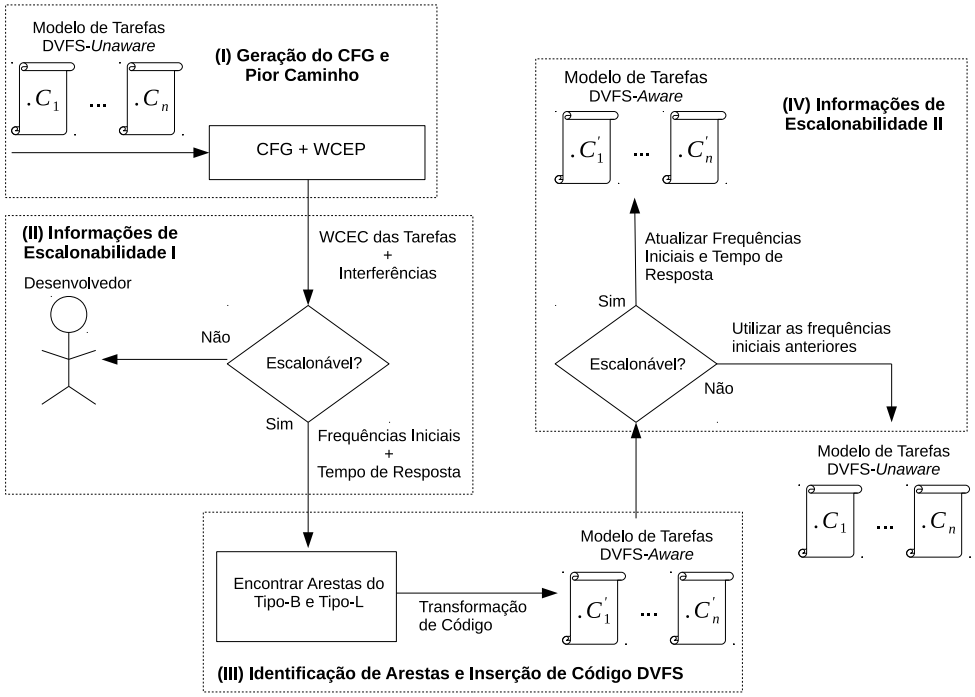


Figura 4.14: Diagrama do método proposto atualizado.

tarefas resultante não fosse escalonável, então não haveria otimização no consumo de energia. Por outro lado, se houvesse a verificação do primeiro teste de escalonabilidade, o modelo de tarefas original fosse escalonável e o segundo teste fosse negativo, então ainda seria possível ter otimização no consumo de energia se as tarefas adotassem as frequências iniciais ótimas do primeiro teste. Todavia, sem as informações DVFS-Aware presentes nas tarefas.

As quatro principais etapas para transformação de código (gerar o CFG e encontrar o pior caso; verificar se o modelo de tarefas é escalonável; se for, identificar as arestas tipo-B e tipo-L, e gerar o código DVFS-Aware; garantir que a modificação do modelo de tarefas não influenciou a escalonabilidade) definem uma metodologia para otimizar o consumo de energia.

Esta abordagem considera não só a execução *inter-tarefas*, mas também uma análise *intra-tarefa*, a fim de diminuir o consumo de energia quando se conhecem informações únicas do comportamento de cada tarefa. A implementação desta proposta está presente na ferramenta cfg (Pinheiro, 2015).

Capítulo 5

Resultados Experimentais

O método proposto consiste em unir dois estudos de abordagens bem diferentes sobre a análise estática do comportamento das tarefas de um sistema.

O primeiro se baseia no trabalho desenvolvido por Shin *et al.* (2001) que se preocupa com a execução intra-tarefa, ou seja, apenas em determinar a frequência ótima a ser usada durante a execução de uma única tarefa, de modo a otimizar o tempo de folga. O segundo trabalho, proposto por Valentin (2010), também procura otimizar o tempo de folga, todavia se preocupa com a execução de múltiplas tarefas e as interferências a que cada uma está sujeita, mas sem se atentar com caminhos de execução abaixo do pior caso.

Este capítulo apresenta dois estudos de casos para medir se esta dissertação proporcionou redução no consumo de energia. A análise destes estudos consiste em simular a execução de um modelo de tarefas dado como entrada.

Cada estudo de caso compara o consumo de energia do método proposto com (i) o da maior frequência disponível no sistema e (ii) do trabalho proposto por Valentin (2010). O trabalho de Shin *et al.* (2001) não é utilizado para comparação, pois os autores não consideram as interferências que uma tarefa possa sofrer em ambientes de múltiplas tarefas.

Antes de apresentar os resultados, primeiramente se define o ambiente de experimentação. Em seguida, analisam-se dois estudos de caso.

5.1 Ambiente de Experimentação

Esta seção apresenta as principais definições do ambiente de experimentação para os estudos de caso.

5.1.1 Caminhos de Execução

Conforme descrito na Seção 4.3, ao explorar o CFG, é possível encontrar todos os possíveis caminhos de execução de uma tarefa, inclusive determinar o pior deles. Conhecer cada caminho é útil, pois, de acordo com a característica de cada tarefa, pode-se verificar se o fluxo de execução percorre um caminho diferente do pior.

Adotou-se três possíveis caminhos de execução nos estudos de caso: o pior, o mediano, e o melhor aproximado.

Para encontrar o **pior caminho** de execução, de posse do CFG ponderado com os custos de execução em ciclos de cada nó, basta realizar um algoritmo de busca em profundidade para determinar o caminho onde ocorre a execução do maior número de ciclos. Desta forma, determinar o conjunto P de caminhos possíveis do CFG:

$$P = \{p_i \,|\, ec(p_1) < ec(p_2) < \ldots < ec(p_n)\}$$

onde $ec(p_i)$ é o custo em ciclos de *clock* para executar o caminho p_i.

Em outras palavras, o conjunto P consiste de todos os caminhos possíveis de uma tarefa, ordenados de modo crescente pelo custo em ciclos de *clock* da execução de cada um. Desta forma, p_n corresponde ao pior caminho do fluxo de execução e o WCEC da tarefa é dado por $WCEC = ec(p_n)$.

Já o **caminho mediano** é dado por:

$$p_m = P(1 + len(P)/2)$$

Onde $len(P)$ corresponde ao número de caminhos existentes na tarefa em estudo, p_m representa um caminho pertencente ao conjunto P na posição $1 + len(P)/2$. Assim, se $len(P)$ for ímpar, a mediana corresponde ao elemento que divide o caminho em duas partes. Se $len(P)$ for par, há dois valores para representar a mediana, o último elemento da primeira parte da divisão dos caminhos – p_k, ou o primeiro elemento da segunda parte – p_z. Adota-se como mediana o caminho referente a p_z, pois $ec(p_z) > ec(p_k)$. O último caminho utilizado nos estudos de caso é o melhor aproximado.

O melhor caminho de P corresponde ao elemento do conjunto que apresentar o menor custo computacional, no caso, p_1. Por exemplo, em termos do comportamento de uma

tarefa, ao encontrar um desvio condicional de uma estrutura de seleção *if*, adota-se o nó de menor custo na continuação do caminho.

Já ao se considerar estruturas de iteração, o melhor caso é o resultado da primeira execução da condição do laço ser falso, assim, nenhuma iteração ocorreria. Em resumo, o melhor caminho é aquele que não tem nenhuma iteração de um laço, além de adotar sempre o menor custo de uma estrutura de seleção.

O problema de adotar o melhor caminho é que o custo computacional para execução é baixo ao se comparar com o pior caso. Para isto, despreza-se esse valor e se adota o melhor caminho aproximado.

Define-se o **melhor caminho aproximado** com um custo computacional igual ou o mais próximo possível de 15% do WCEC da tarefa. Então:

$$p_{ab} = p_j,\ ec(p_1) \leq ec(p_2) \leq \ldots \leq ec(p_j) \leq WCEC * 15\%$$

A adoção de 15% deve-se a testes empíricos com os estudos de casos. Procurou-se obter um caminho que não fosse nem tão baixo nem tão próximo ao mediano como melhor aproximado. Por exemplo, a Figura 5.1 retrata um código em C e o respectivo CFG com os custos presentes de cada nó. A aresta vermelha em destaque é para lembrar que, como há um laço presente, ele será abstraído por um único nó, o qual conterá o número máximo de ciclos na execução do pior caminho do laço.

Neste exemplo, há no total 38 caminhos. Como visto, o laço executa 18 vezes no máximo, porém são feitas 19 comparações com a condição no pior caso, visto que a 19^a encerra o laço. Além disso, há dois caminhos possíveis para se chegar no laço, logo, existem $2 * 19 = 38$ caminhos possíveis. Destes, o caminho correspondente ao pior fluxo de execução é $p_w = \{W.21, W.16, W.59, W.809, W.3, W.13\}$, com custo total de 921 ciclos (veja Figura 5.2).

Pela definição, o caminho mediano encontra-se na posição $1 + len(P)/2 = 20$, então $p_m = P(20) = \{W.21, W.16, W.11, W.457, W.3, W.13\}$, com custo de 521. Já o melhor caminho aproximado é representado por p_z de tal forma que $ec(p_z) <= ceil(WCEC * 0.15) = 139$. O caminho $p_{ab} = \{W.21, W.16, W.59, W.17, W.3, W.13\}$, com custo total de 129 ciclos, é o que mais se aproxima dos 139 ciclos. Neste caminho, a condição do laço é falsa antes da primeira iteração, logo, não há execução do mesmo.

```
1  int big_foo() {
2    int a, b;
3
4    a = 2;
5    b = 2;
6
7    if (a < b) {
8      a *= 5;
9      a += b;
10     a /= 10;
11   } else {
12     a -= 1;
13   }
14
15   while (b < 20) { //@LOOP 18
16     a += b;
17     b++;
18   }
19
20   return 0;
21 }
```

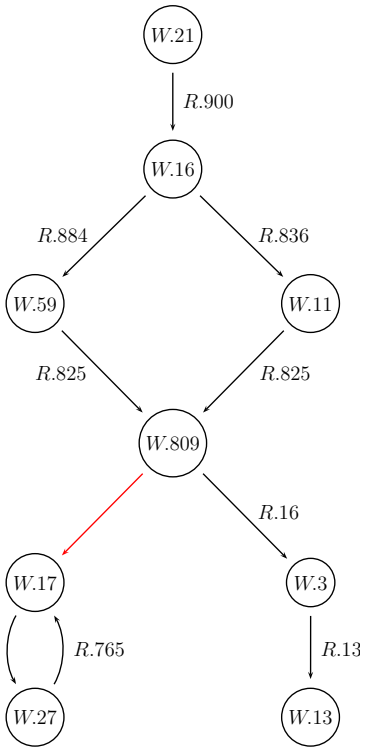


Figura 5.1: Código em C e o respectivo CFG.

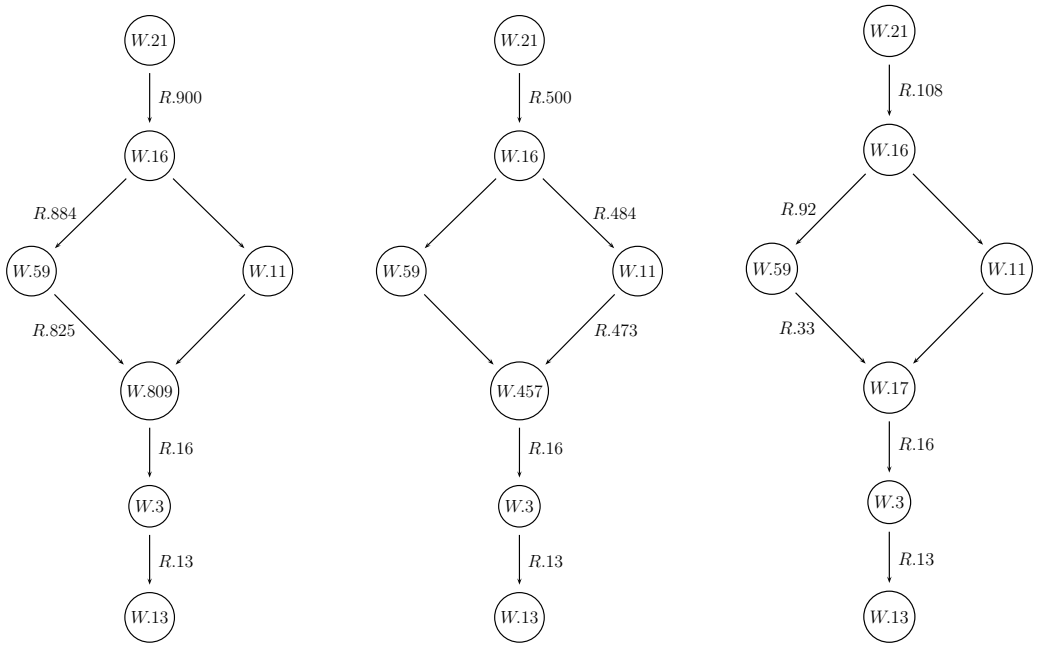
5.1.2 *Overhead*

Overhead refere-se ao custo adicional presente na execução de uma tarefa devido às adições de intruções para troca da tensão e frequência. Em cada novo ponto que for possível aplicar a técnica DVFS, assume-se ser necessário executar mais 100 ciclos de *clock* para calcular a nova frequência e tensão e, assim, determinar se a troca é aceitável e executá-la de fato.

5.1.3 Definição dos *Benchmarks*

Utilizam-se oito tarefas no total para avaliação nos estudos de caso. Uma característica interessante das oito utilizadas é a ausência de dependência com bibliotecas externas (Gustafsson *et al.*, 2010).

Funções como determinar o menor entre dois números e calcular a raiz quadrada foram



(a) Pior Caminho. (b) Caminho Mediano. (c) Melhor Aproximado.

Figura 5.2: Visualização dos três caminhos.

implementadas em vez de utilizar as bibliotecas nativas da linguagem C. A vantagem dessa abordagem é considerar como foco principal apenas a tarefa em execução sem quaisquer eventuais problemas que uma biblioteca possa gerar. Além disso, o código das tarefas é sempre restrito a um único arquivo.

Por fim, adaptou-se as oito tarefas de modo que a execução de cada uma delas ficasse limitada apenas à função principal – *main*, de um código em C. Conforme a explicação da Seção 4.5, anotações do número máximo de iterações de cada laço também foram adicionadas.

As oito tarefas são:

- *CRC*: verificação da redundância de ciclos. Algoritmo voltado para detecção de erros;
- *ST*: cálculo de medidas estatísticas como soma, média, variância, entre outros;
- *FIR*: filtro de resposta ao impulso com duração finita;

- *NDES*: conjunto de algoritmos para manipulação de bits, *shifts*, vetores e matrizes;
- *FFT1*: transformada rápida de Fourier;
- *LUDCMP*: decomposição LU;
- *MINVER*: cálculo da inversa de uma matriz de ponto flutuante;
- *MATMULT*: multiplicação de matrizes quadráticas.

5.1.4 Tensões, Frequências e Medida de Tempo

A escolha do processador alvo, onde os estudos de caso foram executados, teve como base um processador com um número razoável de tensões e frequências disponíveis, a fim de serem mais evidentes otimizações ou não no consumo de energia.

Para isto, adotou-se como referência o processador Intel Xscale (XSCALE, 2002), o qual disponibiliza um conjunto de cinco frequências e tensões (veja Tabela 5.1). Ressalta-se que o conjunto de instruções deste processador faz parte da família *armv5t*, cujo subconjunto de instruções corresponde ao da família *armv4t*.

Tabela 5.1: Frequências e Tensões.

Frequência	Tensão
1000MHz	$1.8V$
800MHz	$1.6V$
600MHz	$1.3V$
400MHz	$1.0V$
150MHz	$0.75V$

Para reduzir a escala dos estudos de caso e apresentar valores de tempo não tão baixos, desprezou-se a unidade MHz. Em outras palavras, consideram-se todas as frequências em Hz apesar do presente texto referenciá-las em MHz.

A adoção dessa abordagem foi necessária para representar o tempo de computação de cada tarefa nos estudos de caso em segundos. Por exemplo, a tarefa *MINVER*

possui $WCEC = 8763$; ao executá-la com uma frequência de 1000MHz, o tempo de computação será de $8.763\mu s$. Todavia, para ocorrerem preempções e troca de frequência, o *deadline* e período das tarefas também precisam ser expressos em μs. Apenas para apresentar os resultados deste capítulo em uma escala maior e sem alterar a aplicabilidade desta dissertação, reduz-se o valor das frequências para Hz, assim, adotam-se os tempos referentes a cada tarefa em segundos.

5.1.5 Tempo de Parada da Simulação

Ao representar um cenário real em simulação, é necessário definir um critério de parada. Caso contrário, a simulação executará indefinidamente. A principal característica das tarefas de tempo real é o tempo. Ele, por si só, permite fazer uma observação interessante quanto ao tempo de simulação, o hiperperíodo – valor igual ao *mínimo múltiplo comum* dos períodos de um conjunto de tarefas periódicas (Farines *et al.*, 2000).

A explicação do tempo de parada é dada a partir do seguinte exemplo: em um sistema de tempo real, executam-se três tarefas τ_1, τ_2 e τ_3, com os períodos iguais a 30, 40 e 60, respectivamente. Assume-se que no tempo zero ocorre a primeira ativação das três tarefas. Este instante no tempo em que todas as tarefas de um sistema estão prontas para ativação, chama-se *Instante Crítico*.

Ao término da execução de cada uma, elas só voltarão a ser executadas nos tempos 30, 40 e 60 devido aos respectivos períodos. As ativações de τ_1 ocorrerão em 0, 30, 60, 90 e 120. Já τ_2 em 0, 40, 80 e 120. A τ_3 em 0, 60 e 120, ou seja, a ativação simultânea das três tarefas ocorre no tempo 120, valor igual ao *mmc* do período delas.

O valor do *mmc* em sistemas de tempo real, calculado a partir dos períodos de um modelo de tarefas, corresponde ao momento no tempo em que as tarefas passam a ter a mesma sequência de ativação desde o tempo zero. Este valor representa, então, o critério de parada da simulação.

5.1.6 Novo *Deadline*

A aplicação do trabalho apresentado em Valentin (2010), informa se o modelo de tarefas dado com entrada é escalonável, as frequências iniciais ótimas e o tempo de resposta de

cada tarefa em consideração com o *jitter*, preempções e recursos compartilhados.

Conforme a Seção 4.8, o tempo de resposta de cada tarefa será o novo *deadline* delas, enquanto o período permanece igual ao original. Isto garante que, para o modelo de tarefas continuar a ser escalonável, o tempo de resposta deve continuar a ser o mesmo, independentemente de qualquer variação da frequência durante a execução das tarefas.

5.1.7 Consumo do Tempo Ocioso

Como definido na Seção 2.1, quando termina a execução de uma tarefa, a diferença entre o tempo de término e o próximo período é definido como tempo ocioso, caso não haja nenhuma nova tarefa para utilizar o processador. Durante este tempo, ainda há consumo de energia. Nos estudos de caso desta dissertação, desprezam-se quaisquer tempos ociosos que venham a acontecer.

5.1.8 Nomenclatura

A seguinte nomenclatura é utilizada nos estudos de caso:

- *Pior Caso*: corresponde à utilização da maior frequência disponível no sistema. Segundo a Tabela 5.1, 1000MHz;
- *Valentin*: refere-se ao estudo apresentado em Valentin (2010);
- *Proposta*: representa o trabalho proposto por esta dissertação;
- *Troca de frequência*: corresponderá sempre à troca de tensão e frequência.

5.2 Estudo de Caso I

Antes de apresentar os resultados da simulação deste estudo de caso, deve-se definir o modelo de tarefas.

Primeiramente, este estudo está sujeito à política de escalonamento *Deadline Monotonic* – DM. Nesta política, o *deadline* não só deve ser menor ou igual ao período, como também as prioridades das tarefas são inversamente definidas por ele, ou seja, quanto menor for o período, maior será a prioridade de uma tarefa. Define-se ainda um conjunto

de três tarefas sujeitas a preempções, sem compartilhamento de recursos e um período de *jitter* fixo de 0.4s. A descrição do modelo de tarefas encontra-se na Tabela 5.2.

Tabela 5.2: Modelo de Tarefas.

Tarefa	*Jitter*	*Deadline*	Período
LUDCMP	0.4s	30s	30s
MINVER	0.4s	40s	40s
MATMULT	0.4s	60s	60s

Para garantir a confiabilidade do sistema e verificar se o modelo é escalonável, aplica-se o trabalho de Valentin – detalhes no Apêndice B, para determinar se o modelo de tarefas é escalonável, as frequências iniciais ótimas e o novo *deadline*, que será igual ao tempo de resposta, sempre com base no Pior Caminho de execução. O resultado desta aplicação mais os custos de execução do Pior Caminho, Mediano e Melhor Aproximado a serem utilizados por Valentin e Pior Caso, estão na Tabela 5.3.

Tabela 5.3: Dados para Pior Caso e Valentin.

Tarefa	Pior	Mediano	Melhor Aproximado	*Deadline'*	Frequência Inicial
LUDCMP	10107	4773	1515	13.03s	800Hz
MINVER	8763	4285	1309	21.8s	1000Hz
MATMULT	13651	6700	2018	56.84s	1000Hz

Como este modelo de tarefas é escalonável, aplica-se a proposta desta dissertação. Identificam-se as arestas do tipo-B e tipo-L do CFG; em seguida, adicionam-se o cálculo para determinar a nova frequência e a verificação da possível troca. Cada uma dessas informações adicionais gera um ou mais *overheads* nas tarefas e, se estes estes custos adicionais ocorrem devido a uma aresta do tipo-L, é necessário verificar se o modelo de tarefas inicial continua válido, pois é possível que o WCEC original da tarefa tenha um valor maior por causa deles.

Assim, após a identificação das arestas, aplica-se novamente a ideia de Valentin, de onde se extraiu as informações presentes na Tabela 5.4.

Tabela 5.4: Dados para Proposta.

Tarefa	Pior	Mediano	Melhor Aproximado	*Deadline"*	Frequência Inicial
LUDCMP	10707	5373	1915	11.11s	1000Hz
MINVER	9563	5085	1809	23.06s	800Hz
MATMULT	13951	7000	2318	59.67s	1000Hz

Tanto o Pior Caso como Valentin utilizam as informações da Tabela 5.3. Já a Proposta, às da Tabela 5.4. Isto é feito devido aos *overheads* adicionados ao WCEC das tarefas, pois não seria justo aplicar as outras duas abordagens em um cenário diferente do previamente calculado.

Além disso, o tempo de simulação deste modelo de tarefas é igual ao *mmc* dos períodos da Tabela 5.2, logo 120s. A seguir, analisa-se a execução simultânea das três tarefas ao percorrer só o pior caminho, depois apenas o mediano e, por fim, o melhor aproximado.

5.2.1 Pior Caminho

Na Figura 5.3 é interessante observar a ordem de execução entre as tarefas. τ_1 corresponde à tarefa de maior prioridade LUDCMP; τ_2 à MINVER; já τ_3 à tarefa MATMULT. Esta figura apresenta um resumo da simulação do Pior Caminho pela Proposta e as frequências utilizadas nas tarefas. Vale ressaltar que as três tarefas no tempo 0 estão prontas. Porém, apenas no tempo 0.4 a ativação de fato ocorre devido ao *jitter*.

Apesar da ativação das três tarefas ocorrer ao mesmo tempo, apenas τ_1 inicia a execução por ser a mais prioritária. Ao terminar de executar, esta tarefa terá uma nova ativação no tempo 30.4, pois, mesmo com o período a cada 30s, há um *jitter* de 0.4s. Já a tarefa τ_2 entra em execução, pois é a que tem maior prioridade dentre as tarefas em execução. Ao final, τ_3 começa a executar, contudo, esta tarefa é preemptada no tempo 30.4 por τ_1, sem finalizar a execução.

Novamente, τ_1 começa a executar e, ao término, o processador começa a execução de τ_2. Por fim, τ_3 volta a ser executada e consegue ser finalizada antes do *deadline* de 59.67s.

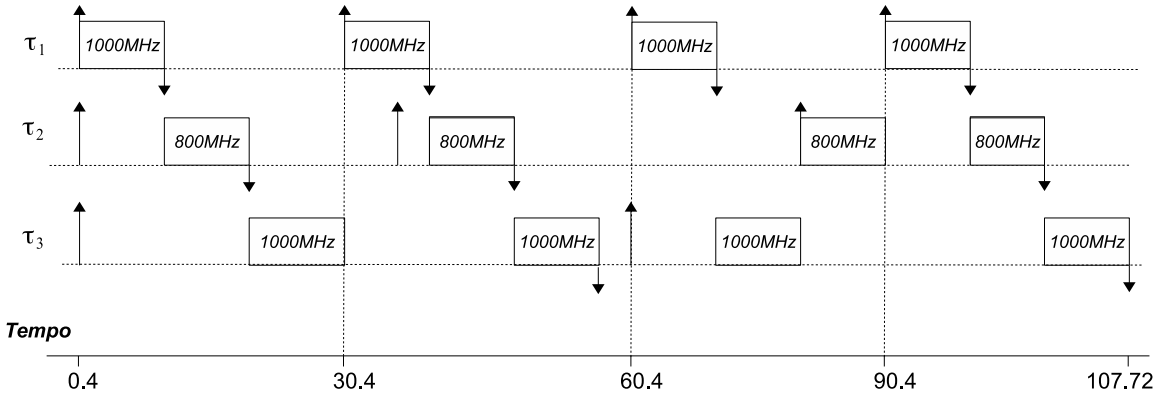


Figura 5.3: Pior Caminho: Simulação das Tarefas.

Por sua vez, em 60.4, ocorre a ativação das tarefas τ_1 e τ_3. Mais uma vez por τ_1 ter maior prioridade, entrará em execução. Após o término de τ_1, o processador executa τ_3. Porém τ_3 é preemptada por τ_2 que, após um momento em execução, também é preemptada, por τ_1. Quando as duas tarefas mais prioritárias terminarem a execução, τ_3 é retomada com o tempo de término em 107.72s. Além disso, como o Pior Caminho do CFG é percorrido, não há troca de frequência.

A Figura 5.4 apresenta uma comparação entre os tempos de término da simulação para cada abordagem.

Se as três tarefas são executadas com a maior frequência, o tempo de término do Pior Caso é menor ao comparar com o Valentin ou mesmo a Proposta. Como visto nas Tabelas 5.3 e 5.4, apenas uma, das frequências iniciais ótimas, tem o valor menor que o da maior frequência disponível. A Proposta só poderia ter um tempo de término igual ao do Pior Caso se o mesmo número de ciclos fossem executados e sempre com a maior frequência do sistema.

Ainda sobre o tempo de término, a aplicação da Proposta adiciona *overheads* ao WCEC das três tarefas, logo, o tempo de execução tende a ser maior se o mesmo cenário do Pior Caso e Valentin fosse utilizado.

No entanto, de acordo com a Tabela 5.4, as frequênciais iniciais das tarefas LUDCMP e MINVER são diferentes aos da Tabela 5.3. Nesta, ocorre a execução das tarefas com 800MHz e 1000MHz, respectivamente, enquanto naquela com 1000MHz e 800MHz. Esta troca de frequência faz com que a tarefa LUDCMP, que tem um WCEC maior que o da

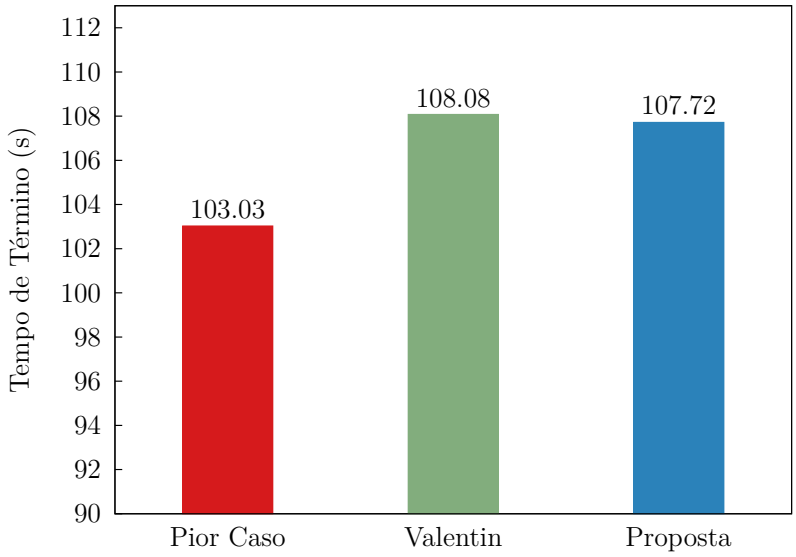


Figura 5.4: Pior Caminho: Comparação do Tempo de Término.

MINVER, seja executada em um tempo menor pela Proposta. Já o tempo de execução de MINVER será maior. Isto resulta em um tempo de término menor pela Proposta, mesmo com a adição dos *overheads*.

Por outro lado, a adição dos *overheads* influencia diretamente no consumo de energia. Pois, como o WCEC da Proposta tem um valor maior ao utilizado por Valentin e Pior Caso, ocorre a execução de um número maior de ciclos com altas frequências e, portanto, o consumo de energia tende a ser maior.

As Figuras 5.5, 5.6 e 5.7 apresentam não só o consumo total de energia ao longo do tempo, como também por intervalo – de 20s neste caso, na aplicação de cada abordagem. Importante observar que, caso não haja consumo parcial em um dado intervalo, mantém-se o consumo total no mesmo ponto anterior.

As informações das três figuras são interessantes por apresentar o quanto o consumo em um dado intervalo influencia no consumo total da energia. Por exemplo, no consumo do intervalo de 60s, o Pior Caso apresentou um valor menor do que os demais – apenas $39206.08J$ (veja Tabela 5.5), pois, com uma frequência maior, menor será o tempo de computação e execução das tarefas. Já os estudos da Proposta e Valentin, por utilizarem

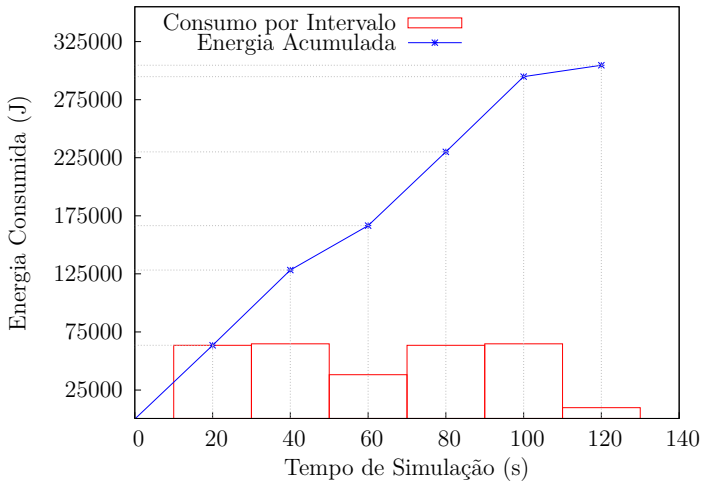


Figura 5.5: Pior Caminho: Consumo Parcial de Energia do Pior Caso.

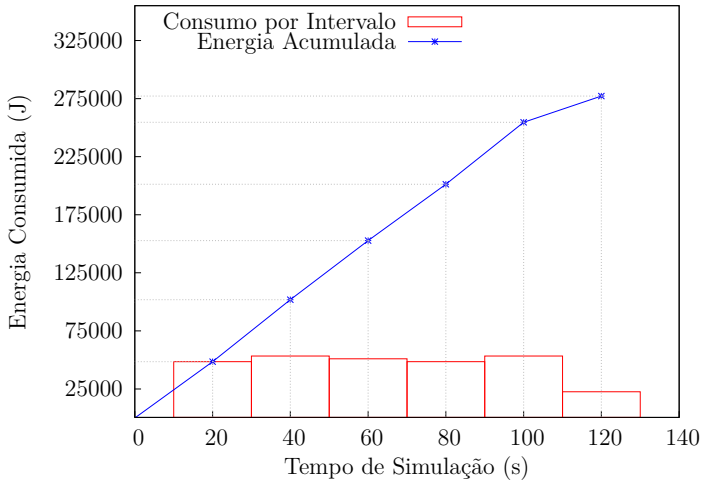


Figura 5.6: Pior Caminho: Consumo Parcial de Energia do Valentin.

uma frequência menor em uma das tarefas, têm um tempo de execução maior, o qual influencia diretamente no consumo por intervalo, pois maior será também o tempo que uma dada tarefa estará em execução. O consumo no intervalo de 120s segue o mesmo princípio: com uma frequência maior, o Pior Caso termina a execução das tarefas antes da Proposta e Valentin, logo, menor será o consumo registrado neste intervalo.

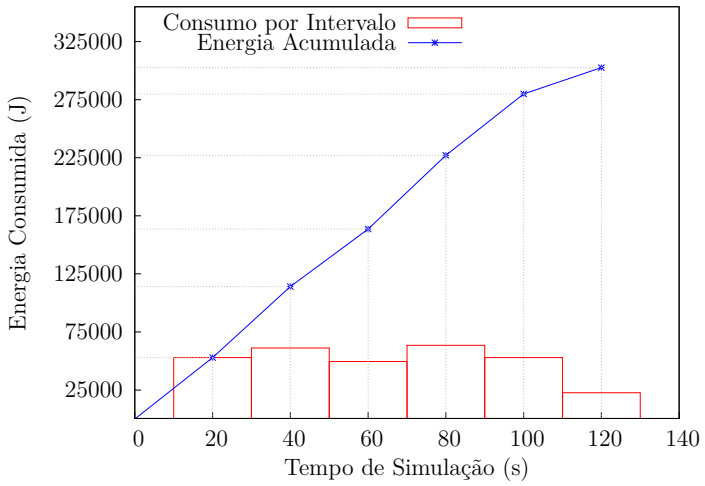


Figura 5.7: Pior Caminho: Consumo Parcial de Energia da Proposta.

Tabela 5.5: Pior Caminho: Consumo Parcial de Energia.

Tempo (s)	Pior Caso (J)	Valentin (J)	Proposta (J)
20	63507.24	48447.00	52905.08
40	64800.00	53360.04	61155.04
60	38206.08	50960.08	49490.84
80	63504.00	48447.00	63507.24
100	64803.24	53360.04	52883.24
120	9813.96	22567.96	22679.88

A Figura 5.8 apresenta uma comparação do consumo total de energia das três abordagens, de maneira a possibilitar uma melhor verificação da diferença entre elas. Mesmo com a apresentação de valores menores do consumo de energia em alguns intervalos (60s e 120s) pelo Pior Caso, a soma total faz com que esta abordagem apresente o maior consumo das três. Já a Proposta, teve um consumo superior ao Valentin, mas inferior ao Pior Caso, mesmo com a adição dos *overheads*.

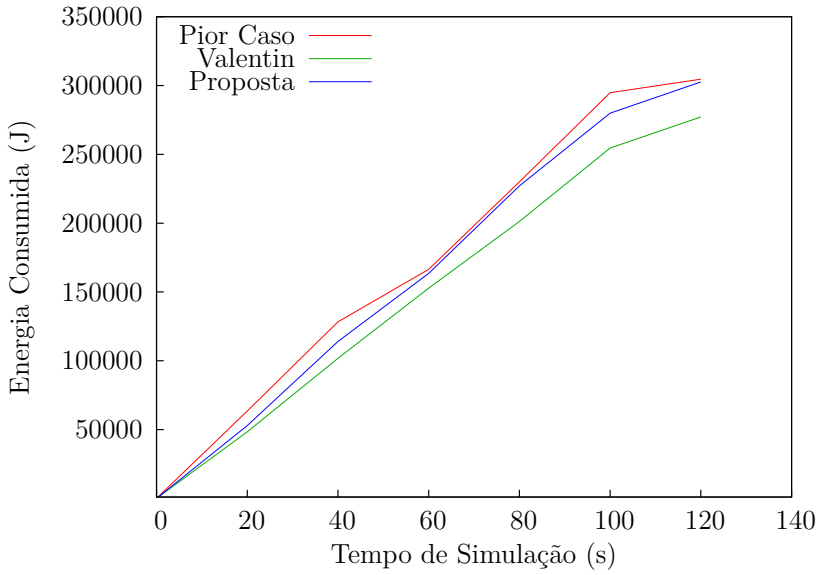


Figura 5.8: Pior Caminho: Comparação do Consumo de Energia.

5.2.2 Caminho Mediano

Da mesma forma do Pior Caminho, a Figura 5.9 apresenta a ordem de execução entre as tarefas no Caminho Mediano. Nesta figura, τ_1 corresponde a tarefa de maior prioridade LUDCMP; τ_2 à MINVER; já τ_3 a tarefa MATMULT.

Ao contrário do Pior Caminho, houve troca de frequência durante a simulação. A Figura 5.9 mostra a ativação simultânea das três tarefas, contudo a tarefa mais prioritária a preemptar as demais e entrar em execução é τ_1.

τ_1 inicia a execução com 1000MHz, conforme o conjunto de frequências ótimas. Como há desvios do Pior Caminho no fluxo de execução, há maior presença de arestas tipo-B e tipo-L, tanto que houve troca de frequência em três possíveis momentos: 1000MHz - 800MHz; 800MHz - 600MHz; 600MHz - 150MHz. Já ao término de τ_1, τ_2 inicia e, durante a execução, também reduz a frequência.

τ_3 apresenta uma situação interessante. Após a utilização de 800MHz, τ_3 troca a frequência para 150MHz. Em outras palavras, como havia uma grande diferença entre o Pior Caminho e o Mediano neste caso, a frequência calculada e utilizada foi a menor dentre o conjunto disponível no sistema.

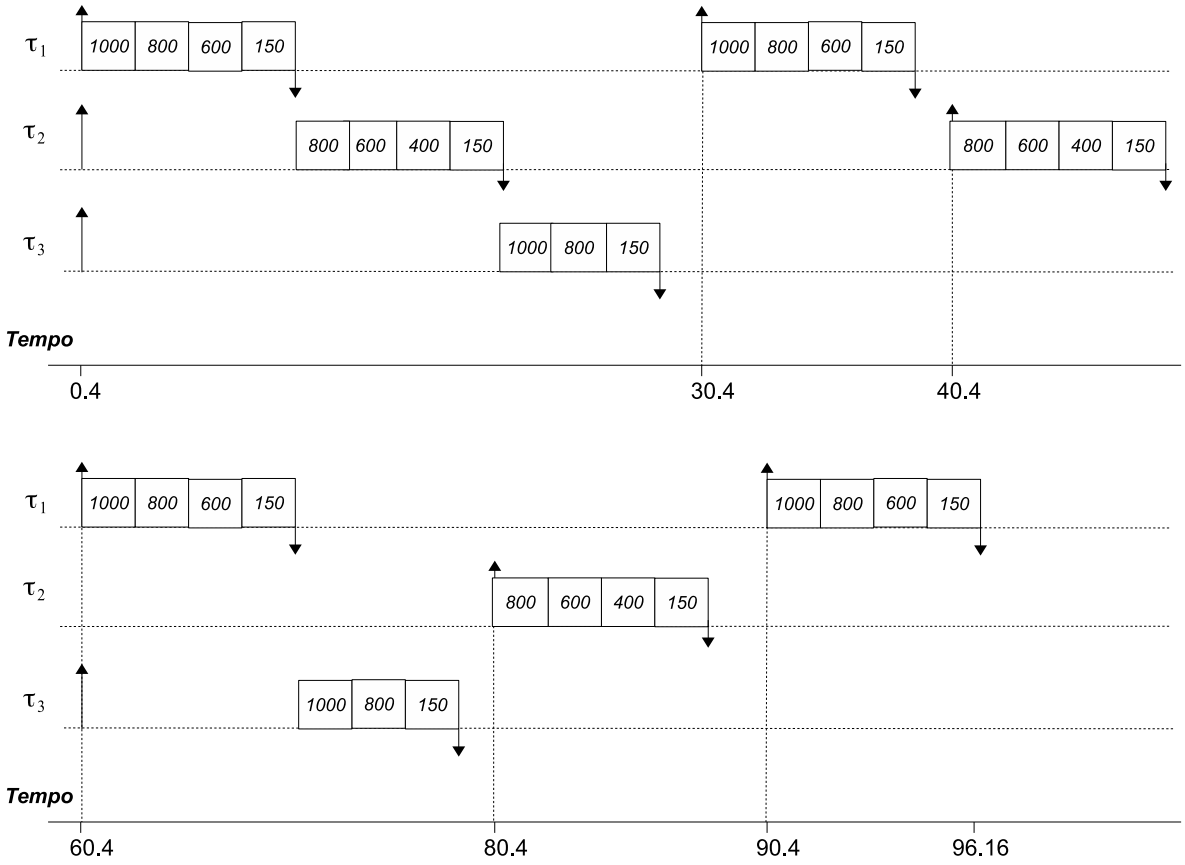


Figura 5.9: Caminho Mediano: Simulação das Tarefas.

Depois do término de τ_3, a próxima tarefa em execução será τ_1, cujo comportamento é o mesmo descrito anteriormente. Inclusive, as demais execuções mantêm o mesmo padrão das frequências e apenas uma preempção ocorre novamente, mas no tempo 60.4s, onde τ_1 e τ_3 são ativadas ao mesmo tempo.

Quanto ao tempo de término, os valores seguem um padrão parecido com o comportamento do Pior Caminho (veja Figura 5.10). O Pior Caso apresenta um tempo de execução menor por utilizar sempre a maior frequência na execução das três tarefas.

Por sua vez, o tempo de término de Valentin é superior ao da Proposta. Este é um ponto interessante, pois, assim como o Pior Caminho, *overheads* foram adicionados, mas, neste caso, a troca de frequência é possível e foi realizada.

Ainda na análise do tempo, mesmo a execução das tarefas com frequências menores mantiveram o tempo de término da Proposta abaixo ao de Valentin. Isto ocorreu pois

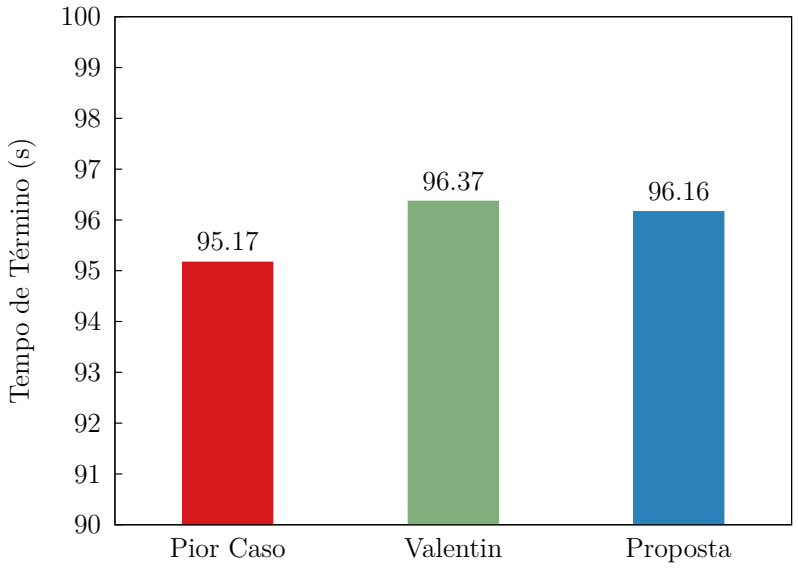


Figura 5.10: Caminho Mediano: Comparação do Tempo de Término.

nem sempre a adoção de uma frequência menor representará um impacto no tempo de execução. Por exemplo, na aplicação da Proposta, a tarefa LUDCMP começou a execução com 1000MHz e, depois, alterou-se para 800MHz, enquanto que na abordagem do Valentin a mesma tarefa começou a execução com 800MHz e manteve-se constante até o fim.

Outro exemplo é o caso da tarefa MATMULT que começa a execução com 1000MHz e, depois, altera-se para 800MHz. Por mais que o número de ciclos executados desta vez por 800MHz seja maior, não apresentou uma diferença no tempo total de término da simulação.

As Figuras 5.11, 5.12 e 5.13 apresentam os consumos de energia por intervalo de tempo e o total de cada abordagem. Nestas figuras, verifica-se um consumo elevado do Pior Caso e da Proposta. Ambos mantêm um alto consumo inicial e pequenas diferenças nos demais intervalos, com a mais significativa no tempo 40 – onde a Proposta obteve um consumo superior entre as abordagens com $21680.06J$ (veja Tabela 5.6). Interessante observar ainda os consumos parciais nos tempos 20s, 60s e 120s.

Em 20s e 60s, o consumo parcial da Proposta foi o menor. Porém, este consumo não

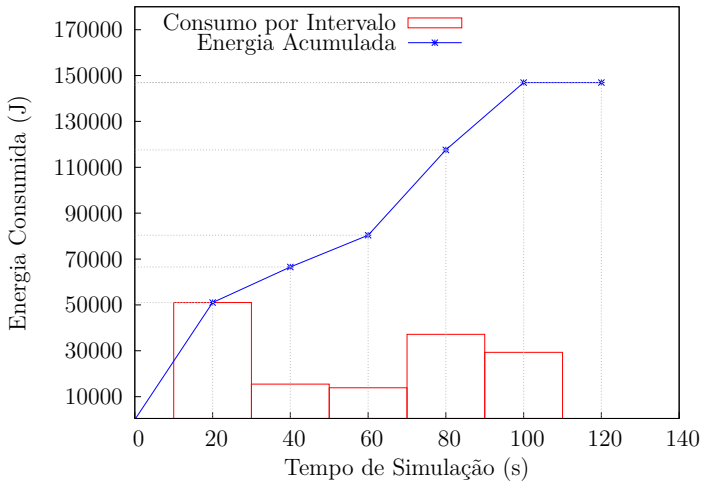


Figura 5.11: Caminho Mediano: Consumo Parcial de Energia do Pior Caso.

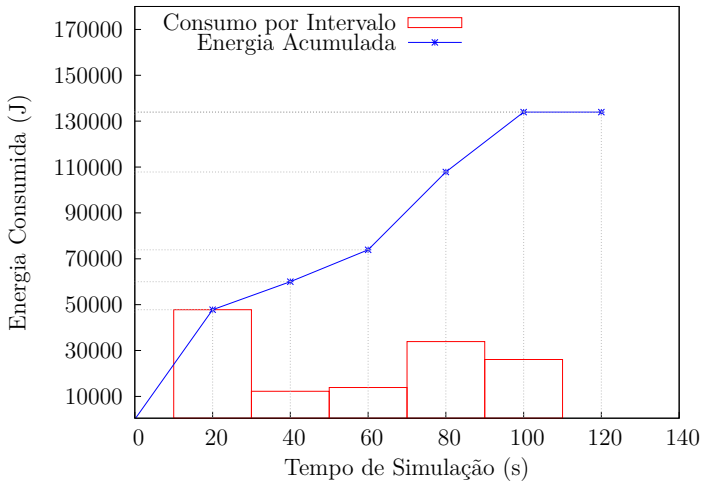


Figura 5.12: Caminho Mediano: Consumo Parcial de Energia: Valentin.

foi suficientemente baixo para compensar o alto consumo nos demais tempos ao longo da simulação.

Já no tempo 120s, o consumo parcial das três abordagens foi zero, pois como o cenário atual consiste do Caminho Mediano cujo custo total de execução é menor do que o Pior Caminho, o tempo de execução das tarefas também é menor, ou seja, a execução das

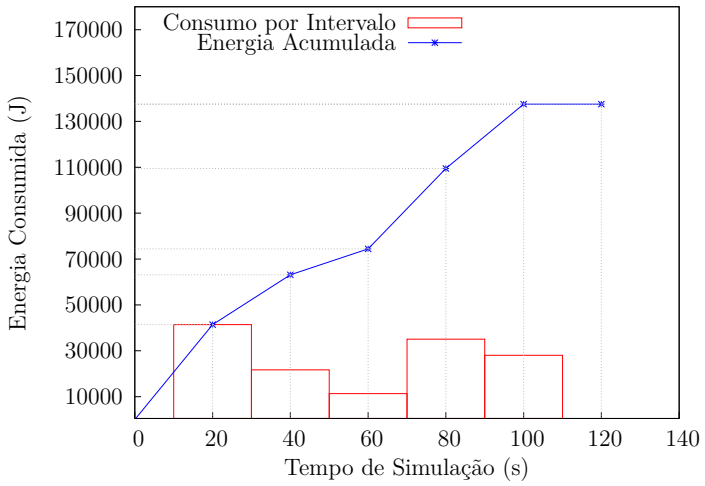


Figura 5.13: Caminho Mediano: Consumo Parcial de Energia da Proposta.

Tabela 5.6: Caminho Mediano: Consumo Parcial de Energia.

Tempo (s)	Pior Caso (J)	Valentin (J)	Proposta (J)
20	51055.92	47810.28	41432.85
40	15464.52	12218.88	21680.06
60	13883.40	13883.40	11320.51
80	37172.52	33926.88	35064.70
100	29347.92	26102.28	28045.65
120	0.00	0.00	0.00

três tarefas terminou antes do último intervalo da simulação em 120s, sem registrar um consumo de energia neste intervalo.

Quanto ao consumo total, a Proposta teve menos redução de energia quando comparada a Valentin. Isto ocorre devido às novas frequências utilizadas e a quantidade de ciclos executados, pois as frequências não foram baixas o suficiente para compensar a execução dos *overheads* e a utilização de um conjunto inicial de frequências, cujo consumo total do modelo de tarefas é maior. Vale destacar que o consumo total da Proposta ainda esteve abaixo do Pior Caso (veja Figura 5.14).

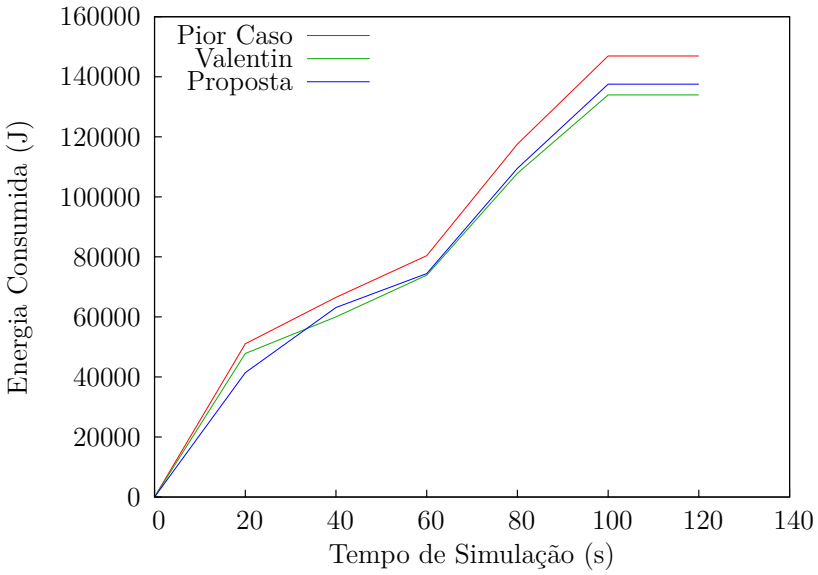


Figura 5.14: Caminho Mediano: Comparação do Consumo de Energia.

5.2.3 Melhor Caminho Aproximado

A simulação do Melhor Caminho Aproximado, retratada na Figura 5.15, apresenta um comportamento parecido ao do Caminho Mediano. Novamente, há troca de frequências durante a execução das tarefas. Mas como se observa nesta figura, as tarefas τ_1 e τ_2 assumem agora apenas três frequências contra quatro do Caminho Mediano.

As trocas de frequências estão relacionadas não só às arestas tipo-B e tipo-L presentes no caminho de execução, como também ao RWCEC. Uma vez que a Figura 5.15 apresenta a simulação do Melhor Caminho Aproximado, sabe-se que a quantidade de ciclos necessários para execução de cada tarefa é 15% do WCEC da tarefa, o que influencia no RWCEC de cada nó do caminho percorrido.

Quanto ao término da simulação, a Proposta apresenta um valor maior ao das demais abordagens (veja Figura 5.16). A troca de frequência é a principal responsável por esta diferença entre os tempos.

Como é possível realizar a troca de frequência durante a execução de uma tarefa, o tempo necessário para executar um número de ciclos será maior já que menos instruções são executadas por unidade de tempo. As frequências utilizadas no Melhor Caminho

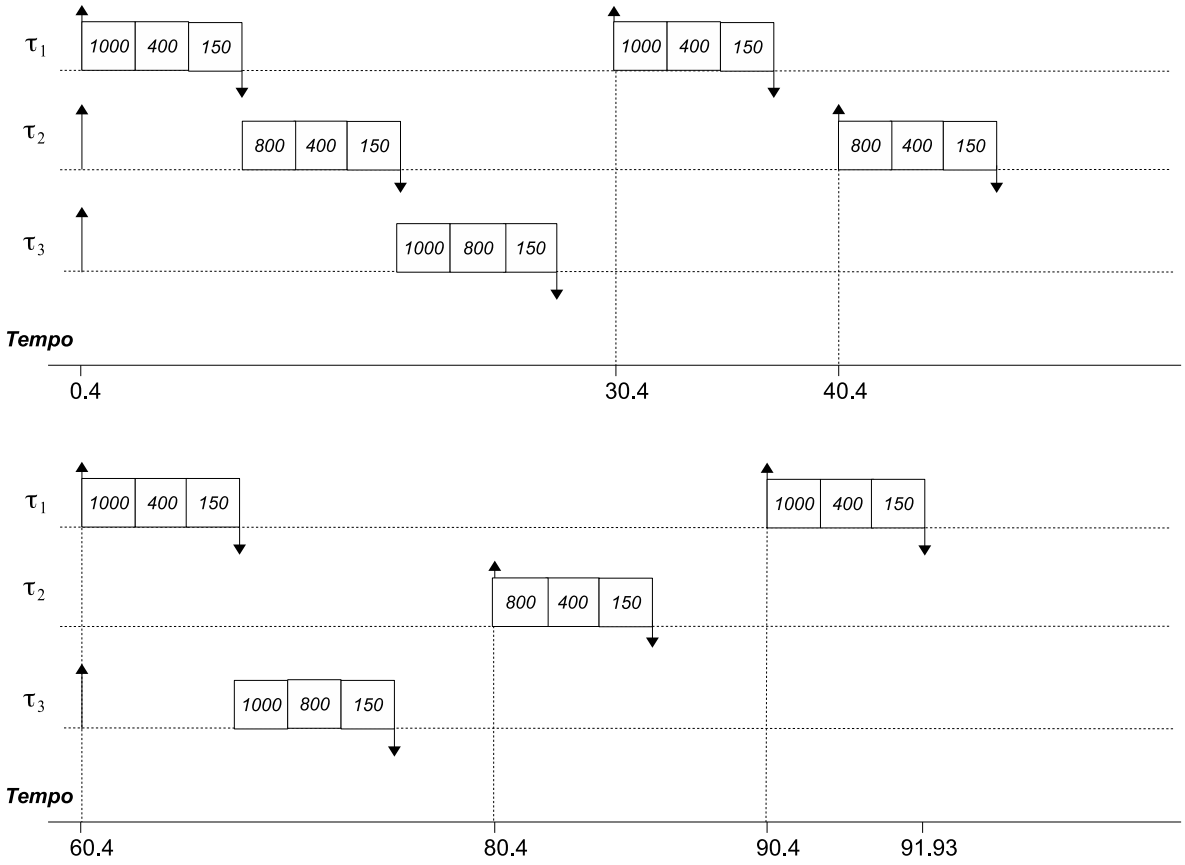


Figura 5.15: Melhor Aproximado: Simulação das Tarefas.

Aproximado foram menores do que as adotadas no Caminho Mediano, o que refletiu diretamente no tempo de término da Proposta.

Por exemplo, a tarefa LUDCMP iniciou a execução com 1000MHz e, depois, adotou 800MHz no Caminho Mediano. Já no Melhor Aproximado, LUDCMP começou com 1000MHz e, em seguida, adotou 400MHz. Esta diferença contribuiu para o aumento do tempo de término da Proposta.

As Figuras 5.17, 5.18 e 5.19 apresentam o consumo de energia total e por intervalo de tempo em cada abordagem. Nelas ainda se verifica que não houve otimizações no consumo de energia da Proposta, mesmo com a troca de frequência. Por exemplo, em 20s, o consumo parcial registrado pela Proposta foi superior ao das demais abordagens - 17134.05J (veja Tabela 5.7). Comportamento este que se repetiu ao longo dos intervalos. Assim, como o consumo parcial da Proposta foi maior em todos os momentos quando

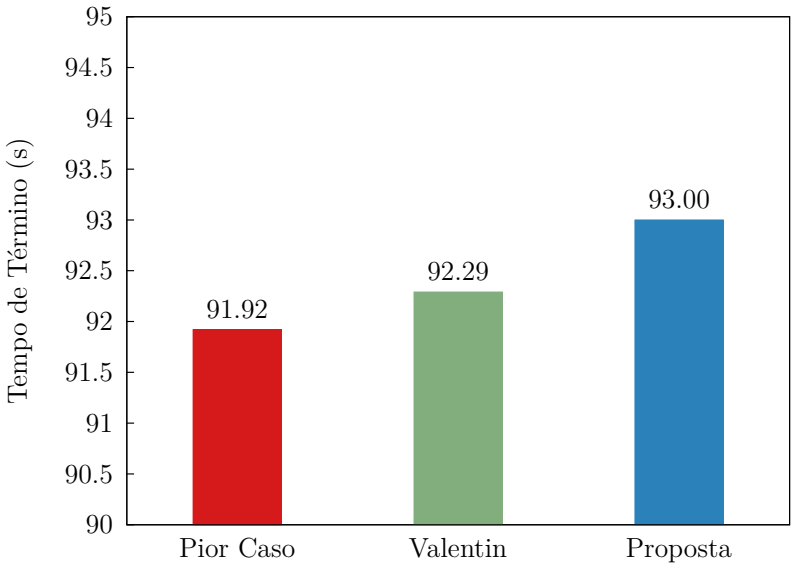


Figura 5.16: Melhor Aproximado: Comparação do Tempo de Término.

comparado ao Pior Caso e Valentin, também se teve um valor maior em termos do consumo total.

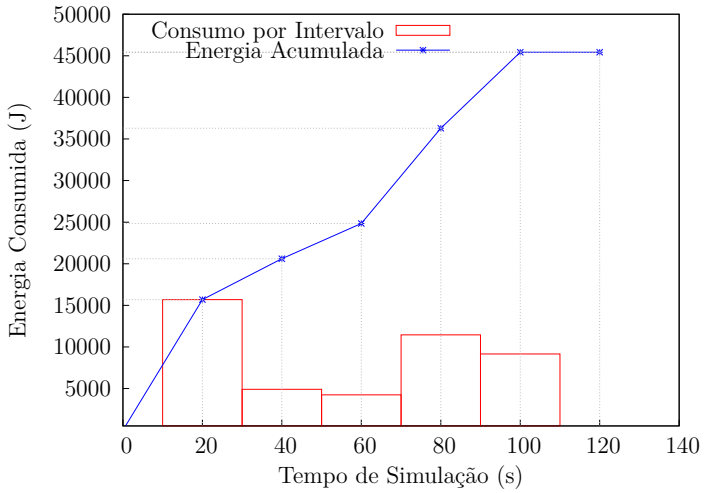


Figura 5.17: Melhor Aproximado: Consumo Parcial de Energia do Pior Caso.

A Figura 5.20 compara o consumo total de energia com a utilização de cada abordagem.

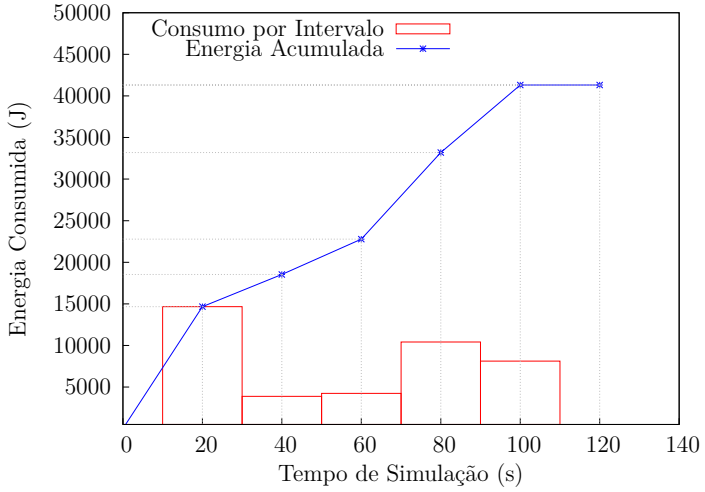


Figura 5.18: Melhor Aproximado: Consumo Parcial de Energia: Valentin.

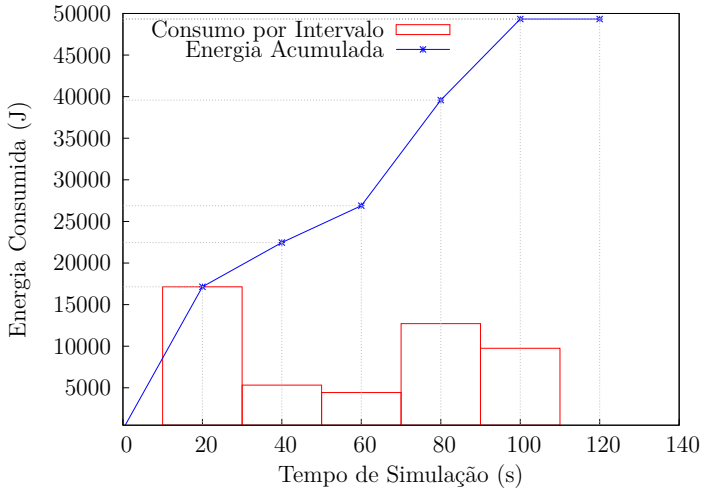


Figura 5.19: Melhor Aproximado: Consumo Parcial de Energia da Proposta.

Ela apresenta uma consequência interessante na aplicação da Proposta, a curva do consumo de energia total é superior não só ao do Valentin, como também ao do Pior Caso. Para melhor compreensão deste caso, deve-se analisar em detalhes o contexto deste cenário.

O presente estudo de caso consiste na execução do Melhor Caminho Aproximado

Tabela 5.7: Melhor Aproximado: Consumo Parcial de Energia.

Tempo (s)	Pior Caso (J)	Valentin (J)	Proposta (J)
20	15688.08	14657.88	17134.05
40	4908.60	3878.40	5322.74
60	4241.16	4241.16	4425.43
80	11446.92	10416.72	12708.62
100	9149.76	8119.56	9748.17
120	0.00	0.00	0.00

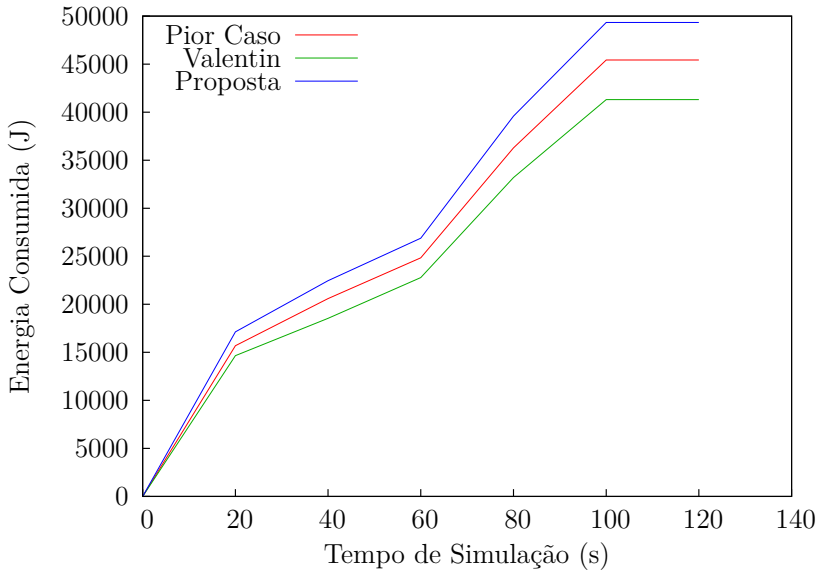


Figura 5.20: Melhor Aproximado: Comparação do Consumo de Energia.

de três tarefas. Ao considerar este caminho como 15% do WCEC da tarefa, têm-se os custos de execução de cada tarefa como valores relativamente baixos (veja Tabela 5.3). Assim, mesmo a adição de um único *overhead* devido à aplicação da Proposta, apresenta um grande impacto no consumo total de energia.

Cada *overhead* corresponde a um custo adicional de 100 ciclos. Se o caminho percorrido contiver **muitos** *overheads*, maior será o custo total da tarefa em execução em termos de ciclos – Tabela 5.4; outro ponto é a execução de cada *overhead* pela **frequência**

atual. Os *overheads* correspondem ao cálculo da nova frequência e verificação se a mesma poderá ser utilizada, devido ao caminho percorrido do CFG. A execução de toda esta fase é realizada pela frequência atual, ou seja, se uma tarefa está em execução com 1000MHz e verificou-se que é possível trocar a frequência para 800MHz, então toda essa etapa de determinar a nova frequência e verificar se a troca é possível, é executada com 1000MHz. Isto aumenta o consumo de energia realizada por uma mesma frequência.

Outro fator a justificar o consumo elevado de energia pela Proposta é: quantos **por cento** do custo total para executar o Melhor Caminho Aproximado corresponde um *overhead*. Em outras palavras, apenas um *overhead* de 100 ciclos refere-se a 4.9% $\sim$ 7.5% do custo total do Melhor Caminho Aproximado de cada tarefa neste estudo de caso, ou seja, quanto maior for o número de *overheads*, mais significativa será a influência dele no custo total.

O último ponto a considerar na aplicação da Proposta é se a troca de frequências durante a execução de qualquer caminho no CFG, **compensará** os custos adicionais dos *overheads* e o novo conjunto inicial de frequências, se houver. Estas observações justificam como a Proposta pode ter um consumo superior ao comparar-se com os consumos de Valentin e Pior Caso.

5.3 Estudo de Caso II

O modelo de tarefas deste estudo de caso assemelha-se com o do estudo I. Considera-se a política de escalonamento DM, um *jitter* de 0.4s, tarefas sujeitas à preempções, mas sem compartilhamento de recursos. Utilizam-se oito tarefas neste estudo, segundo é definido na Tabela 5.8.

O resultado da aplicação do trabalho de Valentin para verificar se o modelo de tarefas é escalonável e quais as frequências iniciais ótimas, está presente na Tabela 5.9.

Por fim, é necessário verificar se o WCEC das tarefas da Tabela 5.9 são alterados a partir da aplicação da Proposta. Em seguida, utiliza-se mais uma vez o trabalho de Valentin a fim de identificar se a adição dos *overheads* manteve o modelo de tarefas escalonável. A Tabela 5.10 contém os resultados dessas análises.

Novamente, ao se comparar as Tabelas 5.9 e 5.10, verifica-se que o conjunto de frequências iniciais foi alterado, igual ao estudo de caso anterior. Já pelos dados da

Tabela 5.8: Modelo de Tarefas.

Tarefa	*Jitter*	*Deadline*	Período
CRC	0.4s	300s	300s
ST	0.4s	320s	320s
FIR	0.4s	400s	400s
NDES	0.4s	420s	420s
FFT1	0.4s	420s	420s
LUDCMP	0.4s	450s	450s
MINVER	0.4s	450s	450s
MATMULT	0.4s	500s	500s

Tabela 5.9: Dados para Pior Caso e Valentin.

Tarefa	Pior	Mediano	Melhor Aproximado	*Deadline'*	Frequência Inicial
CRC	29186	14255	4378	29.59s	1000Hz
ST	44569	20721	6686	74.16s	1000Hz
FIR	56950	28516	8288	169.07s	600Hz
NDES	58779	29425	8817	227.85s	1000Hz
FFT1	61683	30345	9253	289.53s	1000Hz
LUDCMP	10107	4773	1515	375.92s	800Hz
MINVER	8763	4285	1309	384.69s	1000Hz
MATMULT	13651	6700	2018	398.34s	1000Hz

Tabela 5.8, pode-se determinar o tempo de parada da simulação, o qual é igual ao *mmc* dos períodos. Neste caso, o tempo final é de 504 mil segundos.

Devido à extensão da simulação e por analisar a execução simultânea de oito tarefas, este estudo de caso focará nos resultados obtidos sem apresentar como transcorreu a troca de frequência das tarefas. Assim, analisam-se os mesmos caminhos do estudo anterior: pior, mediano e o melhor aproximado.

Tabela 5.10: Dados para Proposta.

Tarefa	Pior	Mediano	Melhor Aproximado	*Deadline"*	Frequência Inicial
CRC	30386	15355	5578	38.38s	800Hz
ST	45869	21821	7686	84.25s	1000Hz
FIR	57150	28716	8488	155.44s	800Hz
NDES	59479	30125	9517	215.17s	1000Hz
FFT1	63183	31745	10653	278.35s	1000Hz
LUDCMP	10707	5373	1915	291.73s	800Hz
MINVER	9563	5085	1809	385.15s	1000Hz
MATMULT	13951	7000	2318	399.1s	1000Hz

5.3.1 Pior Caminho

Na Figura 5.21, observa-se como o tempo de término do Pior Caso é inferior ao das duas outras abordagens. Isto se deve ao fato de sempre utilizar a maior frequência na execução de todas as tarefas, e o custo adicional ao WCEC da Proposta não ser tão significativo ao ponto de ter de utilizar sempre a maior frequência para se cumprir o *deadline*.

Devido a Proposta adicionar *overheads* no WCEC da tarefa, as frequências ótimas iniciais das tarefas CRC e FIR sofreram alteração de 1000MHz e 600MHz para 800MHz, como visto na Tabela 5.10. Se a execução destas duas tarefas ocorresse com as frequências de 1000MHz e 600MHz, caso do Valentin, o tempo de término delas seria de ≈124.10s, enquanto com 800MHz e um WCEC maior devido aos *overheads*, este tempo é reduzido para ≈109.42s. Esta diferença serve para ilustrar um dos motivos do tempo de término da Proposta ser menor ao de Valentin.

Já as Figuras 5.22, 5.23 e 5.24 apresentam não só consumo total de energia, como também o consumo em cada intervalo na simulação para as três abordagens. O intervalo de tempo, neste caso, foi definido a cada 50 mil segundos com o objetivo de restringir a escala do tempo.

É possível identificar que a curva do consumo total de energia da Proposta é superior ao de Valentin, pois o consumo da Proposta está mais próximo dos $1150 * 10^6 J$, enquanto

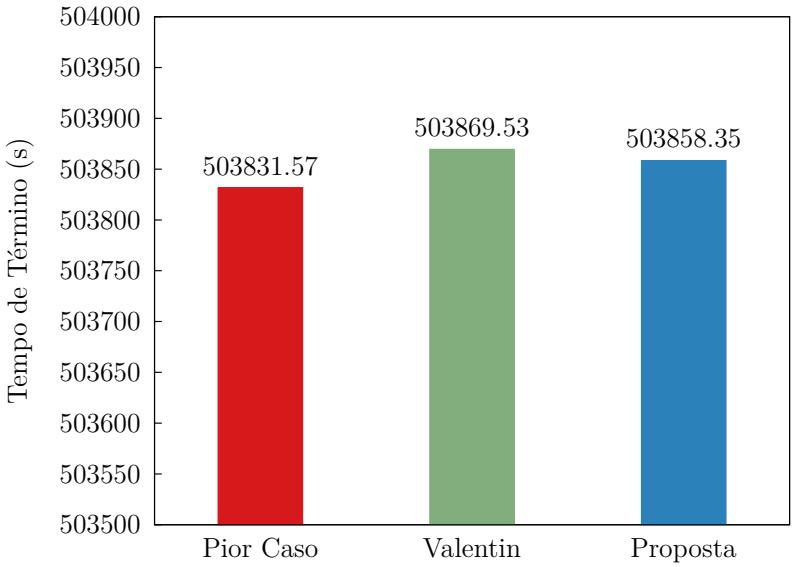


Figura 5.21: Pior Caminho: Comparação do Tempo de Término.

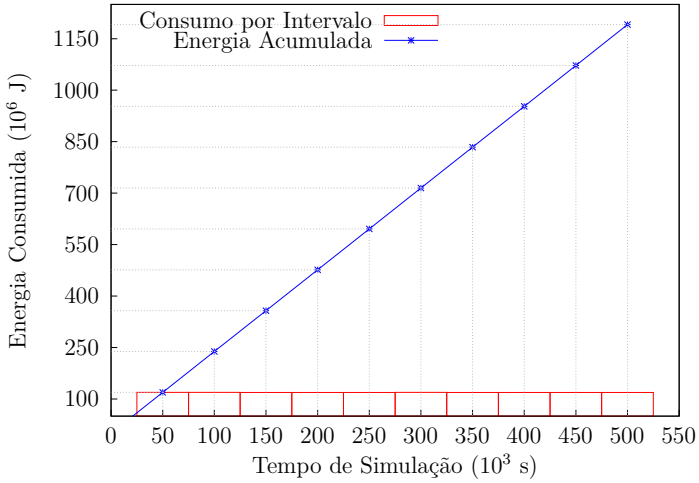


Figura 5.22: Pior Caminho: Consumo Parcial de Energia do Pior Caso.

o Pior Caso se manteve maior do que os demais. Porém, um ponto importante a destacar e algo a gerar dúvidas está no consumo de energia por intervalo.

Diferentemente do estudo de caso anterior, o atual apresentou um consumo constante durante a simulação, o que não retrata o consumo real de energia. De acordo com a

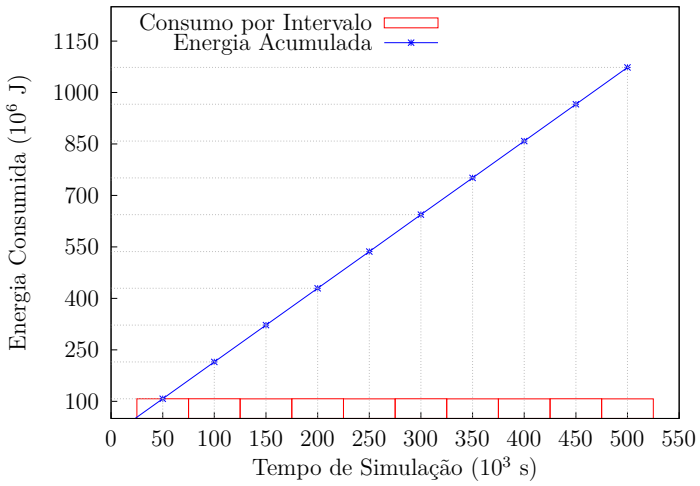


Figura 5.23: Pior Caminho: Consumo Parcial de Energia: Valentin.

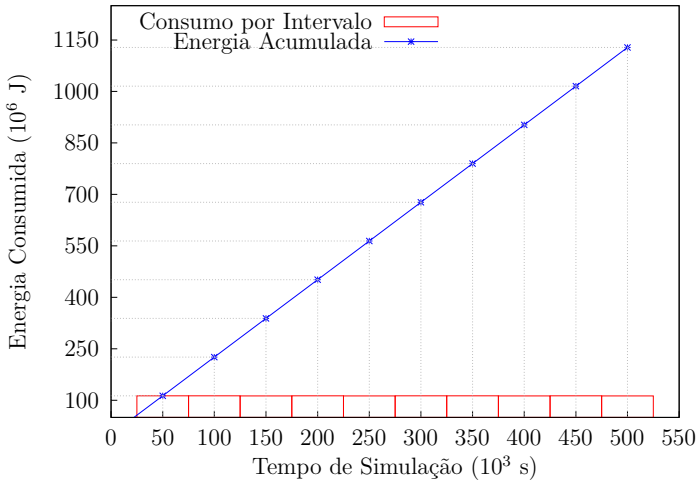


Figura 5.24: Pior Caminho: Consumo Parcial de Energia da Proposta.

Tabela 5.9, o *deadline* da última tarefa, igual a 398.34s, informa que neste tempo todas as tarefas terão executado ao menos uma vez. Verificar o consumo de energia a cada 50 mil segundos, ponto em que o *deadline* da última tarefa é 398.34s e os períodos são baixos, não apresenta o consumo real da execução de cada tarefa. O ideal seria verificar o consumo a cada 200s para se ter uma amostra com maiores variações no consumo de

energia. A Tabela 5.11 procura apresentar os valores dos consumos parciais de cada abordagem nos respectivos intervalos de 50 mil segundos.

Tabela 5.11: Pior Caminho: Consumo Parcial de Energia.

Tempo (s)	Pior Caso (J)	Valentin (J)	Proposta (J)
50000	119296229.76	107484233.90	112997602.72
100000	119114274.60	107325519.98	112850850.64
150000	119008566.36	107211627.50	112747706.24
200000	119103129.00	107306190.14	112823265.28
250000	119008566.36	107211627.50	112699814.80
300000	119232729.00	107435790.14	112997753.24
350000	119055922.20	107258983.34	112787009.68
400000	119038325.76	107241386.90	112753002.64
450000	119088166.68	107291227.82	112833974.48
500000	119055769.92	107252412.82	112735675.12

O consumo parcial manteve um valor parecido ao longo da simulação em uma mesma abordagem. Por exemplo, o consumo registrado no tempo 50 mil e 10 mil não são muito diferentes, o reflexo disto está na curva do consumo total.

Como em cada intervalo a Proposta sempre tem um consumo parcial superior ao Valentin e inferior ao Pior Caso, o consumo total tende a manter mesmo padrão. A Figura 5.25 apresenta um comparativo entre a curva de crescimento no consumo de energia de cada abordagem.

A Proposta manteve um consumo total superior ao de Valentin, já que possui um número de maior de ciclos a ser executado e a utilização de um conjunto inicial de frequências cujo consumo é superior ao da Tabela 5.9. Entretanto, a curva do consumo da Proposta ainda se manteve abaixo do Pior Caso, apesar dos custos adicionais ao WCEC devido aos *overheads*.

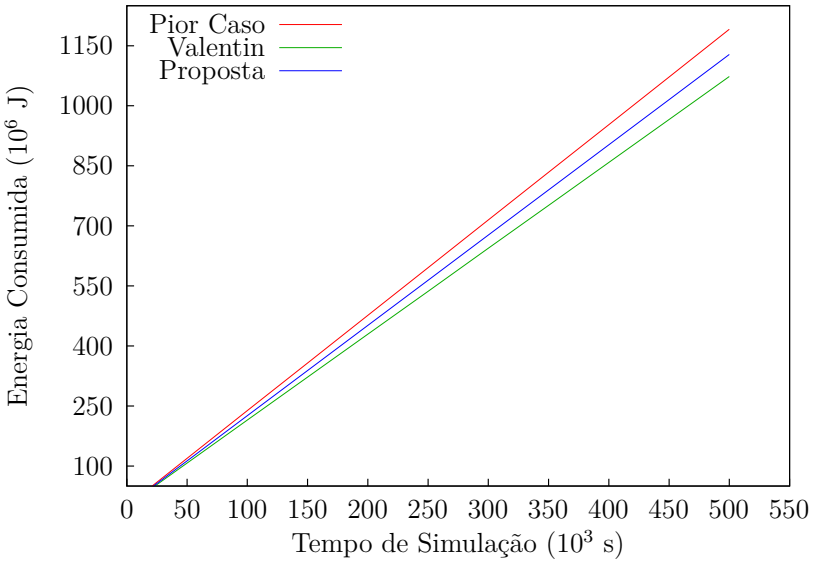


Figura 5.25: Pior Caminho: Comparação do Consumo de Energia.

5.3.2 Caminho Mediano

Durante a execução do caminho mediano, a Proposta apresentou um tempo de término maior do que as demais abordagens, sem infringir o *deadline* (veja Figura 5.26). Resultado bem diferente ao comparar com o Pior Caminho, no qual a Proposta teve um tempo de término menor ao de Valentin.

A diferença neste caso está não só na quantidade de *overheads* executados, mas também nas trocas de frequências, pois, com valores menores, maior será o tempo para executar as instruções.

Na Figura 5.26, contudo, não é possível determinar se a troca de frequência resultou em um consumo de energia menor, já que no primeiro estudo de caso, mesmo com trocas de frequência, o consumo foi maior. As Figuras 5.27, 5.28 e 5.29 apresentam o desempenho de cada abordagem quanto ao consumo de energia durante a simulação.

É importante verificar como a curva do consumo total da Proposta manteve um valor inferior ao das demais abordagens (o valor manteve-se mais próximo da metade entre os intervalos $550 * 10^6$ e $450 * 10^6$ do que Valentin). Entretanto, da mesma forma que o Pior Caminho, não se pode verificar bem a diferença entre os consumos por intervalo já

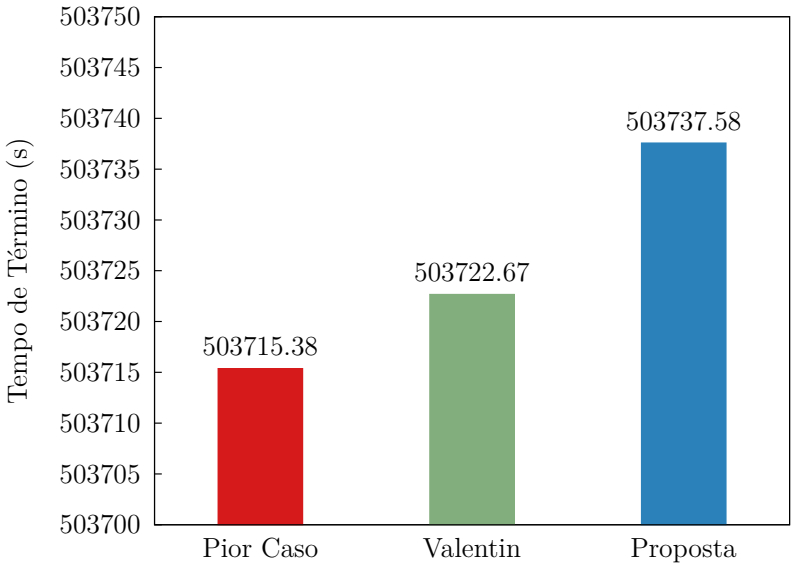


Figura 5.26: Caminho Mediano: Comparação do Tempo de Término.

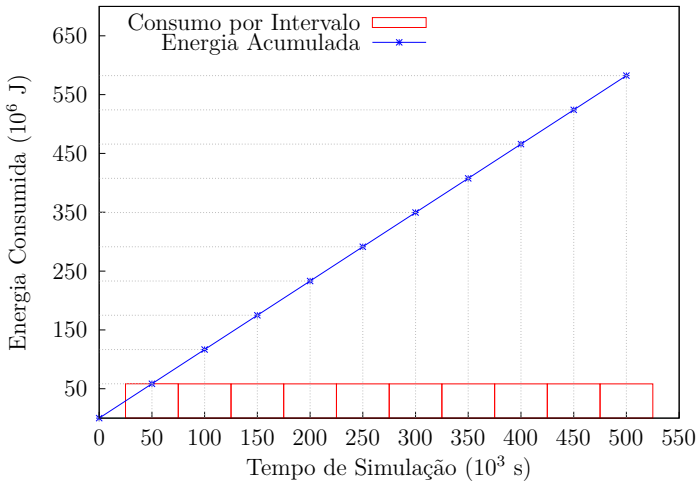


Figura 5.27: Caminho Mediano: Consumo Parcial de Energia do Pior Caso.

que a escala do gráfico deveria ser menor. A Tabela 5.12 apresenta estes valores.

Assim como apresentado nas Figuras 5.27, 5.28 e 5.29, o consumo parcial na Tabela 5.12 para cada abordagem não teve diferenças altas. O consumo realizado pela Proposta estava abaixo dos demais inicialmente, e se manteve com valores ainda baixos até o

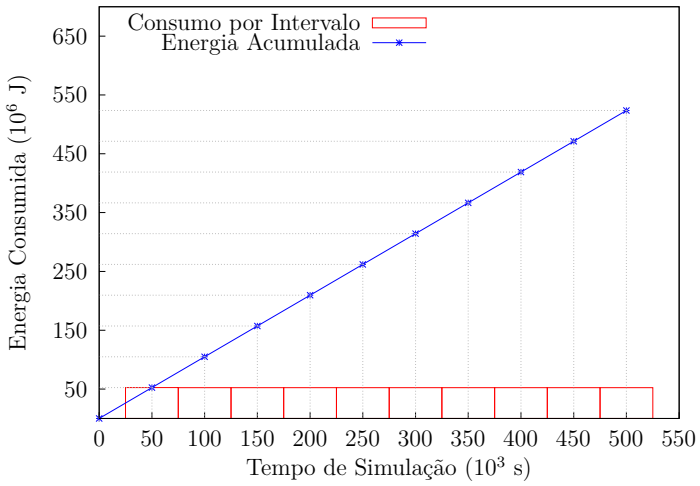


Figura 5.28: Caminho Mediano: Consumo Parcial de Energia: Valentin.

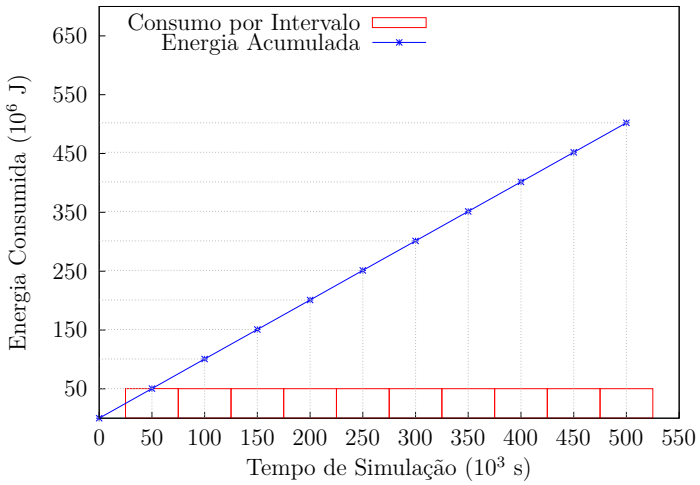


Figura 5.29: Caminho Mediano: Consumo Parcial de Energia da Proposta.

término da simulação. Já o Pior Caso obteve sempre o maior dos consumos, enquanto Valentin teve valores intermediários às duas propostas.

A Figura 5.30 compara as três curvas do consumo total. Ela permite extrair a informação de que trocas não só foram possíveis, como também a Proposta apresentou um consumo total inferior ao das demais abordagens, mesmo com *overheads* e um

Tabela 5.12: Caminho Mediano: Consumo Parcial de Energia.

Tempo (s)	Pior Caso (J)	Valentin (J)	Proposta (J)
50000	58368629.16	52480142.48	50345684.71
100000	58273441.20	52388200.16	50237667.90
150000	58227255.00	52342013.96	50228525.88
200000	58209188.76	52323947.72	50206202.12
250000	58275774.00	52390532.96	50207305.82
300000	58162451.76	52277210.72	50213091.34
350000	58208637.96	52323396.92	50189651.38
400000	58208637.96	52323396.92	50189651.38
450000	58229587.80	52344346.76	50230742.54
500000	58237985.88	52349499.20	50216867.40

conjunto de frequências inicial, cujo consumo é superior ao do Valentin.

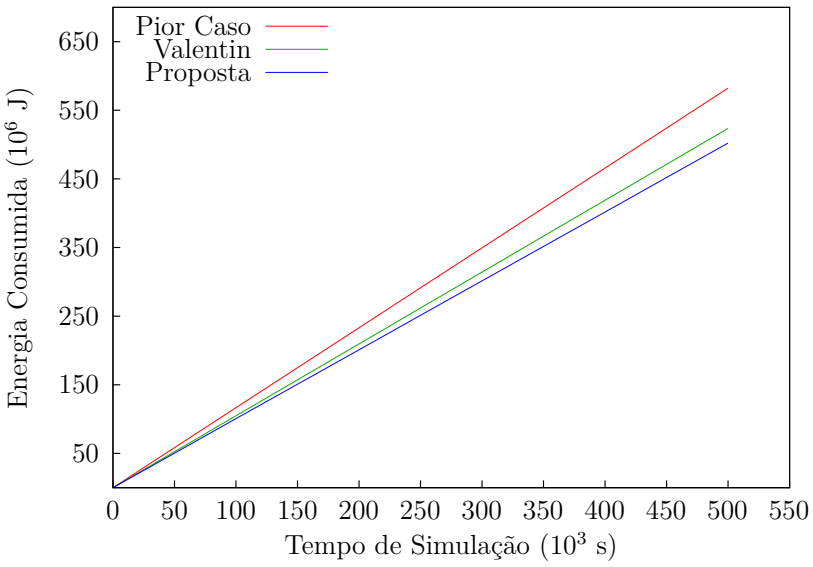


Figura 5.30: Caminho Mediano: Comparação do Consumo de Energia.

5.3.3 Melhor Caminho Aproximado

Outro caso interessante é analisar o tempo de término de cada abordagem na execução do Melhor Caminho Aproximado. A Figura 5.31 mostra que tanto o Pior Caso quanto Valentin terminaram as respectivas execuções ao mesmo tempo. Já a Proposta, devido aos *overheads* e possíveis trocas de frequência, apresentou um tempo de término maior. Contudo, em todos os casos anteriores, Valentin sempre apresentava um tempo de término superior ao Pior Caso e, neste estudo, não. Isto se deve ao cenário em questão.

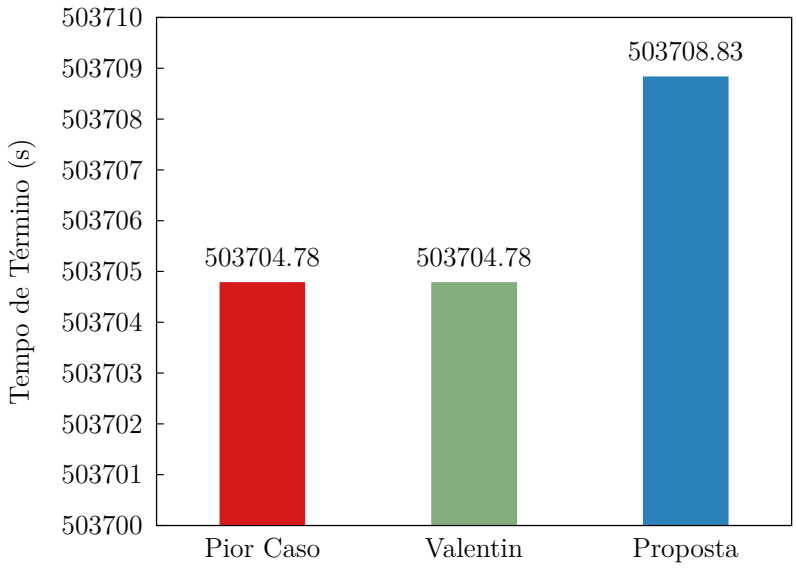


Figura 5.31: Melhor Aproximado: Comparação do Tempo de Término.

O tempo necessário para executar as tarefas é baixo, devido ao Melhor Caminho Aproximado, enquanto o período delas é alto (veja Tabela 5.8). Ao executar cada tarefa com a respectiva frequência, o tempo de execução delas é tão baixo que todas terminam antes do período da tarefa mais prioritária, ou seja, sem preempções.

Além disso, como a última das oito tarefas utiliza a mesma frequência tanto no Pior Caso quanto em Valentin, o tempo de término dela é igual nas duas abordagens e, logo, os tempos de término das simulações são iguais. Este caso apresenta que, mesmo quando Valentin utiliza um conjunto de frequências diferentes do Pior Caso, o tempo de término de ambas as abordagens pode ser igual devido ao modelo de tarefas.

Por outro lado, as Figuras 5.32, 5.33 e 5.34 apresentam os consumos total e parcial de cada abordagem.

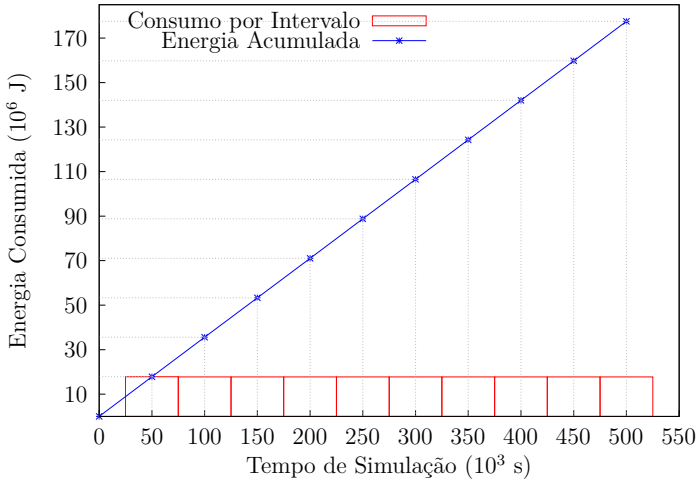


Figura 5.32: Melhor Aproximado: Consumo Parcial de Energia do Pior Caso.

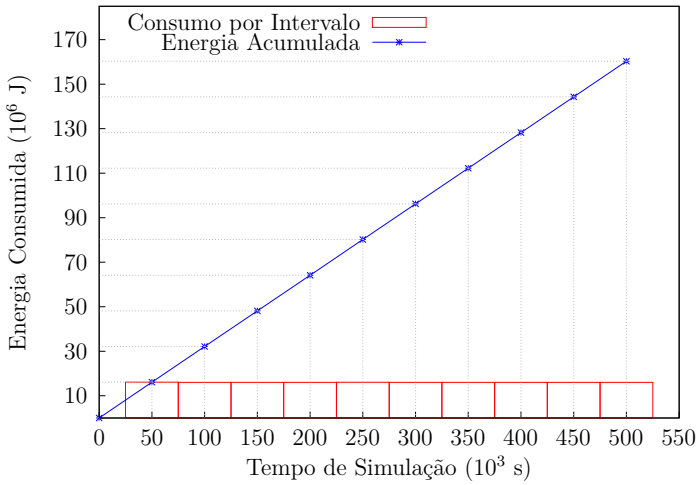


Figura 5.33: Melhor Aproximado: Consumo Parcial de Energia: Valentin.

O Pior Caso manteve um consumo parcial de energia superior ao das demais abordagens, logo, também o consumo total. Já o consumo tanto parcial quanto total da Proposta e Valentin, apresentam valores próximos.

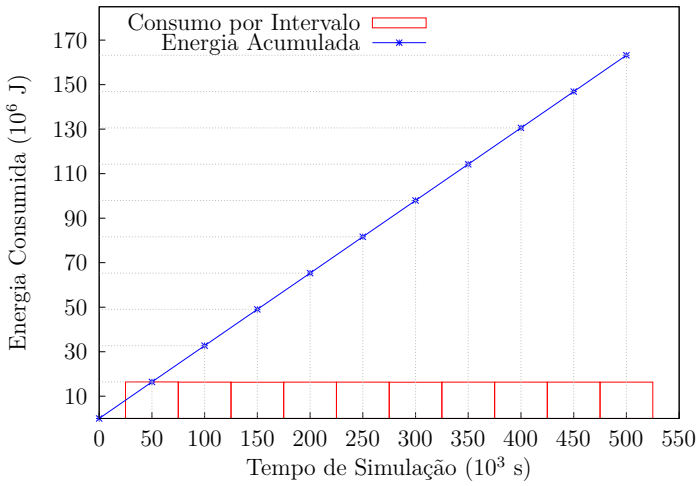


Figura 5.34: Melhor Aproximado: Consumo Parcial de Energia da Proposta.

A Tabela 5.13 apresenta o consumo parcial ao longo da simulação. No tempo de 50 mil segundos, o Pior Caso obteve um consumo superior aos demais, enquanto Valentin, com uma diferença de aproximadamente 288 mil Joules para Proposta, apresenta um consumo inferior. Este comportamento se mantém em cada intervalo da simulação.

Tabela 5.13: Melhor Aproximado: Consumo Parcial de Energia.

Tempo (s)	Pior Caso (J)	Valentin (J)	Proposta (J)
50000	17830743.84	16109561.44	16397638.85
100000	17741384.64	16021232.44	16311278.89
150000	17727199.92	16007047.72	16295570.31
200000	17741384.64	16021232.44	16308532.44
250000	17763047.28	16042895.08	16331452.23
300000	17727199.92	16007047.72	16295570.31
350000	17741384.64	16021232.44	16308532.44
400000	17741384.64	16021232.44	16308532.44
450000	17748862.56	16028710.36	16318490.09
500000	17750534.40	16029352.00	16318385.33

A Figura 5.35 procura comparar a curva do consumo total entre as três abordagens. O Pior Caso, mais uma vez, teve o maior consumo. Já a Proposta apresentou um consumo menor ao Pior Caso, porém maior ao Valentin.

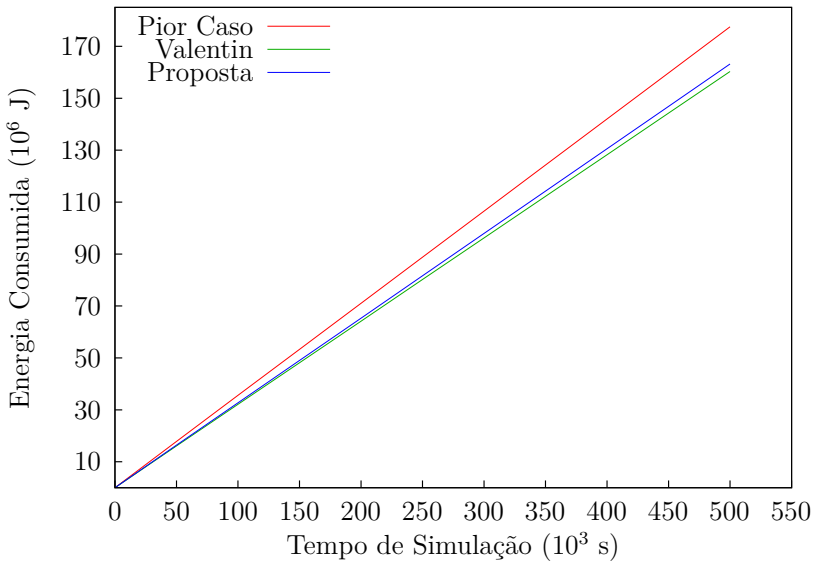


Figura 5.35: Melhor Aproximado: Comparação do Consumo de Energia.

No Melhor Caminho Aproximado do primeiro estudo de caso, a Proposta teve o maior consumo dentre as três abordagens, devido aos *overheads* representarem uma porcentagem significativa do custo total. Apesar do presente estudo de caso utilizar as mesmas três tarefas do primeiro, as cinco tarefas adicionais possuem custos de execução elevados, assim, o custo do *overhead* representa uma parcela bem menor aos 15% do WCEC da tarefa – Tabelas 5.9 e 5.10. Com base nesta informação, quando a troca de frequência foi possível, executou-se um número maior de ciclos com uma frequência menor, o que proporcionou menos consumo de energia. Ainda assim, a Proposta manteve-se com um valor de consumo superior ao Valentin.

É importante observar que, de todas as análises feitas com a Proposta, apenas a do Caminho Mediano do segundo estudo de caso apresentou um consumo de energia menor ao das demais abordagens.

Primeiramente, deve-se destacar que, das oito tarefas utilizadas neste estudo, três são

as mesmas do primeiro. Sabe-se, portanto, que o consumo total de energia ao considerar apenas as três tarefas não foi menor em nenhum dos caminhos ao trocar-se a frequência.

Porém, na análise do Caminho Mediano com o conjunto de oito tarefas neste segundo estudo de caso, o custo total da simulação é superior ao do primeiro. Este custo maior é relevante, pois permite que, quando houver trocas de frequências, mais ciclos sejam executados com frequências menores, o que reflete diretamente no consumo de energia.

Assim, mesmo com a adição de *overheads* e a utilização de um conjunto de frequências iniciais, cujo consumo é maior ao utilizado por Valentin, obteve-se otimizações no consumo de energia.

Capítulo 6

Conclusão

O objetivo principal deste trabalho consistiu em desenvolver uma metodologia para transformação de um código DVFS-*Unaware* para DVFS-*Aware* a partir da análise estática do comportamento de tarefas em sistemas de tempo real crítico. Além disso, não só se procurou garantir o cumprimento temporal das tarefas com otimizações energéticas, como também a escalonabilidade do modelo de tarefas.

6.1 Contribuições

As principais contribuições deste trabalho foram:

- Extração do grafo de fluxo de controle de uma tarefa a partir da análise estática do seu comportamento. Este ponto permitiu o desenvolvimento de uma ferramenta capaz de gerar o CFG de uma tarefa com base na AST de um código em C (Pinheiro, 2015). Além disso, armazena-se neste grafo informações intrínsecas ao código de alto nível, como as linhas iniciais e finais de cada nó, e o custo computacional em termos de ciclos de execução no pior caso;

- Simplificação do CFG para calcular o WCEC de uma tarefa. Este ponto consiste na abstração de todos os nós de um laço como apenas um. A principal vantagem desta abordagem é reduzir significativamente o número de caminhos presentes na execução de uma tarefa com o fim de se identificar o pior caminho, de maneira a resultar em um tempo de computação menor para tal processo;

- Unificação de duas abordagens diferentes do uso da DVFS. Com os estudos de caso apresentados nas Seções 5.2 e 5.3, foi possível verificar como o consumo de energia pode ser reduzido com a troca de frequência durante a execução de uma tarefa, mesmo com a utilização das frequências ótimas iniciais de um modelo de tarefas. A

abordagem inter-tarefa responsabiliza-se pelas frequências ótimas iniciais, enquanto a intra-tarefa preocupa-se em identificar lugares no fluxo de execução de uma tarefa onde se percorre um caminho diferente do pior;

- Transformação de um código DVFS-*Unaware* em DVFS-*Aware*. A principal vantagem desta contribuição é permitir que as informações do CFG possam ser expandidas para adicionar dados de otimizações de energia. Com isto, procura-se retirar de um programador a preocupação com o consumo de energia mediante uma ferramenta que automatiza o processo.

6.2 Pontos Limitantes

Com base nos resultados, seguem algumas limitações do presente trabalho:

- O número de *overheads* é o principal ponto limitante desta dissertação. Ao passo que é possível obter otimizações no consumo de energia com a identificação de pontos nos quais a frequência possa ser alterada, o custo adicional na execução de uma tarefa decorrente dos *overheads* implica um aumento no consumo de energia, inclusive quando o Pior Caminho do CFG de uma tarefa for percorrido. Nesta linha de raciocínio, o consumo com a aplicação do presente trabalho pode ainda ser maior do que a do Pior Caso – utilização da maior frequência disponível no sistema. Isto é possível devido ao conjunto de frequências iniciais ótimas, onde todos os valores podem ser iguais a maior frequência disponível no sistema. Caso este cenário ocorra e haja *overheads* no Pior Caminho de uma tarefa, o consumo de energia desta dissertação pode ter valores altos;

- Devido ao aumento da complexidade nos experimentos, o presente trabalho não considerou compartilhamento de recursos entre tarefas. Um problema comum que este cenário implica é a inversão de prioridades. Contudo, a ferramenta desenvolvida por Valentin (2010) aborda este caso com a aplicação do PCP. Como o conjunto de frequências iniciais utilizadas neste trabalho depende diretamente da resposta da ferramenta de Valentin (2010), se o modelo de tarefas com compartilhamento de recursos for escalonável, será possível aplicar a proposta desta dissertação;

- Com o objetivo de reduzir a complexidade das simulações, algumas variáveis de um ambiente real foram desprezadas, como o consumo de energia no tempo ocioso e o tempo necessário para o processador efetivamente realizar a troca da frequência – este tempo pode variar entre $200\mu s$ e $500\mu s$ em alguns processadores com DVFS, segundo Ishihara *et al.* (2008). Ressalta-se que quando uma tarefa solicita a troca da frequência, ela é preemptada por este evento até a troca de fato da frequência. Neste tempo de transição de frequências, uma tarefa que estava na fila de prontos entra em execução. Porém, está é uma característica do processador e não se considerou nos estudos de caso desta dissertação.

6.3 Trabalhos Futuros

Tem-se como trabalhos futuros:

- Sabe-se que "80% da execução de um programa ocorre em apenas 20% do código" (Ball & Laurus, 2000), também conhecido como regra 80-20. Além disso, ao analisar os resultados desta dissertação percebe-se otimizações no consumo de energia apenas no Caminho Mediano. Em outras palavras, pode ser uma proposta interessante de se otimizar o consumo de energia transformar apenas os caminhos mais comuns em DVFS-*Aware*. Desta forma, verifica-se a identificação dos caminhos do CFG que se encaixam na regra 80-20 como possíveis melhorias para esta proposta;

- A transformação de código é independente se toda a estrutura de uma tarefa encontra-se presente apenas em uma função, ou se está modularizada. Porém, é necessário analisar se a inserção do código DVFS-*Aware* contemplará adequadamente os ciclos executados no retorno de uma chamada de função. Por exemplo, dentro de um laço é possível ter várias chamadas de funções, cada uma possui o respectivo custo em ciclos de *clock*. Ao executar uma iteração do laço, o custo de cada função pode ser um e, na segunda iteração, pode ser totalmente diferente. Por não saber o custo real no retorno de uma função, é importante analisar este caso para identificar o impacto no custo total da tarefa;

- O presente trabalho utiliza como base o *datasheet* do processador em estudo para determinar o custo em ciclos de *clock* de cada instrução em C. Investigar abordagens

diferentes para tornar a solução mais independente do processador em questão, é outro ponto para trabalhos futuros;

- A ferramenta desenvolvida nesta dissertação identifica apenas dois tipos de estruturas em C: as de seleção formados pelo *if* e as de iteração formadas pelo *while* (veja Pinheiro (2015) para mais detalhes). Seria interessante expandir o escopo da ferramenta para reconhecer estruturas do tipo *for*, *do-while* e *switch-case*;

- A proposta desta dissertação consiste mais precisamente em inserir instruções DVFS-*Aware* no código das tarefas dadas como entrada. Um trabalho futuro relevante refere-se a provar que estas inserções não alteraram a funcionalidade do código original. Por exemplo, aplicar testes de cobertura para verificar se todos os testes cobertos pelo código original ainda são válidos no novo código;

- Gonçalves (2015) apresenta uma abordagem colaborativa entre as tarefas de tempo real e o sistema operacional para a troca de tensão e frequência. O trabalho visa a diminuição do tempo de resposta quando ocorrer a troca de tensão e frequência pelo processador. Para isto, Gonçalves (2015) desenvolveu um canal de comunicação entre as tarefas de tempo real e as políticas de escalonamento. É interessante verificar o impacto que a utilização deste canal de comunicação pode vir a causar se for utilizado quando esta dissertação calcular e alterar a frequência atual.

Apêndice A

Biblioteca de Suporte: *cfg_wcec.h*

Este apêndice consiste em apresentar a biblioteca de suporte *cfg_wcec.h* que será incluída sempre que houver a geração de um código DVFS-*Unaware* em DVFS-*Aware*. Esta biblioteca contém funções para calcular a nova frequência se houver a identificação de uma aresta do tipo-B ou tipo-L.

```
/*
 * cfg/cfg_wcec.h
 *
 * It defines how the new frequency should be computed when a type-B or type-L
 * edges are found.
 *
 * By default, the new frequency is always 100. So, ___cfg_get_curfreq() should
 * be changed to return the right frequency when testing it in a real
 * environment. Moreover, ___cfg_typeB_freq() and ___cfg_typeL_freq() should also
 * be changed to set the new frequency.
 *
 * At the user side, only ___cfg_change_freq() must be called with the right
 * parameters to change the frequency if it is possible.
 *
 * Note: type-B and type-L overheads are zero by default.
 */

#ifndef ___CFG_WCEC___
#define ___CFG_WCEC___

typedef enum {
    ___CFG_TYPE_UNKOWN = 0,
    ___CFG_TYPE_B,
    ___CFG_TYPE_L
} ___cfg_edge_type;
```

```
/* Utils */
float ___cfg_get_curfreq(void);
int ___cfg_ceil(float freq);
void ___cfg_change_freq(___cfg_edge_type *type, float rwcec_bi, float rwcec_bj,
        int loop_max_iter, int loop_iter);

/* For Type-B Edges */
float ___cfg_typeB_sur(float rwcec_wsbi, float rwcec_bj);
void ___cfg_typeB_freq(float rwcec_wsbi, float rwcec_bj);

/* For Type-L Edges */
float ___cfg_typeL_cycles_saved(float loop_wcec, int loop_max_iter,
        int loop_iter);
float ___cfg_typeL_sur(float loop_wcec, float rwcec_bout, int loop_max_iter,
        int loop_iter);
void ___cfg_typeL_freq(float loop_wcec, float rwcec_bout, int loop_max_iter,
        int loop_iter);


/* ___cfg_get_curfreq: get processor current frequency
 * @returns: processor current frequency
 */
float ___cfg_get_curfreq(void) {
    return 100; // default frequency
}

/* ___cfg_ceil: implements ceil operation without using external libraries
 * @parameter freq: number that ceil operation should be applied
 * @returns: result of ceil operation
 */
int ___cfg_ceil(float freq) {
    return (freq == (int)freq) ? freq : (int)(freq + 1);
}

/* ___cfg_change_freq: change processor frequency according to the edge type
 * @parameter type: cfg edge type which can be B or L
```

```
 * @parameter rwcec_bi: if edge is of type-B, so it is RWCEC of the worst
 *  successor of bi. However, if it is of type-L, so it is WCEC of one loop
 *  execution
 * @parameter rwcec_bj: if edge is of type-B, so it is RWCEC of bj. However, if
 *  it is of type-L, so it is RWCEC of bout - first node after loop execution
 * @parameter loop_max_iter: maximum number of loop iterations
 * @parameter loop_iter: how many loop iterations were done at runtime
 */
void ___cfg_change_freq(___cfg_edge_type *type, float rwcec_bi, float rwcec_bj,
        int loop_max_iter, int loop_iter) {

    switch(*type) {
    case(___CFG_TYPE_B):
        ___cfg_typeB_freq(rwcec_bi, rwcec_bj);
        break;
    case(___CFG_TYPE_L):
        ___cfg_typeL_freq(rwcec_bi, rwcec_bj, loop_max_iter, loop_iter);
        break;
    case(___CFG_TYPE_UNKOWN):
        break;
    }

    *type = ___CFG_TYPE_UNKOWN;
}


/* ==========================
 * Type-B edges definitions
 * ==========================
 */
float ___cfg_typeB_overhead = 0; /* overhead of type-B operations */

/* ___cfg_typeB_sur: compute speed update ratio from type-B edge
 *       r(bi, bj) = RWCEC(bj) / (RWCEC(WORST_SUCC(bi)) - typeB_overhead)
 * @parameter rwcec_wsbi: RWCEC of the worst successor of bi
 * @parameter rwcec_bj: RWCEC of bj
 * @returns: speed update ratio from a type-B edge
 */
```

```
float ___cfg_typeB_sur(float rwcec_wsbi, float rwcec_bj) {
    if (rwcec_wsbi - ___cfg_typeB_overhead <= 0)
        return 1;

    return rwcec_bj / (rwcec_wsbi - ___cfg_typeB_overhead);
}

/* ___cfg_typeB_freq: compute the new frequency of a type-B edge and apply it if
 * the ratio is less than one. If it is equal or greater than one, the new
 * frequency will be greater than the current one and so it will be the energy
 * consumption.
 * @parameter rwcec_wsbi: RWCEC of the worst successor of bi
 * @parameter rwcec_bj: RWCEC of bj
 */
void ___cfg_typeB_freq(float rwcec_wsbi, float rwcec_bj) {
    float ratio, curfreq;
    int newfreq;

    ratio = ___cfg_typeB_sur(rwcec_wsbi, rwcec_bj);
    if (ratio < 1) {
        curfreq = ___cfg_get_curfreq();
        curfreq = curfreq * ratio;
        newfreq = ___cfg_ceil(curfreq);

        /* change_processor_frequency(newfreq) */
    }
}


/* ========================
 * Type-L edges definitions
 * ========================
 */
float ___cfg_typeL_overhead = 0; /* overhead of type-B operations */

/*
 * ___cfg_typeL_cycles_saved: compute how many cycles were not executed
 *      r(bi, bout) = RWCEC(bout) / (RWCEC(bout) + SAVED(bi) - typeB_overhead)
```

```
 * where bi is loop condition node.
 * @parameter loop_wcec: WCEC of one loop execution
 * @parameter loop_max_iter: maximum number of loop iterations
 * @parameter loop_iter: how many loop iterations were done at runtime
 * @returns: cycles that were not executed from a type-L edge
 */
float ___cfg_typeL_cycles_saved(float loop_wcec, int loop_max_iter,
        int loop_iter) {
    return (float)(loop_wcec * (loop_max_iter - loop_iter));
}

/*
 * ___cfg_typeL_sur: compute speed update ratio from type-L edge
 *       r(bi, bout) = RWCEC(bout) / (RWCEC(bout) + SAVED(bi) - typeB_overhead)
 * where bi is loop condition node.
 * @parameter loop_wcec: WCEC of one loop execution
 * @parameter rwcec_bout: RWCEC of bout - first node after loop execution
 * @parameter loop_max_iter: maximum number of loop iterations
 * @parameter loop_iter: how many loop iterations were done at runtime
 * @returns: speed update ratio from a type-L edge
 */
float ___cfg_typeL_sur(float loop_wcec, float rwcec_bout, int loop_max_iter,
        int loop_iter) {
    float saved = ___cfg_typeL_cycles_saved(loop_wcec, loop_max_iter, loop_iter);
    if (rwcec_bout + saved - ___cfg_typeL_overhead <= 0)
        return 1;

    return rwcec_bout / (rwcec_bout + saved - ___cfg_typeL_overhead);
}

/*
 * ___cfg_typeL_freq: compute the new frequency of a type-L edge and apply it if
 * the ratio is less than one. If it is equal or greater than one, the new
 * frequency will be greater than the current one and so it will be the energy
 * consumption.
 * @parameter loop_wcec: WCEC of one loop execution
 * @parameter rwcec_bout: RWCEC of bout - first node after loop execution
 * @parameter loop_max_iter: maximum number of loop iterations
```

```
 * @parameter loop_iter: how many loop iterations were done at runtime
 */
void ___cfg_typeL_freq(float loop_wcec, float rwcec_bout, int loop_max_iter,
        int loop_iter) {
    float ratio, curfreq;
    int newfreq;

    ratio = ___cfg_typeL_sur(loop_wcec, rwcec_bout, loop_max_iter, loop_iter);
    if (ratio < 1) {
        curfreq = ___cfg_get_curfreq();
        curfreq = curfreq * ratio;
        newfreq = ___cfg_ceil(curfreq);

        /* change_processor_frequency(newfreq) */
    }
}

#endif /* ___CFG_WCEC___ */
```

Apêndice B

Ferramenta *smartenum*

A ferramenta *smartenum* foi um dos resultados do trabalho proposto por Valentin (2010). Esta ferramenta inclui não só a verificação do teste de escalonabilidade, como também realiza as podas definidas pelo autor.

A ferramenta informa se um modelo de tarefas, dado como entrada, é escalonável ou não. Se sim, obtêm-se as frequências iniciais ótimas para aplicar em cada tarefa. O tempo de resposta, a taxa de utilização total do sistema e a redução do consumo de energia ao comparar a execução das mesmas tarefas com a maior frequência, também são informações de saída.

A ferramenta foi adquirida por meio do link *http://repo.or.cz/w/smartenum.git*. Contudo, a versão original contém um pequeno problema. De acordo com o modelo de tarefas dado como entrada, a ferramenta pode informar que o mesmo não é escalonável mesmo que apesar de ser. Este ponto foi discutido com o autor o qual informou que o problema está relacionado a uma comparação de ponto flutuante, especialmente em máquinas de 64-bit. Ele ainda propôs uma correção: alterar a comparação de ponto flutuante no arquivo *smartenum/src/analysis.c*.

A linha 266 do arquivo *smartenum/src/analysis.c* contém as seguintes instruções:

```
if (almostequal2s_complement(Ip, Ipa, 1 << 22))
```

Esta comparação deve ser alterada para:

```
if (Ip <= Ipa && Ip >= Ipa)
```

Outro arquivo importante para se alterar é o *smartenum/src/Makefile.am*. Este arquivo contém as configurações básicas de compilação da ferramenta. Entretanto, *smartenum* tem como dependência o simulador *Akaroa* o qual não será utilizado. Assim, o arquivo de configuração original deve ser alterado de:

```
CC=gcc
CCLD=g++
AM_CFLAGS = -I../include -I/usr/local/akaroa/include
LIBS = -L/usr/local/akaroa/lib -lm -lakaroa -largs\
       -L/usr/local/gnu/lib -lfl -lnsl
bin_PROGRAMS = pseudosim se
se_SOURCES = smartenum.c analysis.c
pseudosim_SOURCES = pseudo-sim.c analysis.c
```

para:

```
CC=gcc
CCLD=g++
AM_CFLAGS = -I../include -I/usr/local/akaroa/include
bin_PROGRAMS = se
se_SOURCES = smartenum.c analysis.c
```

Por fim, na raiz do projeto, a compilação da ferramenta pode ser realizada:

```
$ autoreconf -i
$ ./configure
$ make
$ make install
```

O executável será gerado como *smartenum/src/se* e, assim, a ferramenta pode ser utilizada. Por exemplo, dado um modelo de tarefas como entrada (arquivo *ex.in*), *smartenum* apresenta as informações resumidas sobre a escalonabilidade deste modelo de tarefas como se segue:

```
$ cat ex.in
  3 5 1
  1000    800     600     400     150
  1.8     1.6     1.3     1       0.75
  10707   30      0.4     0
  9563    40      0.4     0
  13951   60      0.4     0
$ ./se -s < ex.in
  Sumario
  Numero de Configuracoes:   125
  Configuracoes Avaliadas:   125
  Configuracoes Viaveis: 3
```

Tempo de processamento: 0s and 449 us
Melhor espalhamento 36.16 com as seguintes frequencias
(1000.00; 1.80) (800.00; 1.60) (1000.00; 1.80)
Utilizacao total do sistema e 88.83%
Energia gasta pelo sistema e 104373.20 x C
Energia gasta pelo sistema e 110876.04 x C se usar apenas a maior frequencia
Reducao de energia: 5.86%

O conteúdo do arquivo de entrada (*ex.in*) corresponde: ao número de tarefas do modelo dado como entrada, seguido do número de frequências e tensões, e a quantidade de recursos compartilhados na primeira linha; na segunda e terceira têm-se as frequências e tensões, respectivamente; na quarta linha em diante é a descrição de cada tarefa com o WCEC, *deadline*, *jitter* e a porcentagem que o primeiro recurso compartilhado refere-se ao WCEC da tarefa, depois a porcentagem do segundo recurso, assim sucessivamente.

Por outro lado, a saída da ferramenta apresentou que para o modelo de tarefas, frequências e tensões dados como entrada, as três tarefas devem adotar 1000MHz, 800MHz e 1000MHz, respectivamente. Esta adoção representa uma redução de energia de 5.86% ao comparar a execução das mesmas tarefas sempre com a maior frequência de entrada, no caso 1000MHz.

Referências Bibliográficas

AbouGhazaleh, N., Mossé, D., Childers, B., Melhem, R., & Craven, M. (2003). Collaborative operating system and compiler power management for real-time applications. *In 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*.

Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A., & Nicolau, A. (2002). Profile-based dynamic voltage scheduling using program checkpoints. *In Design, Automation and Test in Europe*.

Ball, T. & Laurus, R. J. (2000). Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, page 33(7):57–65.

Baums, A. & Zaznova, N. (2008). Power optimization of embedded real-time systems and their adaptability. *Automatic Control and Computer Sciences*, pages 59–73.

Bendersky, E. (2015). pycparser - Complete C99 parser in pure Python. `https://github.com/eliben/pycparser`. Commit: c67c9208d8. Acessado em Julho de 2015.

Burns, A. & Wellings, A. (1997). *Real-Time Systems and Programming Languages*. Addison-Wesley, second edition.

Cohen, D. (2011). Aplicando a técnica escala dinâmica de tensão e frequência baseado em movimento uniforme. *Dissertação de Mestrado, Universidade Federal do Amazonas*.

Cooper, K. D. & Torczon, L. (2012). *Engineering a Compiler*. Elsevier, United States of America, second edition.

Farines, J. M., Fraga, J. S., & Oliveira, R. S. (2000). *Sistemas de Tempo Real*. Universidade Federal de Santa Carina, Departamento de Automação e Sistemas.

Gonçalves, R. (2015). Integrando características preemptivas à técnica de escalonamento dinâmico de tensões e frequências intra-tarefa. *Dissertação de Mestrado, Universidade Federal do Amazonas*.

Gustafsson, J., Betts, A., Ermedahl, A., & Lisper, B. (2010). The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium. OCG.

Heckmann, R., Langenbach, M., Thesing, S., & Wilhelm, R. (2003). The influence of processor architecture on the design and the results of wcet tools. *IEEE Proceedings on Real-Time*.

Ishihara, T., Yamaguchi, S., Ishitobi, Y., Matsumura, T., Kunitake, Y., Oyama, Y., Kaneda, Y., Muroyama, M., & Sato, T. (2008). Ample: An adaptive multi-performance processor for low-energy embedded applications. *Symposium on Application Specific Processors*, pages 83–88.

Kim, W., Kim, J., & Min, L. S. (2002). A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. *In Design, Automation and Test in Europe*.

Kopetz, H. (1992). Scheduling. In Advanced Course on Distributed Systems. Estoril (Portugal).

Lee, W. Y., Ko, Y. W., Lee, H., & Kim, H. (2009). Energy-efficient scheduling of a real-time task on dvfs-enabled multi-cores. *In International Conference on Hybrid Information Technology*, pages 273–277.

Lee, Y. H. & Krishna, C. M. (1999). Voltage-clock scaling for low energy consumption in real-time embedded systems. *Systems and Applications*, pages 272–279.

Levine, J. (2009). *Flex & Bison*. O'Reilly Media Inc., United States of America.

Pering, T., Burd, T., & Brodersen, R. (1998). Dynamic voltage scaling and the design of a low-power microprocessor system. In *In Power Driven Microarchitecture Workshop, attached to ISCA98*.

Pinheiro, D. (2015). cfg - Control Flow Graph of C File. `https://github.com/diegoValhalla/cfg-wcec`. Commit: 9cd6d75840. Acessado em Julho de 2015.

Seo, J., Kim, T., & Dutt, N. D. (2005). Optimal integration of inter-task and intra-task dynamic voltage scaling techniques for hard real-time applications. *IEEE/ACM International conference on Computer-Aided Design*, pages 450–455.

Sha, L., Rajkumar, R., & Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, pages 39:1175–1185.

Shin, D., Lee, S., & Kim, J. (2001). Intra-task voltage scheduling for low-energy hard real-time applications. *In IEEE Design & Test of Computers*.

Tatematsu, T., Takase, H., Zeng, G., Tomiyama, H., & Takada, H. (2011). Checkpoints extraction using execution traces for intra-task dvfs in embedded systems. *IEEE 6th Int'l Symposium on Electronic Design, Test and Applications*, pages 19–24.

Valentin, E. (2010). *smartenum*: Algoritmo *branch-and-bound* para determinação de frequências Ótimas em sistemas com escala dinâmica de tensão e frequência. *Dissertação de Mestrado, Universidade Federal do Amazonas*.

Weiser, M., Welch, B., Demers, A., & Shenker, S. (1994). Scheduling for reduced cpu energy. *In USENIX Symposium on operating systems design and implementation*, pages 13–23.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., & Stenström, P. (2008). The worst-case execution-time problem-overview of methods and survey of tools. *In ACM Transactions on Embedded Computing Systems*, 7(3).

XSCALE (2002). Intel XScale processors. `http://developer.intel.com/design/intelxscale`. Acessado em Julho de 2015.