Universidade Federal de Pernambuco
Centro de Informática - CIn

Pós-graduação em Ciência da Computação

AVALIAÇÃO DE DESEMPENHO E CONSUMO DE ENERGIA DE APLICAÇÕES EMBARCADAS: UMA ESTRATÉGIA BASEADA EM MODELOS DA ARQUITETURA DE HARDWARE E NO CÓDIGO DA APLICAÇÃO

Bruno Costa e Silva Nogueira

DISSERTAÇÃO DE MESTRADO

Recife
5 de março de 2010

Universidade Federal de Pernambuco
Centro de Informática - CIn

Bruno Costa e Silva Nogueira

AVALIAÇÃO DE DESEMPENHO E CONSUMO DE ENERGIA DE APLICAÇÕES EMBARCADAS: UMA ESTRATÉGIA BASEADA EM MODELOS DA ARQUITETURA DE HARDWARE E NO CÓDIGO DA APLICAÇÃO

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática - CIn da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Paulo Romero Martins Maciel*

Recife
5 de março de 2010

Nogueira, Bruno Costa e Silva

Avaliação de desempenho e consumo de energia de aplicações embarcadas: uma estratégia baseada em modelos da arquitetura de hardware e no código da aplicação / Bruno Costa e Silva Nogueira. - Recife: O Autor, 2010.

xix, 82 p. : il., fig., tab.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2010.

Inclui bibliografia e apêndice.

1. Avaliação de desempenho. 2. Redes de Petri - Modelagem de sistemas. 3. Consumo de energia. 4. Sistemas embarcados. I. Título.

004.029 CDD (22. ed.) MEI2010 – 051

Dissertação de Mestrado apresentada por **Bruno Costa e Silva Nogueira** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Avaliação de Desempenho e Consumo de Energia em Sistemas Embarcados: Uma Abordagem Baseada na Arquitetura da Plataforma"**, orientada pelo **Prof. Paulo Romero Martins Maciel** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Ricardo Massa Ferreira Lima
Centro de Informática / UFPE

Prof. Eduardo Antônio Guimarães Tavares
Departamento de Estatística e Informática / UFRPE

Prof. Paulo Romero Martins Maciel
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 05 de março de 2010.

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Aos meus pais

AGRADECIMENTOS

- Meu orientador, Paulo Romero Martins Maciel. A ideia inicial deste projeto foi dele e durante todo mestrado Paulo me incentivou e ensinou, sempre com bastante objetividade, sobre pesquisa, engenharia e escrita. Os seus sábios conselhos foram de grande importância para que eu me transformasse em um engenheiro mais capacitado.
- Amigos do MoDCS, com quem convivi nos últimos 4 anos (desde o período de Iniciação Científica), pelo companheirismo e períodos de descontração. Em especial: Ermeson Carneiro, Gustavo Callou e Julian Menezes.
- Eduardo Tavares e Ricardo Massa, pelos conselhos valiosos que ajudaram a melhorar este trabalho.
- Amigos, família e namorada, Renata Veras, pelo suporte necessário durante todo o período do mestrado.
- E que venha o Doutorado!!!

RESUMO

O projeto de sistemas embarcados usualmente deve levar em consideração diversas restrições não funcionais, tais como: tamanho, peso, custo, confiabilidade e durabilidade. Adicionalmente, com a proliferação de dispositivos portáteis operados por baterias, o projeto de sistemas embarcados de baixo consumo de energia tem despertado muito interesse nos últimos anos. O maior problema é que a evolução da tecnologia das baterias não vem acompanhando a demanda por mais desempenho nestes dispositivos. Além dos aspectos relacionados à demanda do mercado, o aumento da frequência de operação e das funcionalidades colocadas por unidade de área nos circuitos integrados tem levado ao crescimento constante da potência dissipada destes sistemas.

Mudanças de projeto tanto nos componentes de hardware, como de software custam caro e podem levar a atrasos na entrega do projeto. Desta forma, projetistas de sistemas embarcados precisam avaliar suas escolhas nas etapas bem iniciais de projeto e devem caracterizar as otimizações do sistema através de estimativas acuradas.

Esta dissertação apresenta um método para a avaliação de desempenho e consumo de energia para sistemas embarcados. O método proposto adota modelos formais baseados em Redes de Petri Coloridas para modelar o comportamento funcional de processadores e arquiteturas de memória em um alto nível de abstração. Um grupo de aplicações e um microcontrolador de propósito geral foram adotados para demonstrar a aplicabilidade do método. Resultados experimentais demonstram uma exatidão em torno de 96% em comparação com as medidas adquiridas da plataforma real. Adicionalmente, o alto nível de representação comportamental permite a rápida avaliação de consumo de energia e desempenho de sistemas complexos.

Palavras-chave: Sistemas Embarcados, Avaliação de Desempenho e Consumo de Energia, Redes de Petri

ABSTRACT

The design of embedded systems usually must take into account several non-functional constraints, such as performance, size, weight, cost, reliability and durability. Moreover, with the rapid growth of portable devices, the low-power design of embedded systems has gained close attention in recent years. The biggest problem is that the evolution of battery technology is not being able to keep up with the performance requirements. In addition to the market demands, the increase in the operation frequency and transistor density, has led to steady growth in the power consumption of these systems.

Design changes in the software and platform organization at late design phases might be expensive and significantly impact the project delivery. Therefore, designers need to verify their choices at a very early stage in the design flow, and have to characterize the system optimizations through accurate estimates.

This dissertation presents a performance and energy consumption modeling technique for embedded systems. The proposed method adopts a formal model based on Coloured Petri Nets for modeling the functional behavior of processors and memory architectures at a high-level of abstraction. The applicability of the proposed method is illustrated by evaluating a set of applications and a general-purpose microcontroller. Experimental results demonstrate an average accuracy of 96% in comparison with the respective measures acquired from the real hardware platform. Moreover, the high-level behavioral representation of platforms allows the rapid analysis of performance and energy consumption of complex systems.

Keywords: Embedded Systems, Performance and Energy Consumption Evaluation, Petri Nets

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

Atualmente os sistemas embarcados estão presentes em praticamente todas as áreas de nosso cotidiano, muitas vezes sem que nós os percebamos. Exemplos desses sistemas podem ser encontrados em diversos dispositivos eletrônicos comuns como, por exemplo: celulares, geladeiras, câmeras digitais, alarmes e calculadoras, sendo utilizados para dar o suporte necessário para que funções sofisticadas sejam disponibilizadas nestes dispositivos. Com o surgimento de novas tecnologias e metodologias de desenvolvimento, a presença de tais sistemas em nossas vidas tende a crescer cada vez mais.

Alguns números ajudam a dimensionar a importância dos sistemas embarcados nos dias de hoje. Em [Kri05], o mercado mundial de sistemas embarcados foi avaliado em $ 45,9 bilhões em 2004, com previsão de dobrar e chegar a $ 88,8 bilhões em 2009. De acordo a pesquisa feita em [art04], no ano de 2004 existiam por volta de três dispositivos embarcados por pessoa no mundo, e segundo [FFY05], 98% dos processadores produzidos no ano de 2005 eram empregados em sistemas embarcados. A Figura 1.1 exibe a participação dos processos de desenvolvimento e implementação de um sistema embarcado no custo final de desenvolvimento de um produto, considerando diversas áreas de aplicação desses sistemas [H+05].

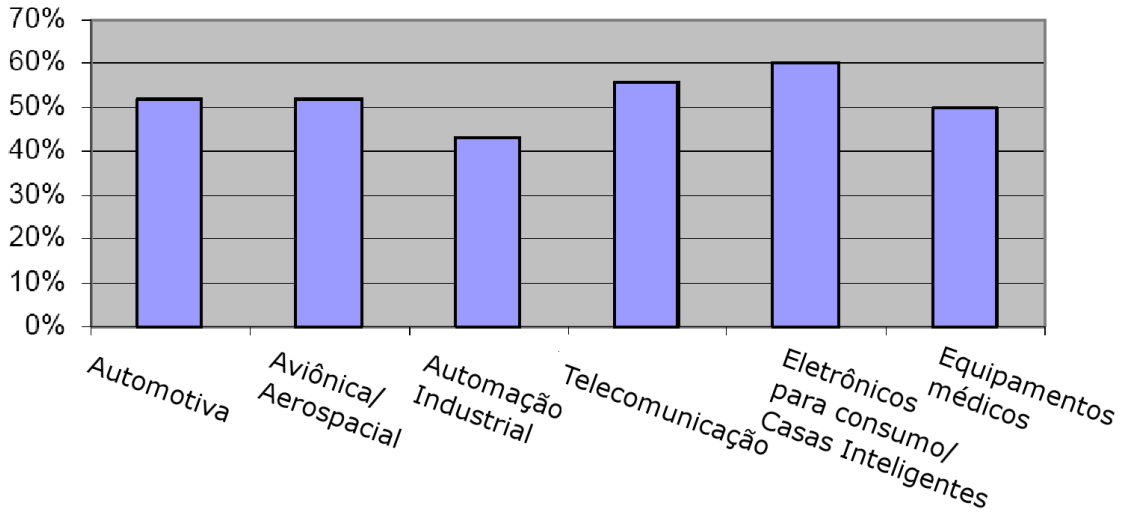

Figura 1.1 Participação dos sistemas embarcados no custo final [H+05].

Usualmente o desenvolvimento de projetos para sistemas embarcados deve considerar diversas restrições não funcionais, tais como: tempo, tamanho, peso, custo, confiabilidade e durabilidade. O rápido crescimento dos sistemas embarcados em novas áreas de

aplicação introduz novas restrições, que por sua vez geram novos desafios técnicos e científicos. Uma proeminente restrição está relacionada aos dispositivos portáteis operados por baterias, nos quais o consumo de energia desempenha um papel importante. Com a proliferação de tais dispositivos, o baixo consumo de energia em sistemas embarcados tem despertado muito interesse nos últimos anos. O maior problema é que a evolução da tecnologia das baterias não vem acompanhando a demanda por mais desempenho nestes dispositivos [JML06]. Além dos aspectos relacionados à demanda do mercado, o aumento da frequência de operação e das funcionalidades colocadas por unidade de área nos circuitos integrados tem levado ao crescimento constante da potência dissipada destes sistemas.

Não obstante, consumo de energia também vem ganhando bastante atenção em sistemas computacionais em geral, principalmente os de alto desempenho, como *data centers*. Nestes sistemas, quanto mais alto o pico de potência, mais elaborado deve ser a infraestrutura de resfriamento [BR04]. [Con03] estima que os equipamentos de potência, eletricidade e resfriamento são responsáveis por 63% do custo total de aquisição da infraestrutura de TI de um *data center*.

Mudanças de projeto tanto nos componentes de hardware, como de software, custam caro e podem levar a atrasos na entrega do projeto. Portanto, projetistas de sistemas embarcados precisam avaliar suas escolhas nas etapas bem iniciais de projeto, e devem caracterizar as otimizações no consumo de energia do sistema através de estimativas acuradas. Assim, métodos de estimativa de consumo de energia acurados e rápidos são cruciais para o desenvolvimento de um projeto de sistema embarcado de sucesso, pois eles permitem avaliar e prever se o projeto atenderá aos requisitos de consumo de energia antes que seja tarde demais.

Além do consumo de energia, desempenho é outra importante métrica na maioria dos projetos para sistemas embarcados. No entanto, desempenho e consumo de energia são métricas conflitantes: otimizando uma, tipicamente leva a uma degradação na outra [VG02]. Isso torna importante a avaliação simultânea das duas métricas para que se permita um balanço adequado entre elas.

Frequentemente a estimativa de desempenho e consumo de energia em projetos de sistemas embarcados é feita através do uso de protótipos. A utilização de protótipos é a maneira mais direta de se estimar estas métricas, além disso, ela provê resultados com bastante exatidão. No entanto, o custo de desenvolvimento de um protótipo e o tempo despendido para a sua criação torna impossível a análise de todas as possíveis soluções em produto comercial. Isso dificulta a tomada de decisão do projetista. Neste contexto, a avaliação através de modelos é uma interessante alternativa.

Na avaliação de consumo de energia e desempenho através de modelos, o nível de abstração do modelo adotado está diretamente relacionado à exatidão dos resultados obtidos. Em um modelo de baixo nível, os resultados são, na maioria das vezes, mais exatos que em modelos de alto nível, uma vez que mais detalhes arquiteturais são utilizados neste tipo de modelo. No entanto, avaliação através de modelos de baixo nível é demorada, mais custosa e menos flexível, dificultando a análise nas etapas iniciais do projeto. Além disso, quanto mais o baixo nível de abstração, obviamente, maiores detalhes da implementação do sistema real são necessários. No entanto, detalhes implementação nem

sempre estão disponíveis.

Na literatura, diversas abordagens têm sido desenvolvidas para modelar o consumo de energia em sistemas embarcados. Estes trabalhos podem ser classificados em: (i) modelos no nível arquitetural, e (ii) modelos no nível de instrução. No primeiro, o consumo de energia é calculado a partir de descrições detalhadas que podem compreender o nível de circuito (*circuit level*), nível lógico (*gate level*) , RTL (*register transfer level*) ou nível de sistema (*system level*). Por outro lado, modelos no nível de instrução lidam apenas com instruções e unidades funcionais do ponto de vista do software e sem o conhecimento detalhado do hardware subjacente [BCSRM06].

1.1 OBJETIVOS E ABORDAGEM PROPOSTA

Considerando o contexto apresentado, o objetivo principal desta dissertação é a concepção de uma abordagem apoiada em modelos para avaliação de desempenho e consumo de energia em sistemas embarcados.

Mais especificamente, os objetivos desta dissertação são:

- Propor uma abordagem para estimativa de consumo de energia e desempenho do sistema ainda em tempo de projeto, i.e., sem a necessidade da implementação real para obtenção destas medidas;
- Propor modelos flexíveis e de alto nível que permitam a avaliação das métricas em um código ANSI C executando em uma dada arquitetura embarcada;
- Desenvolver uma ferramenta automática que, apoiada nos modelos propostos, estime consumo de energia e desempenho sem que o projetista necessite conhecer os detalhes dos modelos e o formalismo destes.

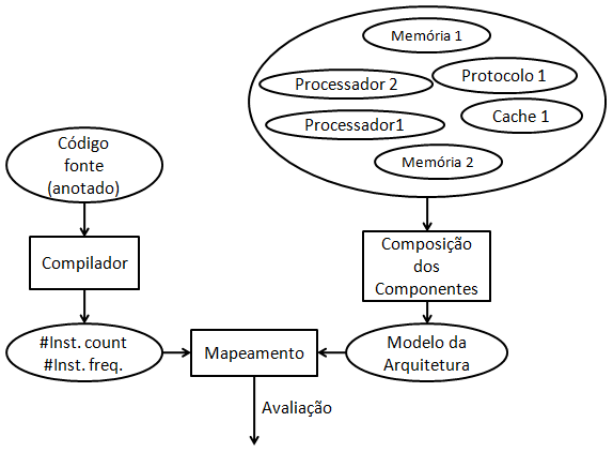

Figura 1.2 Método proposto.

A Figura 1.2 exibe uma visão geral do método proposto por este trabalho, no qual as Redes de Petri Coloridas (CPN) [JK09] são utilizadas para modelar a arquitetura da

plataforma embarcada. A construção do modelo arquitetural é feita através da composição de blocos básicos construídos em CPN. Os blocos básicos representam unidades funcionais da arquitetura em avaliação e são modelados em um alto nível de abstração, permitindo assim, flexibilidade, re-uso e rapidez na avaliação. Estes blocos básicos são anotados com os valores de consumo de energia e desempenho das unidades funcionais modeladas. Em seguida, o código que executará na plataforma embarcada é mapeado no modelo arquitetural através de um compilador, desenvolvido neste trabalho. Por último, a avaliação do modelo é feita através de simulação estocástica, levando em consideração a precisão desejada para obtenção dos valores das métricas.

Para fins de validação, este trabalho considerou um microcontrolador baseado na arquitetura ARM, chamado LPC2106 [Phi04]. O método proposto foi aplicado para avaliar consumo de energia e desempenho de aplicações de diversos tamanhos e características, levando em consideração a arquitetura de memória da plataforma e seus efeitos sob as métricas. Posteriormente, um estudo de caso considerando arquiteturas multiprocessadas (vários processadores) é analisado.

Este trabalho adotou um ambiente de medição para validar o método proposto. Assim, os valores reais de desempenho e consumo de energia foram medidos na plataforma e comparados com as estimativas do método proposto. Adicionalmente, uma ferramenta foi desenvolvida para automatizar o processo de estimativa. Esta ferramenta libera o projetista da necessidade de conhecer o formalismo das CPN, permitindo assim, que os detalhes dos modelos sejam transparentes ao usuário.

O método proposto por este trabalho, assim como resultados preliminares, foi publicado nos anais do *21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2009)*. O leitor deve consultar [NMT+09] para maiores informações.

As Redes de Petri [Mur89] são um ótimo método para modelar arquiteturas de computadores, uma vez que paralelismo e conflito, duas características presentes na maioria dos sistemas computacionais modernos, são facilmente modeladas utilizando este formalismo. Além disso, extensões aos modelos originais das Redes de Petri, como as CPN, têm mostrado uma poderosa técnica para avaliar índices de desempenho em sistemas computacionais [Wel02].

1.2 METODOLOGIA DE DESENVOLVIMENTO PARA SISTEMAS EMBARCADOS - MEMBROS

O método proposto por este trabalho está inserido em um projeto maior desenvolvido pelo grupo de pesquisa MODCS [mod], que visa o desenvolvimento de uma metodologia, apoiada por ferramentas, para o desenvolvimento de sistemas embarcados de tempo-real com restrições de energia. A metodologia proposta se chama MEMBROS (*Methodology for Embedded Critical Software Construction*) e a seguir, serão discutidos brevemente cada atividade do fluxo de tarefas e seus respectivos artefatos de entrada e saída.

A Figura 1.3 apresenta as atividades principais da metodologia MEMBROS, cuja organização é feita em três grupos: (i) validação de requisitos, (ii) avaliação de desempenho, e (iii) síntese de software.

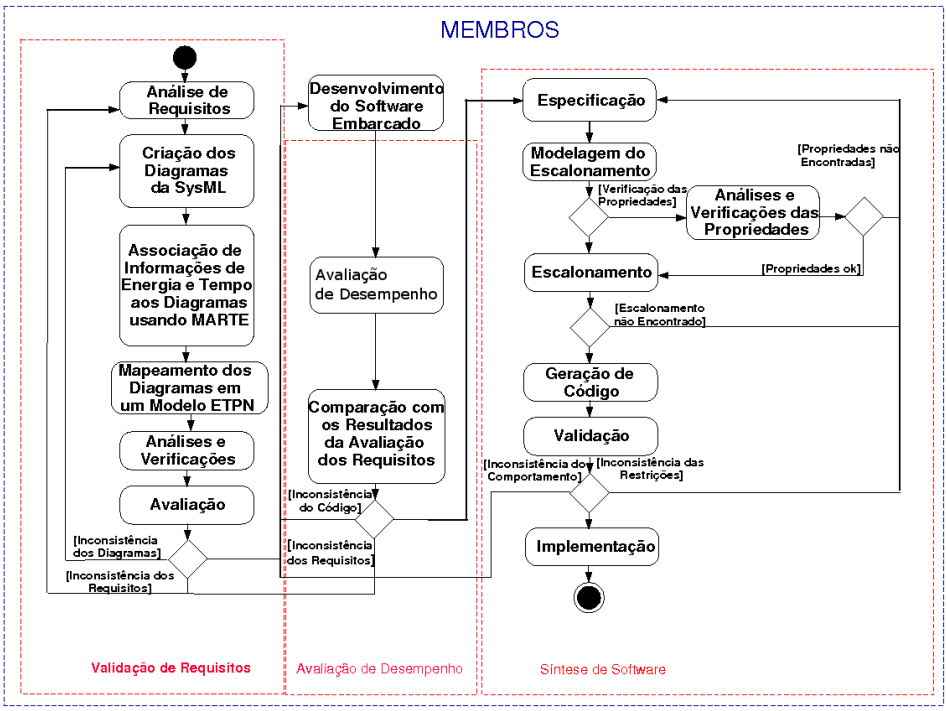

Figura 1.3 Metodologia MEMBROS.

O primeiro grupo está relacionado às atividades de validação dos requisitos. Depois da análise de requisitos, o sistema é projetado utilizando um conjunto de diagramas SysML (*Systems Modeling Language Diagrams* - SDs) [FMS06]. A SysML é uma extensão da UML para a engenharia de sistemas. Estes diagramas representam as funcionalidades do projeto a ser codificado e provêem ao desenvolvedor uma linguagem amigável e intuitiva para modelagem dos requisitos.

Uma vez que restrições de tempo e consumo de energia são de extrema importância no desenvolvimento de sistemas embarcados, os diagramas SysML são anotados com informações sobre o consumo de energia e tempo de execução, utilizando MARTE (*Modeling and Analysis of Real-Time and Embedded Systems*). Em seguida, o diagrama SysML anotado é automaticamente mapeado em uma Rede de Petri Temporizada com restrições de Energia (ETPN) [TMSO08]. De posse do modelo ETPN, análises quantitativas e qualitativas das propriedades do modelo são realizadas, permitindo a validação dos requisitos do projeto. É importante ressaltar que as análises qualitativas capturam os aspectos lógicos da evolução dos sistemas. Entre as propriedades de interesse, pode-se destacar, por exemplo, a análise da existência ou ausência de *deadlocks*, repetitividade e a reversibilidade. As análises quantitativas, por outro lado, têm por principal objetivo a análise de desempenho. Mais informações sobre este grupo de tarefas podem ser encontradas em [AMCN09, And09].

Terminada a fase de projeto do sistema, inicia-se a fase de codificação e também o segundo grupo de atividades da metodologia. Este grupo de tarefas tem como objetivo estimar o consumo de energia e tempo de execução do código do projeto. As atividades

deste grupo correspondem às atividades nas quais este trabalho está inserido e que serão detalhadas em Capítulos específicos desta dissertação (ver Capítulo 4 e Capítulo 5).

O terceiro grupo de atividades, chamado síntese de software, consiste em traduzir uma especificação de alto nível em código fonte, incluindo todo o suporte operacional necessário para execução do software embarcado. A síntese de software também é conhecida como geração automática de código.

As atividades de síntese do software são compostas por dois subgrupos de atividades: (i) tratamento das tarefas, e (ii) geração de código. O tratamento das tarefas é responsável pelo escalonamento das tarefas, gerência dos recursos, e a comunicação inter-tarefa. A geração de código trata da geração estática do código fonte final, que inclui um mecanismo auxiliar de suporte em tempo de execução, denominado despachante. É importante ressaltar que o conceito de tarefa é semelhante à de um processo, no sentido de que é uma unidade ativada concorrentemente durante a execução do sistema. Para as seguintes atividades, assume-se que o software foi implementado como um conjunto de tarefas concorrentes de tempo-real.

De posse das informações do tempo de execução das tarefas, bem como as informações sobre o consumo de energia do hardware obtidas através do grupo de tarefas anterior, o projetista especifica as restrições do sistema. Esta especificação consiste de um conjunto de tarefas concorrentes com suas respectivas restrições, descrições comportamentais, informações relacionadas à plataforma alvo (tensão/níveis de frequência), bem como as restrições de energia. Em seguida, a especificação é traduzida em um modelo ETPN capaz de representar as atividades concorrentes, as informações de tempo de execução, as relações inter-tarefa, tais como a exclusão mútua e de precedência, bem como as restrições de energia.

Após gerar o modelo, o projetista pode primeiramente escolher realizar a análise /verificação das propriedades ou realizar atividade de escalonamento das tarefas. O método proposto adota um método *off-line* de escalonamento para encontrar uma escala exeqüível que satisfaça as restrições de tempo, energia e relações inter-tarefa. Mais especificamente, o algoritmo de escalonamento adotado na modelagem é um algoritmo de busca em profundidade que tenta encontrar uma escala exeqüível a partir do modelo ETPN. Em seguida, a escala encontrada é passada para um mecanismo de geração automática de código, o qual produz um código customizado e previsível para a aplicação. O código resultante contém apenas os serviços estritamente necessários para execução da aplicação, ou seja, a etapa de geração de código tem como resultado o código do despachante customizado para a execução das tarefas de tempo real crítico. Por último, a aplicação é validada na plataforma a fim de verificar o comportamento do sistema, bem como as respectivas restrições. Uma vez validado o sistema, ele pode ser implantado em um ambiente real. Mais informações sobre este grupo de tarefas podem ser encontradas em [TMSO08, Tav09].

1.3 ESTRUTURA DA DISSERTAÇÃO

Este trabalho está organizado como se segue:

O Capitulo 2 apresenta os trabalhos relacionados. O Capítulo 3 descreve os conceitos básicos para um melhor entendimento deste trabalho. O Capítulo 4 apresenta o método

proposto de modelagem da arquitetura. O método proposta assim como a ferramenta desenvolvida por este trabalho para estimar consumo de energia e desempenho são apresentadas no Capítulo 5. O Capítulo 6 apresenta os estudos de caso e o Capítulo 7 conclui este trabalho.

CAPÍTULO 2

TRABALHOS RELACIONADOS

Em sistemas embarcados, principalmente os portáteis, consumo de energia é uma das métricas mais importantes, uma vez que as baterias só podem armazenar uma quantidade limitada de energia. Esta métrica pode ser calculada pela expressão:

$$E = \int P(t)dt \tag{2.1}$$

Onde $P(t)$ é a potência no tempo t (medida em Watts), i.e., a taxa de consumo de energia no tempo t. Percebe-se que reduzir a potência não significa necessariamente que o consumo de energia também será reduzido, já que o tempo total de execução tem um papel fundamental.

Em processadores baseados na tecnologia CMOS, potência pode ser modelada por algumas equações [BB95]. A potência em circuitos CMOS possui componentes estáticos e dinâmicos e pode ser calculada pela equação abaixo [Mud01]:

$$P = ACV^2f + \tau AVI_{short}f + VI_{leakage} \tag{2.2}$$

O primeiro termo captura o componente predominante do consumo, onde A é o nível de atividade (chaveamento) das portas lógicas (*gates*); C, a capacitância vista pelas portas lógicas; f, a frequência de operação; e V, a tensão de alimentação. O segundo termo caracteriza a potência decorrente da corrente de curto-circuito I_{short} que, durante o tempo τ, flui entre a fonte de alimentação e o terra durante o chaveamento da porta. O terceiro termo mede a potência desperdiçada devido à corrente de fuga $I_{leakage}$ que está presente independentemente do estado do circuito.

Como pode ser observado pela Equação 2.3, devido à relação quadrática entre P e V, reduções na tensão de alimentação implicam em grandes reduções na potência. No entanto, a frequência máxima de operação depende linearmente da tensão de alimentação (ver Equação 2.3 [Mud01]), assim, diminuir a tensão de alimentação implica em maiores tempos de execução. Pode-se também diminuir a tensão de limiar $V_{threshold}$ (*threshold voltage*), porém uma diminuição neste valor acarreta em um aumento na corrente de fuga $I_{leakage}$.

$$f_{max} \propto (V - V_{threshold})^2/V \tag{2.3}$$

Outra forma eficiente de diminuir a potência é reduzir o nível de atividade do circuito. Frequentemente isso é feito desligando partes ociosas do sistema.

Em última análise, os trabalhos relacionados a consumo de energia podem ser classificados em: (i) técnicas para reduzir consumo de energia, e (ii) técnicas para modelar e estimar consumo de energia. Este trabalho se concentra no último tipo, que por sua vez pode ser classificado em duas abordagens: (i) modelos no nível arquitetural, e (ii) modelos no nível de instrução. Neste capítulo, os trabalhos relacionados serão apresentados de acordo com esta classificação.

2.1 MODELOS NO NÍVEL DE INSTRUÇÃO

Baseando-se no fato de que o consumo de energia é diretamente proporcional ao software executado. Nos modelos no nível de instrução, o cálculo do consumo de energia lida apenas com instruções e unidades funcionais do ponto de vista do software e sem o detalhamento do hardware subjacente.

O primeiro método adotando esta abordagem, também conhecida como ILPA (*Instruction Level Power Analysis*), foi apresentado por Tiwari et al. [TMW94, TMWTCL96, TL98], na qual um modelo para estimar o consumo de energia em processadores de uso geral (ex.: Intel 486DX2 e Fujitsu SPARClite 436) e de aplicação específica (ex.: Fujitsu DSP) foi desenvolvido. Neste método, um custo de energia é associado a todas as instruções do conjunto de instruções da arquitetura através da medição da corrente média consumida pelo processador quando ele executa esta instrução. Para medir a corrente média, um programa com um *loop* infinito contendo em seu corpo muitas instâncias da instrução em análise é usado, em seguida, a corrente média consumida pelo processador ao executar este *loop* é medida utilizando um amperímetro digital. Efeitos inter-instrução também são considerados, os quais podem ser de dois tipos: (i) estado do circuito, e (ii) restrição de recursos. O primeiro tipo leva em conta o fato de que o consumo de um par de instruções diferentes e consecutivas não é igual à soma dos consumos individuais. Esta diferença ocorre devido ao *overhead* na transição entre o último estado interno do processador, referente à primeira instrução, e o estado seguinte, referente à segunda instrução (ver Figuras 2.1 e 2.2).

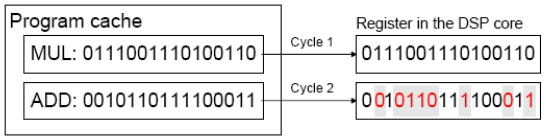

Figura 2.1 Execução de duas instruções diferentes [SBN05].

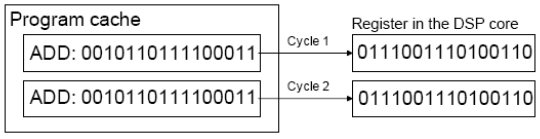

Figura 2.2 Execução de duas instruções iguais [SBN05].

Em um certo estágio, geralmente as instruções são transferidas da memória para registradores internos do processador para posterior processamento. A Figura 2.1 exibe o que ocorre em um dado registrador interno do processador quando a instrução ADD é substituída pela instrução MUL. Neste exemplo, os bits em cinza representam os bits que sofreram transição. Mudanças como esta ocorrem em diversos estágios do *pipeline*, demandando um custo adicional de energia que muitas vezes não pode ser desconsiderado.

Por outro lado, quando duas instruções consecutivas são iguais, nenhum bit precisa mudar de estado (ver Figura 2.2).

O segundo tipo de efeito inter-instrução considera restrições de recursos que podem levar, por exemplo, a bolhas no *pipeline* e *cache misses*. Assim, o consumo de energia é estimado pela equação:

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_h E_h \tag{2.4}$$

onde B_i é o custo individual da instrução i; N_i é o número de ocorrências da instrução i; $O_{i,j}$ é o *overhead* do efeito inter-instrução do tipo estado do circuito para instruções consecutivas i e j e $N_{i,j}$ o número de ocorrências do par de instruções i seguido por j; E_h representa os efeitos inter-instrução do tipo restrição de recursos.

Apesar da boa exatidão do método proposto por Tiwari et al. (erros em torno de 2% a 4%), o tempo necessário para caracterizar uma arquitetura (i.e., obter os valores dos custos B_i, $O_{i,j}$ e E_h) é uma grande desvantagem, já que para obter os valores $O_{i,j}$ todas as possíveis combinações de pares de instruções devem ser medidas.

Depois do trabalho seminal de Tiwari, diversos trabalhos utilizando estratégias similares foram propostos para estimar consumo de energia em outras arquiteturas (ex.: 8051 [JNM+06], ARM7TDMI-S [CMC+09], M3DSP [LWD02], Hitachi SH-4 [SC01], i960 [RJ98] e Motorola 68HC11 [CG99]) e diminuir o tempo de caracterização.

Na direção de diminuir o tempo de caracterização da abordagem de Tiwari, trabalhos como [SC01, NKN+02] propõem simplificar o processo de caracterização agrupando instruções similares em classes de instruções, assim o número de medições depende do tamanho do conjunto de classes definidas, e não do tamanho do conjunto de instruções. Em [JNM+06, CMC+09], os efeitos inter-instrução do tipo estado do circuito são desconsiderados. Trabalhos como [RJ98, SC01], nos quais são relatados erros em torno de 8%, propõem maiores simplificações ao desconsiderar também o consumo individual das instruções e levar em conta apenas a tensão de alimentação e frequência de operação do processador. Em [KTSN98], o modelo NOP é apresentado. Este modelo se baseia no fato de que, para o processador modelado, o efeito do tipo estado do circuito não depende fortemente da instrução consecutiva, e sim se elas são iguais ou não. Assim, dependendo da instrução anterior, o consumo de energia de uma dada instrução é definido como B_{instr} ou $B_{instr} + O_{inst}$, onde O_{inst} é o *overhead* adicionado caso a instrução anterior e a atual não sejam iguais. Na abordagem original de Tiwari, o custo O_{inst} deve ser medido para cada combinação de pares de instrução (no mínimo proporcional a $O(N^2)$), no modelo NOP, este custo deve ser medido apenas uma vez (proporcional a $O(N)$) para cada instrução, e esta medição é baseada no *overhead* do par: instrução NOP + instrução alvo. Ribeiro [Rib07] propõe associar custos às estruturas básicas da linguagem C (como atribuições, operações aritméticas, condicionais e lógicas), ao invés de medir instruções. É importante ressaltar que os erros de estimativa produzidos pelas simplificações propostas por estes trabalhos dependem do processador analisado.

Alguns métodos também foram desenvolvidos no sentido de melhorar o ambiente de medição proposto por Tiwari. [RJ98] propõe o uso de métodos estatísticos para obter dados de corrente média com maior confiança. Em [NKN+02], ao invés de medir a corrente

média, adota-se um espelho de corrente de alto desempenho, no qual uma junção bipolar de transistores é utilizada como circuito de detecção de variação de corrente.

Baseado no modelo de instrução de Tiwari, Nogueira et al. [JNM+06, OJ06] propuseram uma abordagem de simulação utilizando Redes de Petri Coloridas (CPN). Este método adotou as CPN para modelar o fluxo de controle da aplicação e associou probabilidades as instruções de desvio condicional (traduzidas em transições de guarda em CPN). A principal desvantagem do método proposto é a complexidade dos modelos, que cresce com o tamanho da aplicação, causando assim um grande impacto no tempo de simulação. Tal restrição torna impraticável a avaliação de códigos grandes. Este trabalho foi estendido em [CMC+09, Cal09], onde os modelos foram simplificados. Apesar do tempo de simulação ter diminuído, experimentos mostram que ele ainda é fortemente afetado pelo tamanho do código.

Um outro modelo de instrução, conhecido como FLPA (*Functional-Level Power Analysis*), foi introduzido por Laurent et al. [LSJM01] e depois estendido em [SLJM04]. Nesta abordagem, o processador é separado em blocos funcionais, como unidade de busca de instrução, unidade de processamento e memória interna. Em seguida, uma análise inicial é feita e os blocos funcionais que não possuem grande impacto no consumo total são descartados. Depois, verifica-se que fatores (ex.: frequência de operação do processador) influenciam a potência consumida em cada bloco. Em seguida, a potência dissipada por cada bloco, em função destes fatores, é caracterizada por funções matemáticas obtidas através de medições e/ou simulações. Assim, a potência total é obtida através da soma da potência dissipada em cada bloco. Apesar da facilidade para realizar o processo de caracterização e da boa exatidão para estimar potência, o método proposto possui limitações para estimar tempo de execução, que por usa vez afeta a estimativa de consumo de energia, como apresentado pelos resultados apresentados em [SLJM04].

O modelo inicial proposto por Laurent et al. não considera processadores nos quais o consumo de energia é fortemente dependente da instrução executada. Diante disso, [BBR+07] propõe a união da facilidade de caracterização do FLPA com a exatidão do ILPA, proposto por Tiwari. Este método ficou conhecido como FLPA/ILPA e foi aplicado em processadores como ARM940T [BBR+07], OMAP5912 [BBR+07] e LEON2 (implementado em um FPGA) [ZHD+07].

2.2 MODELOS NO NÍVEL DA ARQUITETURA

Os modelos no nível da arquitetura utilizam descrições do hardware para realizar a estimativa de consumo de energia. Estes métodos já existem há alguns anos e podem atuar em diversos níveis abstração como, por exemplo, no nível de circuito (*circuit-level*), nível lógico (*gate-level*), RTL (*register-transfer level*) ou nível de sistema (*system-level*). Os modelos no nível de arquitetura podem ser classificados em: (i) modelos no nível de circuito/lógico e (ii) modelos comportamentais.

No nível de circuito (também conhecido como nível de transistores) ferramentas de simulação, como SPICE [Nag75], podem ser utilizadas para obter estimativas muito acuradas, ao custo do longo tempo de simulação. Uma alternativa um pouco menos acurada, no entanto mais rápida, é a ferramenta PowerMill [HZDS95] que reporta velocidades

entre 10 e 1000 vezes maiores que o SPICE. PowerMill e SPICE necessitam de vetores de entrada, que caracterizam entradas típicas do circuito, para realizar as simulações. O tamanho destes vetores geralmente precisa ser grande, provocando longo tempo de simulação.

Em circuitos baseados na tecnologia CMOS, grande parte da potência é consumida durante a carga e descarga das capacitâncias de carga [TP98] (ver Equação 2.3). As abordagens no nível lógico (ex.: [BNYT93, TP98, GDKW92]) se baseiam neste fato para estimar consumo de energia mais rapidamente e em um nível de abstração superior em comparação às abordagens de simulação no nível de circuito.

Nas etapas iniciais de projeto é essencial avaliar diversas alternativas de projeto. As abordagens no nível de circuito e lógico não são adequadas nessas fases, pois seria necessário construir descrições de baixíssimo nível para cada alternativa em avaliação, o que é uma tarefa cara e demorada. Desta forma, as abordagens que adotam modelos comportamentais são uma interessante alternativa. Tais métodos usualmente adotam simuladores ciclo-a-ciclo e estimam consumo de energia modelando unidades funcionais do processador. Esta abordagem pode ser classificada em dois diferentes métodos: (i) métodos analíticos (ex.: [BTM00, AMG]) e (ii) métodos empíricos (ex.: [VKI+00, BBS+03]). Não obstante, alguns trabalhos combinem ambos os métodos (ex.: [PKG02, DLCD00]). Nas abordagens que adotam modelos comportamentais, os simuladores ciclo-a-ciclo são utilizados para obter os níveis de atividade das unidades funcionais. Adicionalmente, devido ao uso destes simuladores, além do consumo de energia, índices de desempenho são obtidos com boa exatidão.

Nos modelos empíricos, os dados de consumo de energia das unidades funcionais são obtidos através de implementações reais da plataforma. Normalmente, estes dados são medidos através de simulações nos níveis de circuito ou lógico. A ferramenta PowerTimer [VKI+00] é um exemplo de abordagem que adota este método. Esta ferramenta utiliza o conjunto de ferramentas SimpleScalar [BA97] como simulador ciclo-a-ciclo e armazena os dados de energia das unidades funcionais em tabelas, que são consultadas durante a simulação. Por outro lado, nos métodos analíticos, os dados de consumo de energia das unidades funcionais são obtidos analiticamente. Assim como PowerTimer, a ferramenta Wattch [BTM00] também adota o SimpleScalar como simulador, no entanto, a estimativa do consumo de energia é baseada em um conjunto de modelos analíticos de potência. Nesta ferramenta, os modelos utilizados são parametrizáveis, podendo ser aplicados em uma grande variedade plataformas. A Figura 2.3 exibe uma visão geral da estrutura desta ferramenta, onde pode ser observado que o consumo é estimado através de modelos de potência que se baseiam nos dados da configuração da plataforma alvo e nos índices de acesso às unidades funcionais produzidos pelo código em avaliação.

A vantagem dos métodos analíticos sobre os métodos empíricos é que os primeiros são mais flexíveis, podendo ser aplicados em várias plataformas sem grandes modificações. Por outro lado, modelos analíticos dependem da exatidão do modelo adotado, no entanto, é difícil construir modelos com razoável exatidão para plataformas complexas, assim, modelos empíricos geralmente são mais exatos pois os dados são gerados a partir de implementações reais da plataforma.

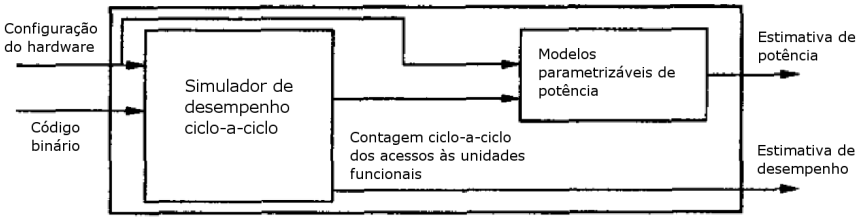

Figura 2.3 Estrutura da ferramenta Wattch [BTM00].

2.3 CONSIDERAÇÕES FINAIS

Este capítulo apresentou diversas abordagens para estimar consumo de energia e desempenho em sistemas embarcados. A Figura 2.4 compara estas abordagens em termos de velocidade e exatidão. É possível observar, por exemplo, que a abordagem FLPA é a mais rápida e os métodos no nível de circuito/lógico são os mais exatos.

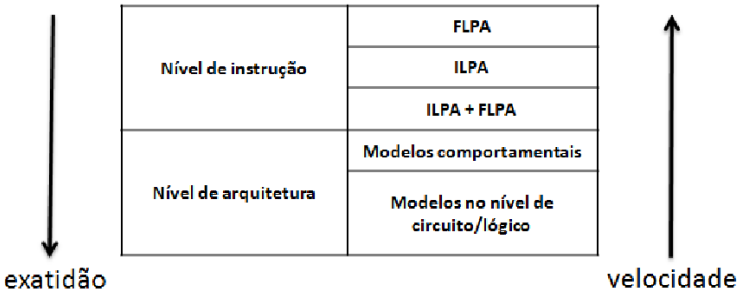

Figura 2.4 Comparação das abordagens.

Esta dissertação apresenta uma abordagem baseada na arquitetura, no entanto, diferentemente dos trabalhos apresentados na Seção 2.2, os modelos propostos possuem alto nível de abstração, permitindo flexibilidade e velocidade. Adicionalmente, os modelos são alimentados com dados da plataforma real através de medições, fazendo com que o método proposto possua boa exatidão. A abordagem baseada em medições também permite que os modelos sejam construídos sem que sejam necessárias descrições de baixo nível da arquitetura, que nem sempre estão disponíveis, como ocorre nos métodos propostos da Seção 2.2.

CAPÍTULO 3

FUNDAMENTOS

Um sistema estocástico de eventos discretos (*Stochastic Discrete Event System* - SDES) [Zim07] é um sistema que ocupa um único estado durante certo período de tempo até que um evento atômico faz com que haja uma mudança instantânea em seu estado. Eles são chamados de sistemas de eventos discretos porque seus estados não mudam entre eventos subsequentes, ao passo que mudanças de estado ocorrem continuamente em sistemas de eventos contínuos. Nos SDES, atrasos estocásticos (descritos por funções de distribuição probabilísticas) e escolhas probabilísticas [Zim07] são utilizados para modelar incertezas no sistema, que podem ser introduzidas por diversos fatores tais como ações humanas imprevisíveis e falhas em equipamentos.

Diversos modelos dentro da classe dos SDES têm sido desenvolvidos como, por exemplo, *stochastic automata*, *queuing models* e Redes de Petri estocásticas. Neste trabalho as Redes de Petri Coloridas (*Coloured Petri Nets* - CPN) e as Cadeias de Markov de Tempo Discreto (*Discrete Time Markov Chain* - DTMC) são utilizadas para avaliar consumo de energia e desempenho em sistemas embarcados. Como será descrito nos próximos capítulos, as CPN são usadas para modelar a arquitetura em avaliação e as DTMC são adotadas para realizar o mapeamento da aplicação no modelo arquitetural que por fim é avaliado através de simulação estocástica. Uma visão completa das possibilidades de modelagem com os SDES não está no escopo desta dissertação, no entanto, os conceitos fundamentais utilizado por este trabalho serão abordados. Uma descrição mais completa sobre os SDES pode ser encontrada em [CL08, BGdMT05, Zim07].

3.1 REDES DE PETRI COLORIDAS

As Redes de Petri [Mur89] são uma ferramenta matemática e gráfica que permite modelar diferentes tipos de sistemas, tais como: paralelos, concorrentes, assíncronos e não-determinísticos. O conceito das Redes de Petri foi inicialmente introduzido por Carl Adam na sua tese de doutoramento intitulada *Kommunikation mit Automaten* (Comunicação com Autômatos), em 1962 na Universidade de Damstadt, Alemanha [Pet62]. Deste então, elas têm sido amplamente utilizadas nas mais diversas áreas, como matemática, engenharia, manufatura dentre outras.

Desde o modelo inicial das Redes de Petri proposto por Carl Adam, várias extensões têm sido propostas para permitir descrições mais concisas e representar características não contempladas no modelo original. De acordo com [JK09], as Redes de Petri podem ser classificadas em: (i) Redes de Petri de baixo nível e (ii) Redes de Petri de alto nível. Dentre as redes de alto nível, que são caracterizadas por combinarem as Redes de Petri com as linguagens de programação, destaca-se o modelo proposto por Jensen [JK09], chamado de Redes de Petri Coloridas (CPN). Para um melhor entendimento das CPN,

esta seção abordará primeiro um exemplo de Rede de Petri de baixo-nível chamada de Rede *Place/Transition* (ou Rede de Petri clássica).

3.1.1 Redes Place/Transition

Uma Rede *Place/Transition* [DR98] é um grafo dirigido bipartido com dois vértices chamados de lugares e transições. Estes dois elementos são interligados através de arcos dirigidos, nos quais se a origem de um arco for um lugar, seu destino necessariamente deve ser uma transição, e vice-versa.

Graficamente, lugares são representados por círculos (Figura 3.1(a)), transições por retângulos (Figura 3.1(b)) e arcos por setas (Figura 3.1(c)). Um lugar pode estar "marcado" por zero ou mais tokens, que são representados por pontos pretos (Figura 3.1(d)). Não existe restrição no número de arcos conectando um lugar p a uma transição t, e vice-versa. Normalmente esta multiplicidade é graficamente representada por um único arco valorado, onde o valor associado (chamado de peso do arco) representa a quantidade de arcos na conexão. O número de tokens em cada lugar representa o estado do sistema e pode mudar durante a execução do modelo.

Figura 3.1 Elementos de uma Rede de Petri: (a) Lugar, (b) Transição, (b) Arco, (d) Token.

Uma transição t é dita habilitada se para cada arco saindo de um lugar p para t existe um token distinto. Uma transição habilitada pode disparar e assim remover tokens de seus lugares de entrada e adicionar tokens em seus lugares de saída.

Alguns conceitos sobre as Redes de Petri serão apresentados a partir do modelo da Figura 3.2(a), que modela o funcionamento de uma lâmpada. Neste exemplo, os lugares representam os possíveis estados da lâmpada: *aceso* e *apagado*. As transições representam as ações sobre a lâmpada: *ligar* e *desligar*. A posição do token na rede indica o estado atual da lâmpada.

No exemplo da Figura 3.2(a), o arco partindo de *aceso* para *desligar*, indica que a transição *desligar* só pode ser disparada se houver um token no lugar *aceso*. De maneira análoga, o arco partindo de *apagado* para *ligar*, indica que é necessário que haja um token no lugar *apagado* para que a transição *ligar* seja disparada. Quando a transição *apagar* é disparada, um token é adicionado no lugar *apagado* e um token é removido do lugar *aceso* (Figura 3.2(b)). Analogamente, quando a transição *ligar* é disparada, um token do lugar *desligado* é removido e um token no lugar *ligado* é adicionado.

Na modelagem de um sistema, lugares e transições podem ter inúmeras interpretações. Utilizando o conceito de condições e eventos, lugares representam condições e transições eventos, tal que, um evento pode ter várias pré-condições e pós-condições. A Tabela 3.1 exibe outras interpretações.

Formalmente, as Rede de Petri podem ser definidas conforme mostra a Definição 3.1.

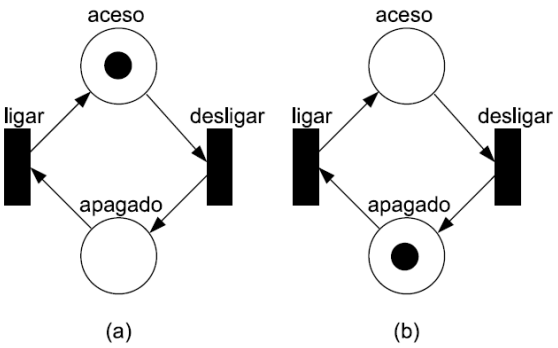

Figura 3.2 Exemplo de uma Rede de Petri

Tabela 3.1 Interpretações para os lugares e transições [Mur89]

.

Lugares de Entrada	Transições	Lugares de Saída
pré-condições	eventos	pós-condições
dados de entrada	passo e computação	dados e saída
sinal de entrada	processamento de sinal	sinal de saída
disponibilidade de recursos	tarefa	liberação de recursos
condição	cláusula lógica	conclusões
buffers	processador	buffers

Definição 3.1. (Redes de Petri) Uma Rede de Petri é uma 5-tupla, $PN = (P, T, F, W, M_0)$, em que:

- $P = \{p_1, p_2, ..., p_n\}$, é um conjunto finito de lugares;
- $T = \{t_1, t_2, ..., t_n\}$, é um conjunto finito de transições;
- $F \subseteq (P \times T) \cup (T \times P)$, é um conjunto de arcos;
- $W : F \to \mathbb{N}$, é a função de atribuição de peso aos arcos;
- $M_0 : P \to \mathbb{N}$, é a marcação inicial, em que: $P \cap T = \varnothing$ e $P \cup T \neq \varnothing$.

O comportamento dinâmico de uma Rede de Petri é dado pelas seguintes definições:

Definição 3.2. (Habilitação de Transição) Uma transição $t \in T$ é habilitada se cada lugar de entrada p da transição t contém no mínimo $w(p;t)$ tokens;

Definição 3.3. (Regras de Disparo de Transições) O disparo de uma transição habilitada t remove $w(p;t)$ tokens dos lugares de entrada p da transição t, e adiciona $w(p;t)$ tokens nos lugares de saída p da transição t.

3.1.2 Redes de Petri Coloridas Não-Hierárquicas

As CPN unem o poder das Redes de Petri ao poder das linguagens de programação. As Redes de Petri fornecem os fundamentos para a descrição da sincronização de processos concorrentes e as linguagens de programação fornecem os fundamentos para a definição de tipos de dados e a manipulação de seus valores. Em uma Rede de Petri *Place/Transition*, tokens são indistinguíveis. Em uma CPN, tokens são associados a valores, que podem ser testados e manipulados através de linguagens de programação. A CPN definida por Jensen adota a linguagem de programação funcional CPN ML, uma variação da linguagem funcional Standard ML [MTMH97].

Os modelos propostos por este trabalho foram desenvolvidos através da ferramenta CPN Tools [RWL+03], que é uma ferramenta para criação e avaliação de Redes de Petri Coloridas.

Os conceitos apresentados nesta seção serão informalmente introduzidos através de exemplos. As definições formais das CPN podem ser consultadas em [JK09].

Estrutura de uma CPN. O modelo da Figura 3.3 representa dois processos executando em paralelo. O lado esquerdo do modelo representa o processo 1, a parte do meio modela o *pool* de recursos dos processos, e o lado direito representa o processo 2. O modelo é composto por cinco transições (desenhadas como retângulos), oito lugares (desenhados como elipses), arcos conectando os lugares e transições, algumas anotações (chamadas de inscrições e escritas utilizando a linguagem CPN ML) ao lado destes elementos, e por fim, as declarações exibidas na Listagem 3.1.

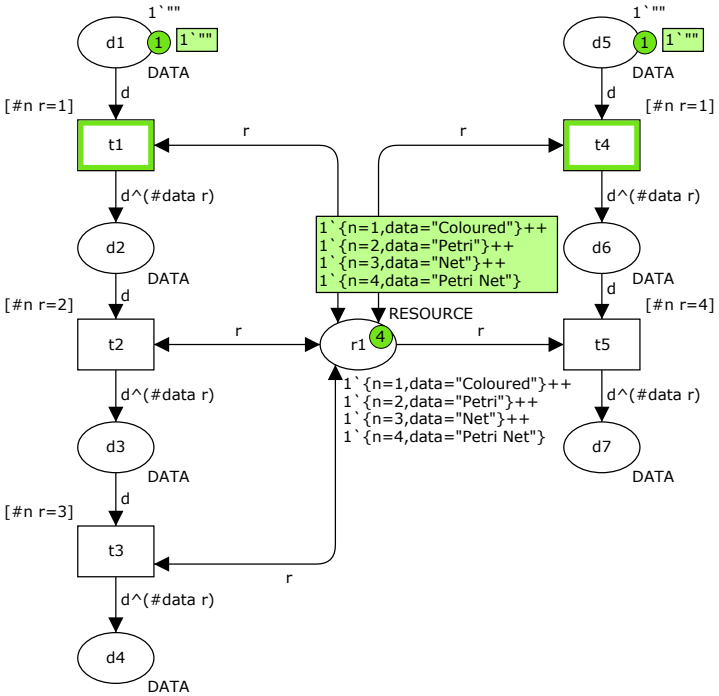

Figura 3.3 Modelo de dois processos executando em paralelo

Listagem 3.1 Declarações do modelo da Figura 3.3.

```
1  colset DATA = string;
2  colset INT = int;
3  colset RESOURCE = record n:INT * data:DATA;
4
5  var d : DATA;
6  var r : RESOURCE;
```

Cada lugar pode estar marcado com um ou mais tokens, e cada token possui um valor associado a si, este valor é chamado de **cor do token**. Por convenção, o nome dos lugares é escrito dentro das elipses. O estado do sistema, chamado de **marcação do modelo CPN**, é representado pela quantidade e cor dos tokens presentes em cada lugar, e o estado inicial do sistema é chamado **marcação inicial da CPN**. O estado do processo 1 é representado pelos lugares *d1*, *d2*, *d3* e *d4*. Similarmente, o estado do processo 2 é representado pelos lugares *d5*, *d6* e *d7*, e o estado do *pool* de recursos, pelo lugar *r1*.

A faixa de cores (valores) de tokens que um lugar pode ter é determinada pelo **conjunto de cores** do lugar, que é definido através de inscrições ao lado de cada lugar. Estes conjuntos de cores são semelhantes aos tipos das linguagens de programação. Os lugares *d1*, *d2*, *d3*, *d4*, *d5*, *d6* e *d7* possuem conjunto de cores *DATA*. Em CPN ML, um conjunto de cores é definido pela palavra chave *colset*. No modelo, *DATA* é definido da seguinte forma:

colset DATA = string;

Isto significa que os tokens nos lugares *d1*, *d2*, *d3*, *d4*, *d5*, *d6* e *d7* só podem assumir valores do tipo *string*.

O lugar *r1* possui conjunto de cores *RESOURCE*, que possui a seguinte definição:

colset INT = int;
*colset RESOURCE = record n:INT * data:DATA;*

Este conjunto de cores define um registro contendo os campos *n* e *data*, onde *n* é um inteiro e *data* uma *string*. No modelo apresentado, *n* é utilizado para identificar cada recurso e *data* modela o nome do recurso.

O número de tokens em cada lugar é exibido em um círculo (ver lugares *d1*, *d5* e *r1*). Ao lado destes círculos existe um retângulo que exibe informações sobre os valores de cada token. Em cima dos lugares *d1* e *d5* existe uma inscrição especificando a marcação inicial destes lugares, que consiste de um token com valor “ ” (i.e., uma string vazia). Similarmente, a inscrição embaixo do lugar *r1* define que a marcação inicial deste lugar possui quatro tokens. Intuitivamente, esta marcação representa os recursos disponíveis inicialmente.

As transições representam os eventos que podem ocorrer no sistema. Assim como os lugares, os nomes das transições são escritos dentro do retângulo. As transições *t1*, *t2* e *t3* representam as tarefas do processo 1, e as transições *t4* e *t5*, representam as tarefas do processo 2. Quando uma transição **dispara**, ela remove tokens dos seus lugares de entrada e adiciona tokens nos seus lugares de saída. Os valores dos tokens removidos e

adicionados dos lugares de entrada e saída são definidos pelas **expressões nos arcos**, que são inscrições posicionadas ao lado dos arcos.

As expressões nos arcos são escritas em CPN ML e podem ser construídas a partir de variáveis, constantes, funções e operadores. Considere, por exemplo, a expressão *d* no arco do lugar *d1* para a transição *t1*. Essa expressão possui a variável *d*, declarada como:

var d : DATA;

Isto significa que *d* precisa ser associada a um valor do tipo *DATA* (i.e., um valor do tipo *string*). Assim, a seguinte associação é possível:

$<d =$“ ”$>$

Associando *d* a uma string vazia.

Habilitação e disparo. As expressões nos arcos de entrada e as guardas (expressão booleana) determinam se uma transição está habilitada. Para que uma transição fique habilitada é necessário que sua guarda seja avaliada como verdadeira e haja tokens nos lugares de entrada tal que seja possível associar valores às variáveis das expressões de seus arcos de entrada. No modelo apresentado, é possível observar que as transições *t1* e *t4* estão habilitadas, ao passo que as outras transições estão desabilitadas.

A marcação da Figura 3.3 possui um token com valor “ ” no lugar *d1* e um token com valor “ ” no lugar *d5*. Isto significa que a variável *d* (variável do tipo *DATA*) presente nos arcos (*d1,t1*) e (*d5,t4*) deve ser associada ao valor “ ”. Caso contrário, as expressões nos arcos de entrada de *t1* e *t4* seriam associadas a um token cujo valor não está presente nos lugares *d1* ou *d5*, implicando que as transições *t1* e *t4* não estariam habilitadas. Por outro lado, a expressão *r* (variável do tipo *RESOURCE*) pode ser associada a qualquer um dos valores presentes *r1*, assim existem quatro possíveis associações:

$<r = \{n=1, data=$“*Coloured*”$\}>$
$<r = \{n=2, data=$“*Petri*”$\}>$
$<r = \{n=3, data=$“*Net*”$\}>$
$<r = \{n=4, data=$“*Petri Net*”$\}>$

No entanto, o leitor deve notar a expressão de guarda [$\sharp n\ r=1$] presente nas transições *t1* e *t4*. Esta expressão determina que *n* só pode assumir o valor 1, portanto, dos possíveis valores mostrados anteriormente, apenas o primeiro é valido. Assim, as possíveis associações para a marcação inicial do modelo mostrado na Figura 3.3 são:

(t1,$<d =$“ ”, $r = \{n=1, data=$“*Coloured*”$\} >$)
(t4,$<d =$“ ”, $r = \{n=1, data=$“*Coloured*”$\} >$)

Intuitivamente, isto significa que as tarefas t1 do processo 1 e t4 do processo 2 estão prontas para executar. No entanto, estas tarefas estão em conflito, uma vez que ambas as tarefas precisam de um mesmo e único recurso.

As expressões nos arcos de saída determinam os valores dos tokens adicionados nos lugares de saída. Se, por exemplo, a transição *t1* for disparada (ver Figura 3.4(a)), um token com valor igual à concatenação do valor associado à variável *d* e o valor do campo *data* da variável *r* é adicionado no lugar *d2* (o operador ^ concatena duas *strings*). Similarmente, um token com o valor associado à variável *r* é colocado no lugar *r1*. O leitor deve notar o arco bidirecional conectando a transição *t1* e *r1*, que é um atalho para a situação em que dois arcos opostos entre um lugar e uma transição compartilham a mesma expressão. O disparo da transição *t1* representa o fim da tarefa t1 e a respectiva devolução do recurso utilizado para executar esta tarefa.

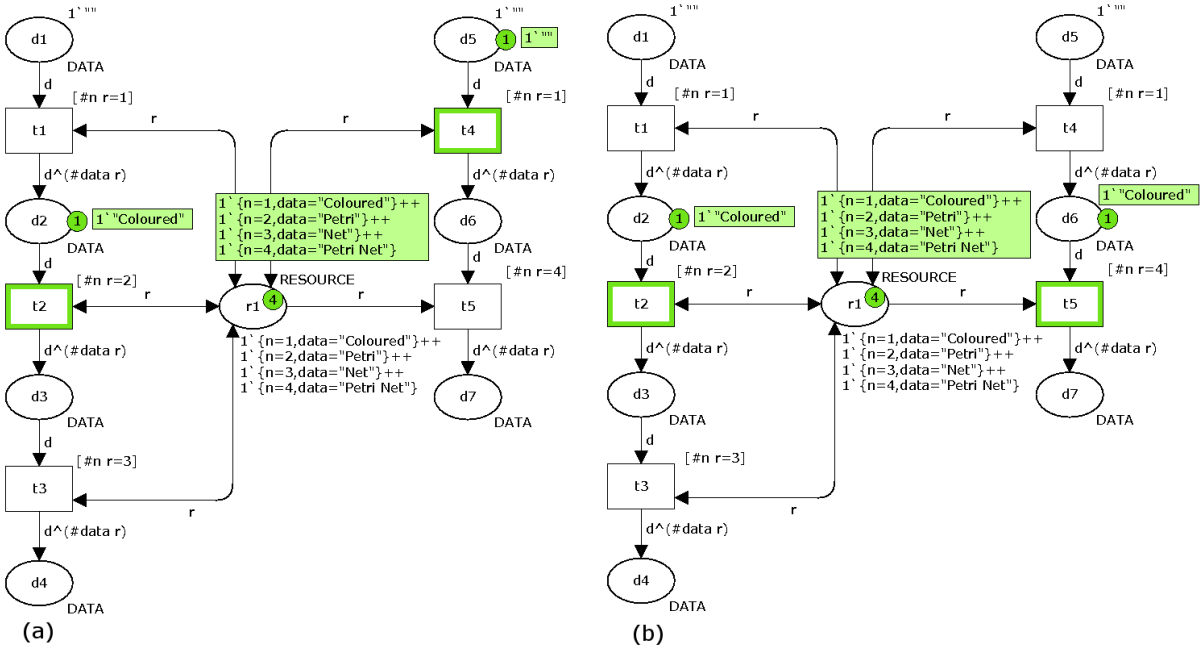

Figura 3.4 (a) Marcação após o disparo da transição *t1*. (b) Marcação após o disparo da transição *t4*.

Após o disparo de *t1*, duas associações são possíveis:

(t2,<*d* =“Coloured”, *r* = {*n=2,data=“Petri”*} >)
(t4,<*d* =“ ”, *r* = {*n=1,data=“Coloured”*} >)

Intuitivamente isto representa a situação em que as tarefas t2 e t4 estão prontas para executar. Porém, diferentemente da situação anterior em que duas tarefas estavam em conflito, t2 e t4 podem executar concorrentemente, uma vez que elas não utilizam o mesmo recurso. A Figura 3.4(b) apresenta a marcação alcançada após o disparo de *t4*.

Neste momento, as seguintes associações são possíveis:

(t5,<*d* =“Coloured”, *r* = {*n=4,data=“Petri Net”*} >)
(t2,<*d* =“Coloured ”, *r* = {*n=2,data=“Petri”*} >)

Dando sequência a execução do modelo, a Figura 3.5(a) mostra a marcação após o disparo da transição *t5*, e a Figura 3.5(b) apresenta a marcação alcançada após o subsequente disparo de *t2*.

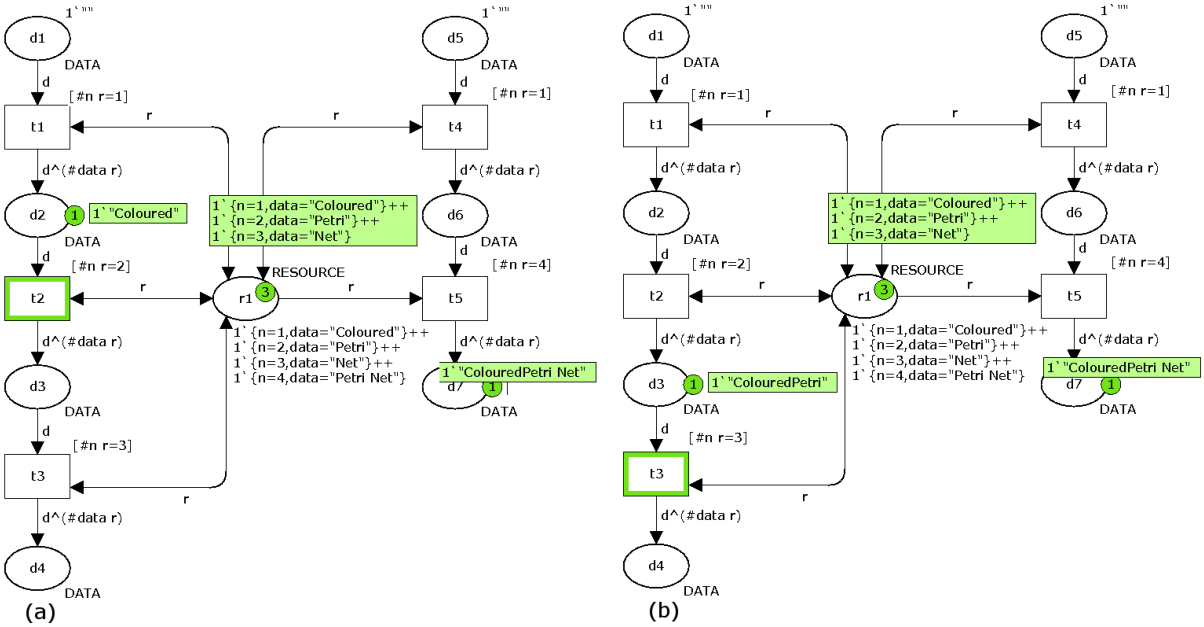

Figura 3.5 (a) Marcação após o disparo da transição *t5*. (b) Marcação após o disparo da transição *t2*.

Neste momento, apenas uma associação é possível:

$$(\text{t3}, <d = \text{“ColouredPetri”},\ r = \{n{=}3, data{=}\text{“}Net\text{”}\} >)$$

É possível observar que isto significa que o processo 2 terminou de executar. A Figura 3.6 apresenta a marcação depois do disparo da transição *t3*, representando o fim da execução das tarefas do processo 1.

3.1.3 Redes de Petri Coloridas Temporizadas e Hierárquicas

Hierarquia. As CPN permitem a modelagem de estruturas hierárquicas. A idéia básica é permitir a construção de modelos grandes a partir da utilização de modelos menores. Estes modelos menores são chamados de páginas e são conectados através de lugares chamados de portas. Tais lugares podem ser do tipo porta de entrada, porta de saída ou porta de entrada/saída. Para ilustrar o uso de hierarquias nas CPN, esta seção apresenta um exemplo de modelo hierárquico construído a partir do modelo da Figura 3.3.

A maneira mais direta de hierarquizar o modelo da Figura 3.3 é criar um módulo para o processo 1 e um módulo para o processo 2. As Figuras 3.7(a) e 3.7(b) apresentam tais módulos. O módulo do processo 1, contém 5 lugares e 3 transições. O lugar *r1* do módulo do processo 1 é uma porta de entrada/saída (*input/output port*). Isto significa

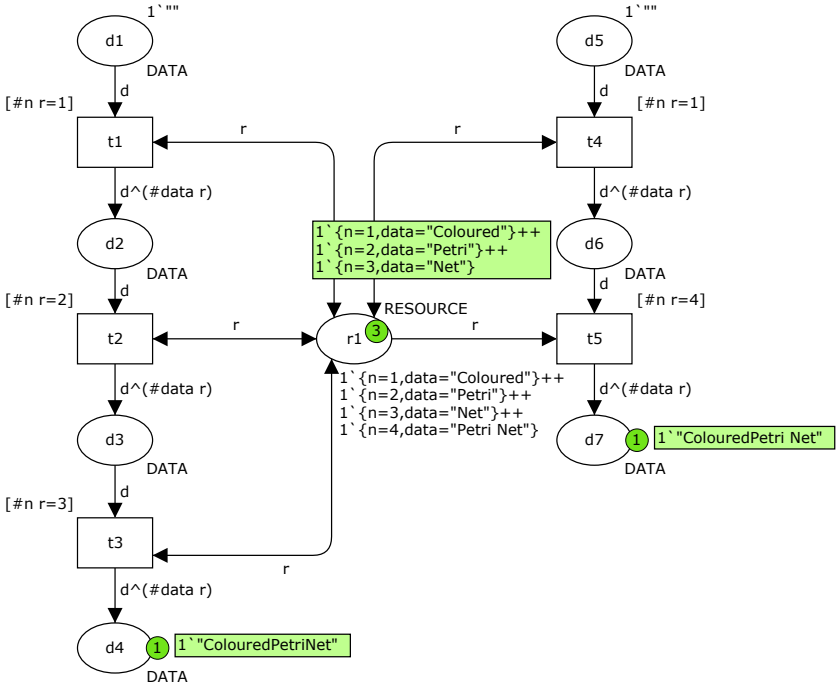

Figura 3.6 Marcação após o disparo da transição *t3*.

que o lugar *r1* é uma interface com a qual este módulo troca tokens com o seu ambiente (i.e., o módulo do processo 2). Uma porta de entrada/saída é o tipo de porta usado para importar e exportar tokens para outros módulos. Os outros lugares são lugares internos e são relevantes apenas para o módulo do processo 1. Semelhantemente, o módulo do processo 2 possui 4 lugares e 2 transições, e da mesma forma que o módulo do processo 1, o módulo do processo 2 utiliza uma porta de entrada/saída para realizar a comunicação com o ambiente. Além das portas de entrada/saída, as CPN permitem definir portas de entrada e portas de saída, que são usadas, respectivamente, para importar e exportar tokens de outros módulos.

A Figura 3.8 apresenta uma visão em alto nível dos módulos apresentados, onde é possível observar que o lugar *r1* realiza a comunicação entre os dois módulos.

CPN temporizadas. As CPN permitem a utilização de tempo nos modelos, que é especialmente importante para a obtenção de métricas de desempenho do modelo. Isto é feito através da introdução de um tempo global e permitindo que cada token seja associado a uma etiqueta de tempo (*time stamp*). A Figura 3.9 exibe a versão com tempo do modelo da Figura 3.3 e a Listagem 3.2 as respectivas declarações.

Listagem 3.2 Declarações do modelo da Figura 3.3.

```
1  colset DATA = string timed;
2  colset INT = int;
3  colset RESOURCE = record n:INT * data:DATA timed;
4
5  var d : DATA;
6  var r : RESOURCE;
```

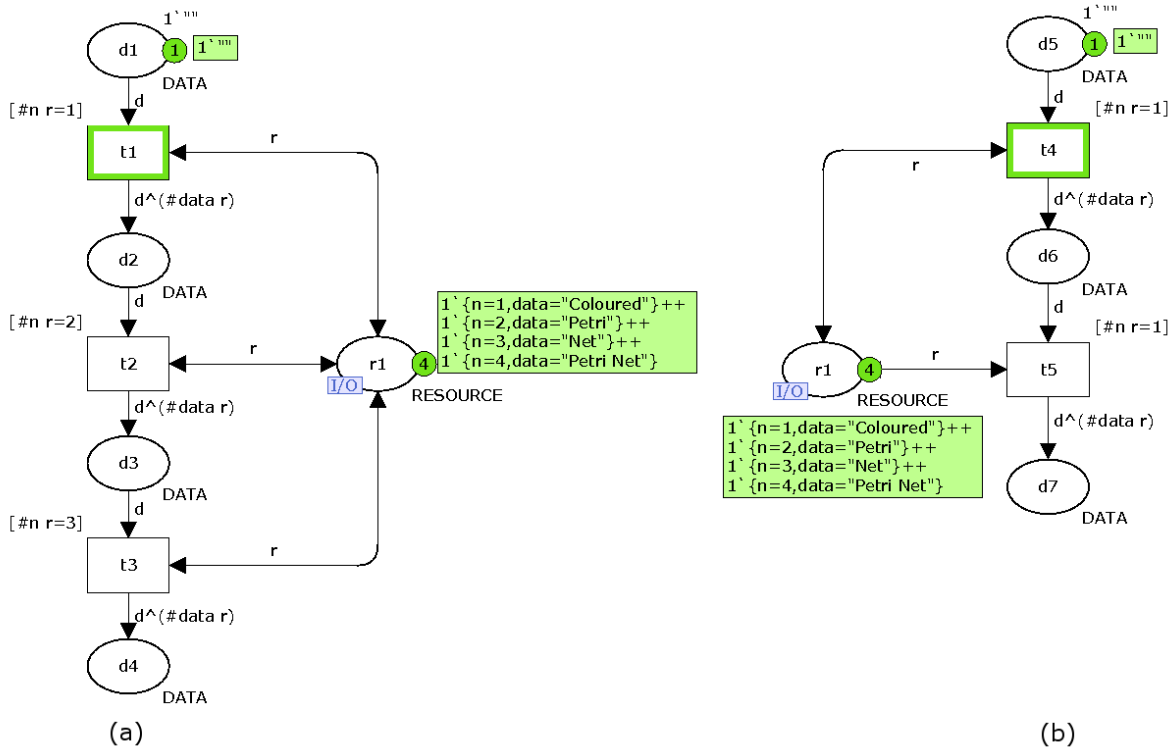

Figura 3.7 (a) Módulo representando o processo 1. (b) Módulo representando o processo 2.

Na Figura 3.9 é possível notar as etiquetas de tempo associada aos tokens do modelo, as quais indicam o momento em que cada token pode ser usado (i.e., o tempo no qual o token pode ser removido por uma transição habilitada). Um conjunto de cores é declarado como temporizado através da palavra chave *timed* (ver linhas 1 e 3 da Listagem 3.2). Adicionalmente, é possível observar também o tempo global do modelo. Uma transição em um modelo CPN temporizado só pode estar habilitada se o tempo associado aos tokens necessários para o disparo forem menores ou iguais ao tempo global.

Como mostrado anteriormente, as possíveis associações para as marcação inicial são:

$$(\text{t1}, <d = \text{“ ”},\ r = \{n{=}1, data{=}\text{“}Coloured\text{”}\} >)$$
$$(\text{t4}, <d = \text{“ ”},\ r = \{n{=}1, data{=}\text{“}Coloured\text{”}\} >)$$

Como os tokens necessários para o disparo possuem etiqueta de tempo com valor igual a 0 (que significa que estes tokens podem ser disparados no tempo 0) as transições *t1* e *t4* estão habilitadas. Se *t4*, por exemplo, for a transição disparada (ver Figura 3.10(a)), os novos valores para as etiquetas de tempo dos tokens criados nos lugares de saída são calculados da seguinte forma: tempo global + tempo de atraso. O tempo de atraso é especificado ao lado das transições. No exemplo, estes tempos representam o tempo necessário para executar cada tarefa dos processos 1 e 2. É possível observar que o tempo de atraso de *t4* é igual a uma unidade de tempo, assim os novos tokens possuem etiqueta de tempo igual a 1.

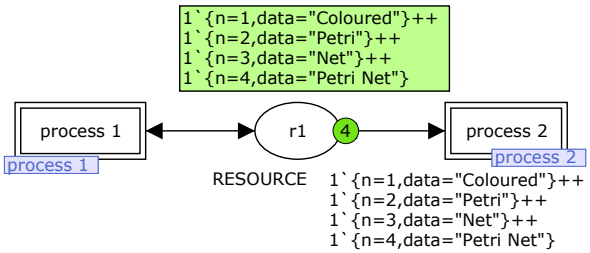

Figura 3.8 Visão em alto nível dos módulos da Figura 3.7

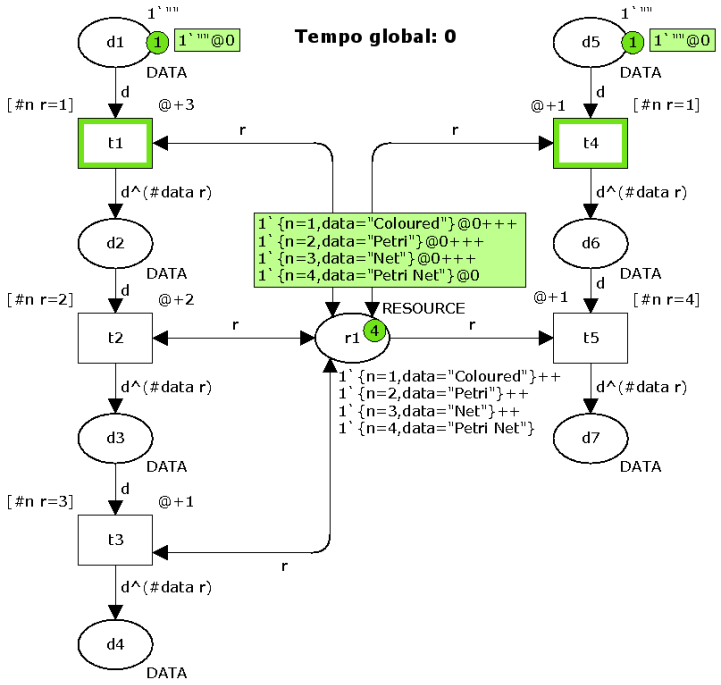

Figura 3.9 Versão com tempo do modelo da Figura 3.3

Depois do disparo de *t4*, o tempo global é avançado para o menor tempo no qual outras transições podem ser disparadas. Assim, o tempo global avança para 1, uma vez que os tokens necessários para o disparo de *t1* possuem etiquetas de tempo com valores 0 e 1 (i.e., *t1* só pode ser disparada no tempo 1), e os tokens necessários para o disparo de *t5*, possuem etiquetas de tempo com valores 1 e 1 (i.e., *t5* também só pode disparar no tempo 1). A Figura 3.10(b) exibe a marcação após o disparo de *t1*. Como a transição *t5* ainda pode ocorrer no tempo 1, o tempo global permanece com valor 1. Apesar da transição *t2* possuir os tokens necessários para o disparo, os valores das etiquetas associadas a estes tokens possuem valores iguais a 0 e 4, assim *t2* só pode ocorrer no tempo 4, que por sua vez é maior que o tempo global atual.

A Figura 3.10(c) exibe a marcação do modelo após o disparo de *t5*. O tempo global é avançado para 4, uma vez que *t2* só pode ocorrer no tempo 4. A Figura 3.10(d) exibe a marcação após o disparo de *t2*.

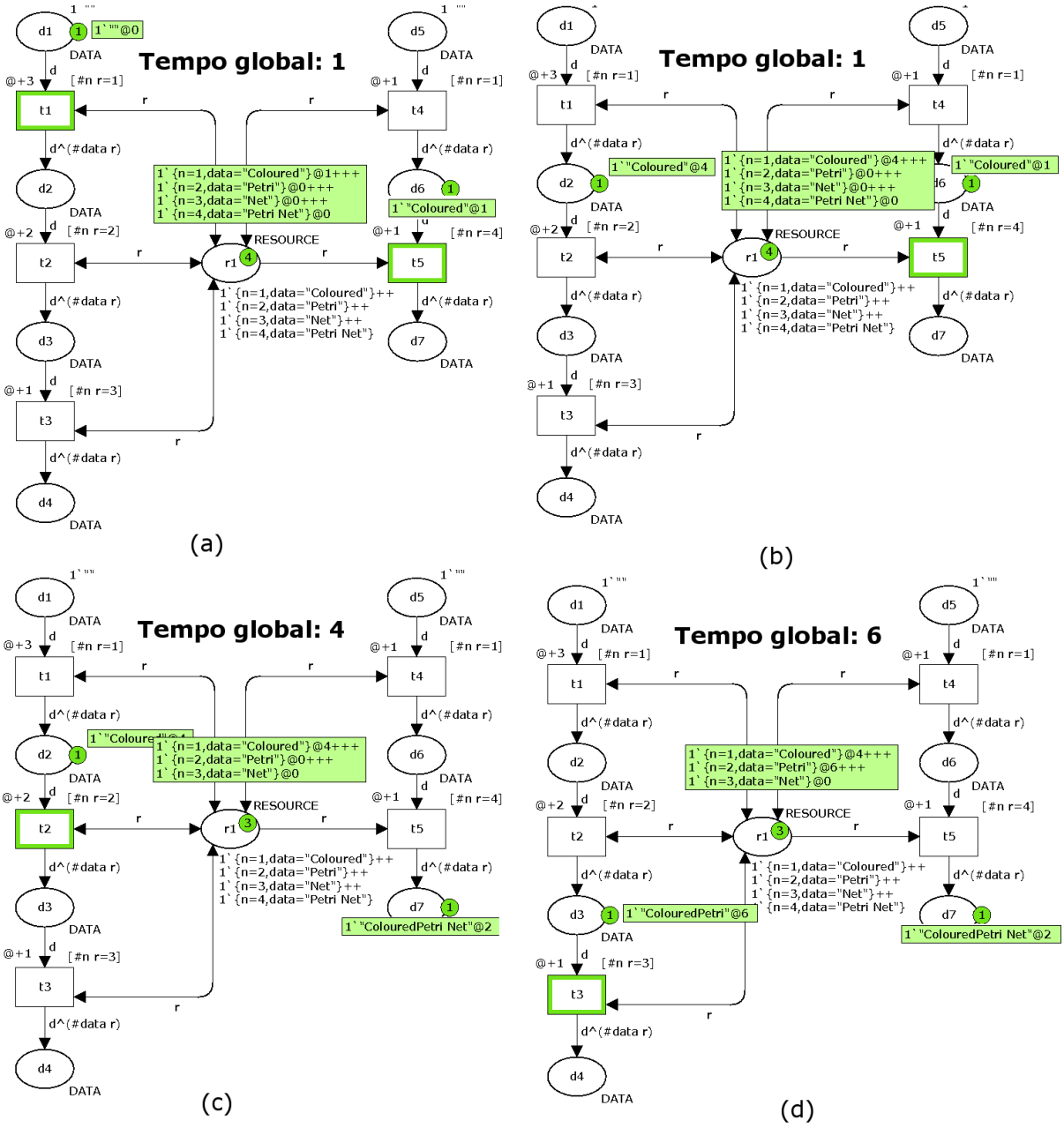

Figura 3.10 Marcações do modelo da Figura 3.9.

3.2 CADEIAS DE MARKOV DE TEMPO DISCRETO

Uma Cadeia de Markov de Tempo Discreto (DTMC) pode ser definida como uma sequência de variáveis aleatórias $X_0, X_1, X_2, ..., X_k$, na qual cada uma delas possui um número discreto[1] de possíveis valores e t é definido sobre um conjunto discreto. O valor assumido por X_t é chamado de estado da DTMC no tempo t.

Seguindo a propriedade de Markov [BGdMT05], em cada $t = 0, 1, 2, ..., k$ a probabilidade de distribuição condicional da variável aleatória X_k, dado os valores de seus predecessores $X_0, X_1, ..., X_{k-1}$, depende apenas do valor de seu predecessor imediato X_{k-1}, e não nos valores de $X_0, X_1, ..., X_{k-2}$. Em outras palavras, a probabilidade de estar em estado futuro depende apenas do estado presente, e não dos estados anteriores. Esta propriedade pode ser formalmente definida como:

Uma DTMC é dita homogênia, se $Pr(X_{k+1} = j|X_k = i)$ é independente de k. Isto é, se a probabilidade de transição de um estado para o outro é independente do tempo. Neste trabalho, apenas cadeias homogêneas são consideradas.

$$\begin{aligned} Pr(X_k = x_k|X_0 = x_0, X_1 = x_1, ..., X_{k-1} = x_{k-1}) \quad &= \\ Pr(X_k = x_k|X_{k-1} = x_{k-1}) \end{aligned} \tag{3.1}$$

Associada a uma DTMC existe a matriz chamada de matriz de transição (*one-step probability transition matrix*), denotada por P, cujo elemento $(i; j)$ é dado pela probabilidade p_{ij} de uma transição do estado $X_k = i$ para $X_{k+1} = j$ em um único passo ($p_{ij} = Pr[X_{k+1} = j|X_k = i]$):

$$P = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{pmatrix}, \; 0 \leq p_{ij} \leq 1, \; e \; \sum_{j=1}^{n} p_{ij} = 1 \; para \; cada \; i.$$

DTMCs podem ser representadas por um grafo direcionado, chamado de diagrama de estados e transições (*state-transition diagram*), no qual os vértices representam os estados da DTMC, e os arcos representam as transições entre estados. Os arcos são anotados com as respectivas probabilidades de transição em um passo. A Figura 3.11 exibe um exemplo deste diagrama.

Uma DTMC é dita irredutível quando todos os estados são possíveis de serem alcançados a partir de qualquer outro estado. A Figura 3.11, exibe um exemplo de DTMC irredutível.

O objetivo principal em estabelecer uma DTMC e a correspondente matriz de transição P é obter a probabilidade de o sistema modelado estar em um dado estado. A partir das probabilidades dos estados, diversas métricas de desempenho podem ser obtidas. Seja $\pi = (\pi_1, \pi_2, \pi_3, ..., \pi_n)$ o único vetor que satisfaz $\pi = \pi P$ e $\sum_{k=1}^{n} \pi_k = 1$. Se a DTMC possui número de estados finito e é irredutível, tal vetor único existe e é chamado de

[1] Conjunto finito ou infinito contável.

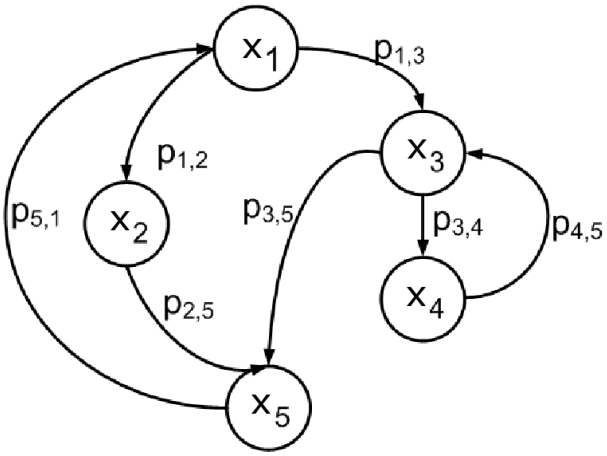

Figura 3.11 Exemplo de uma DTMC

vetor de probabilidade estacionária [Ste09]. Mais especificamente, π_i é a proporção de tempo em que o sistema fica no i no longo prazo de execução. Adicionalmente, pode ser mostrado que, no longo prazo, o número médio de visitas v_j ao estado j entre ocorrências do estado i é dado por:

$$v_j = \frac{\pi_j}{\pi_i} \tag{3.2}$$

3.3 SIMULAÇÃO DE SDES

De acordo com [Ban99], simulação é a imitação do funcionamento de um sistema (ou processo) real ao longo do tempo. Na simulação tentamos reproduzir as características ou propriedades em estudo do sistema real e gerar um histórico artificial deste funcionamento. Este histórico é posteriormente analisado, para que em seguida os resultados sejam inferidos. Assim, as simulações permitem que as inferências sobre os sistemas reais sejam feitas sem a necessidade de construí-los, perturbá-los ou destruí-los.

Na simulação de SDES, duas execuções de um mesmo modelo de simulação, com o mesmo estado inicial, muito provavelmente irão gerar saídas diferentes. Na prática, isto significa que devemos utilizar técnicas de inferência estatística para examinar os dados gerados pela simulação [BGdMT05].

Existem dois tipos de simulação em relação à análise da saída: simulações terminais e de estado estacionário.

- Simulações terminais: Neste tipo de simulação, a natureza do problema define explicitamente o final de uma execução da simulação. Por exemplo, pode-se estar interessado na simulação de um banco que fecha em um horário específico a cada dia.

- Simulações de estado estacionário: Aqui, o comportamento no longo prazo do sistema é estudado. Neste tipo de simulação, os resultados devem ser independentes

do estado inicial. Um exemplo é a simulação de um banco ao longo dos dias, para o qual o analista está interessado em alguma medida de desempenho no longo prazo de funcionamento do banco.

Existem diversas técnicas para examinar as saídas de ambos os tipos de simulação. Na simulação de estado estacionário, uma técnica muito usada é a Média de Lotes. A idéia desta técnica é dividir uma longa execução da simulação em um número de lotes contíguos. Suponha que a saída da simulação tenha sido $Y_1, Y_2, ..., Y_m$, onde m é o número de observações da métrica de interesse. Neste método, as saídas são particionadas em b lotes contíguos, disjuntos, cada um contendo n observações, tal que $m = nb$. Assim, o i-ésimo lote, $i = 1, 2, ..., b$ consiste nas variáveis aleatórias: $Y_{(i-1)\times n+1}, Y_{(i-1)\times n+2}, ..., Y_{i\times n}$.

A média do i-ésimo lote é a média amostral das n observações do lote i:

$$Z_i = \frac{1}{n}\sum_{j=1}^{n} Y_{(i-1)\times n+j} \tag{3·3}$$

A variância das médias dos lotes é definida como:

$$\hat{V}_r = \frac{1}{b-1}\sum_{i=1}^{b}(Z_i - \overline{Z}_b)^2 \tag{3·4}$$

Onde a média amostral dos lotes é dada por:

$$\overline{Z}_b = \overline{Y}_m = \frac{1}{b}\sum_{i=1}^{n} Z_i \tag{3·5}$$

Se m for suficientemente grande, então as médias dos lotes são aproximadamente normais IID (Teorema Central do Limite) e, assim, pode-se obter um intervalo de confiança para média com 100(1 - α)% de confiança dado por:

$$\{\overline{Y}_m - t_{\alpha/2,b-1}\sqrt{\frac{\hat{V}_r}{b}}; \overline{Y}_m + t_{\alpha/2,b-1}\sqrt{\frac{\hat{V}_r}{b}}\} \tag{3.6}$$

Onde $t_{\alpha/2,b-1}$ é o ponto percentual *1 - α/2* da distribuição *t-Student* com *b - 1* graus de liberdade.

A inferência estatística também pode ser utilizada para controlar o tempo de simulação, i.e., definir critérios de parada para a simulação. Seja $\overline{Y}_m$ um estimador da métrica de interesse θ (média amostral), a precisão da estimativa pode ser medida através de:

$$\varepsilon = t_{\alpha/2,b-1}\sqrt{\frac{\hat{V}_r}{b}/Y_m} \tag{3·7}$$

Onde ε é conhecida como precisão relativa. Assim, durante a simulação, a precisão relativa pode ser sequencialmente medida em *checkpoints* consecutivos e comparada a um dado nível precisão desejada, ε_{max}. A simulação para quando a precisão relativa for

menor ou igual a ε_{max} (i.e., m e b crescem até que a condição de parada seja satisfeita). Quando a precisão desejada é atingida, pode-se afirmar com 100(1 - α)% de certeza que o valor correto de θ é:

$$\overline{Y}_m \pm t_{\alpha/2,b-1}\sqrt{\frac{V_r}{b}} \tag{3.8}$$

CAPÍTULO 4

MODELO ARQUITETURAL

Este capítulo apresenta o método proposto para modelar plataformas embarcadas. A apresentação será baseada na plataforma utilizada para validação deste trabalho, um microcontrolador baseado na arquitetura ARM, chamado NXP LPC2106 [Phi04]. Para modelar esta arquitetura, uma série *building blocks* genéricos foi concebida. Utilizando uma abordagem *bottom-up* (de baixo para cima), estes *building blocks* podem ser combinados para representar comportamentos sofisticados. Assim, modelar uma arquitetura complexa se torna um processo relativamente simples. Devido ao alto nível dos modelos concebidos, apesar de um microcontrolador específico ter sido adotado para validar este trabalho, o método proposto pode ser aplicado a outras arquiteturas.

As primeiras seções deste capítulo detalham cada *building block* proposto. A apresentação adotará uma abordagem incremental partindo de um modelo básico inicial, que será sucessivamente incrementado pela introdução dos *building blocks* concebidos. Em seguida, o método de avaliação do modelo é apresentado. Por fim, o ambiente de medição, que é utilizado para caracterizar e validar a arquitetura é abordado.

O LPC2106 possui 128 kB de memória FLASH e 65kB de memória SRAM. Este microcontrolador adota um processador ARM7TDMI-S [ARM01], permitindo aos projetistas desenvolverem aplicações embarcadas com pequeno tamanho, baixo consumo de energia e alto desempenho. O ARM7TDMI-S é um processador RISC de 32 bits que consiste de uma unidade de controle, um gerador de endereços, um banco de registradores de uso geral e um *integer datapath* (caminho de dados de inteiros). Uma importante característica do LPC2106 é a presença de um mecanismo de aceleração de memória, chamado de *Memory Accelerator Module* (MAM). O MAM está presente no barramento local, e situa-se entre a memória FLASH e o processador ARM7TDMI-S. Como uma memória *cache*, este mecanismo tenta prever a próxima instrução a ser buscada da memória em tempo de prevenir bolhas no *pipeline*.

4.1 MODELO DO PIPELINE

O ARM7TDMI-S possui três estágios de *pipeline* [Fur00b], descritos a seguir:

- **Fetch:** No estágio fetch as instruções são buscadas da memória e colocadas no *pipeline*.
- **Decode:** Neste estágio as instruções são decodificadas e o sinais de controle do *datapath* são preparados para o próximo ciclo.
- **Execute:** No estágio execute a instrução é de fato executada. Leituras no banco de registradores ou na memória de dados podem ser feitas, assim como operações

na ALU (Unidade Lógica e Aritmética) ou na unidade de multiplicação. Por fim, o resultado das operações é armazenado na memória ou no banco de registradores.

O *pipeline* permite que sejam executadas até três instruções simultaneamente. Assim, apesar do processador precisar de três ciclos para executar instruções do tipo mais simples (para instruções mais complexas são necessários mais ciclos [Fur00b]), a vazão do *pipeline* é de uma instrução por ciclo. A Figura 4.1 exibe o *pipeline* executando instruções do tipo mais simples.

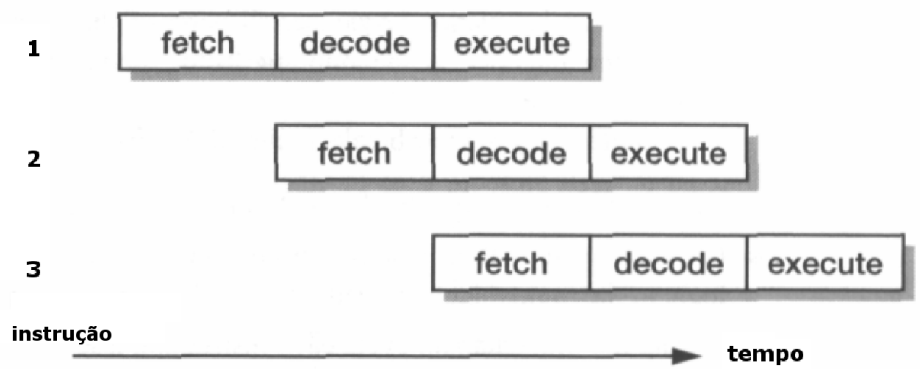

Figura 4.1 Instruções em execução no *pipeline* do ARM7TDMI-S [Fur00b].

A Figura 4.2 exibe o modelo CPN proposto para representar o *pipeline* do LPC2106 e a Listagem 4.1 mostra as declarações deste modelo. O lugar *control* representa as unidades funcionais disponíveis na arquitetura. Este lugar possui o conjunto de cores *RESOURCE* (linha 2 da Listagem 4.1), indicando que este lugar pode ter apenas os valores *fetch*, *decode* e *execute* como valores de tokens. Um token com valor *fetch* no lugar *control* indica que a unidade *fetch* está disponível, o mesmo se aplica aos outros dois possíveis valores. Inicialmente, as três unidades estão disponíveis, assim este lugar possui três tokens como marcação inicial, sendo um token com valor *fetch*, um com valor *decode* e um com valor *execute* (ver inscrição *1‘fetch++1‘decode++1‘execute* em cima do lugar *control*).

Os lugares *b1*, *b2*, *b3*, *fetching*, *decoding* e *executing* modelam os estados das instruções presentes no *pipeline*. Tais lugares possuem conjunto de cores *INSTRUCTION* (linha 1 da Listagem 4.1), indicando que nestes lugares só podem residir tokens com valores *undefined* e *dataop*. O valor *undefined* representa instruções que ainda não foram decodificadas e o valor *dataop* representa instruções que realizam operação de dados (*data operations*), o tipo mais simples de instrução.

A variável *i*, usada como expressão na maioria dos arcos do modelo, é do tipo *INSTRUCTION* (ver linha 4 da Listagem 4.1). Assim, *i* deve ser associada a um valor do tipo *INSTRUCTION*.

Listagem 4.1 Declarações do modelo da Figura 4.2.

```
1  colset INSTRUCTION = with undefined | dataop timed;
2  colset RESOURCE = with fetch | decode | execute timed;
```

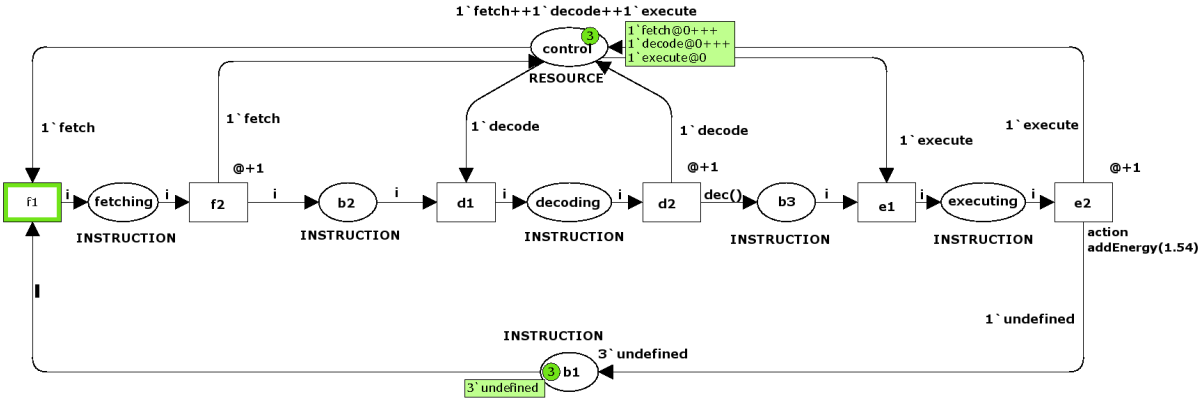

Figura 4.2 Modelo do *pipeline*.

```
3
4   var i: INSTRUCTION;
5
6   globref energy=0.0 : real;
7
8   fun addEnergy(ene : real)=
9   (
10     energy:=(!energy)+ene
11  );
12
13  fun dec()=dataop;
```

Um dado só começar a ser processado se a respectiva unidade funcional estiver disponível. Adicionalmente, quando uma unidade funcional termina sua atividade ela repassa os dados para a unidade funcional seguinte e aguarda por novas informações provenientes de estágios anteriores. Os lugares *b1*, *b2* e *b3* representam as entradas e saídas de cada unidade funcional, portanto tokens nestes lugares indicam que existem dados para serem processados e que dados terminaram de serem processados.

Como o *pipeline* modelado suporta apenas três instruções simultâneas em execução, *b1* possui marcação inicial *3‘undefined* (ver inscrição em cima do lugar *b1*). Os três tokens possuem valor *undefined* porque as instruções só são classificadas no estágio *decode* através da função *dec* presente no arco da transição *d2* para o lugar *b2*. Esta função, definida na linha 13 da Listagem 4.1, classifica instruções em apenas um tipo, *dataop*, o tipo mais simples e que requer apenas um ciclo de clock para executar. Na Seção 4.2, a função *dec* assim como o estágio *execute* serão refinados para considerar os outros tipos de instrução.

Tempo de execução neste modelo e nós próximos a serem apresentados é representado através de atrasos nas transições. Associado às transições *f2*, *d2* e *e3* existe um atraso de uma unidade, que significa um ciclo de clock (i.e., o tempo necessário para processar informações em cada unidade funcional). Consumo de energia nos modelos é representado pela função *addEnergy* (ver inscrição embaixo da transição *e2*). Esta função, definida nas linhas 8-11 da Listagem 4.1, adiciona o consumo especificado (medido em nJ) ao consumo total (armazenado na referência global *energy* - linha 6 da Listagem 4.1) sempre que uma transição possuindo esta inscrição é disparada. As informações de tempo de execução e consumo de energia foram obtidas através de documentos técnicos da plataforma [ARM01, Phi04] e medições (ver Seção 4.5).

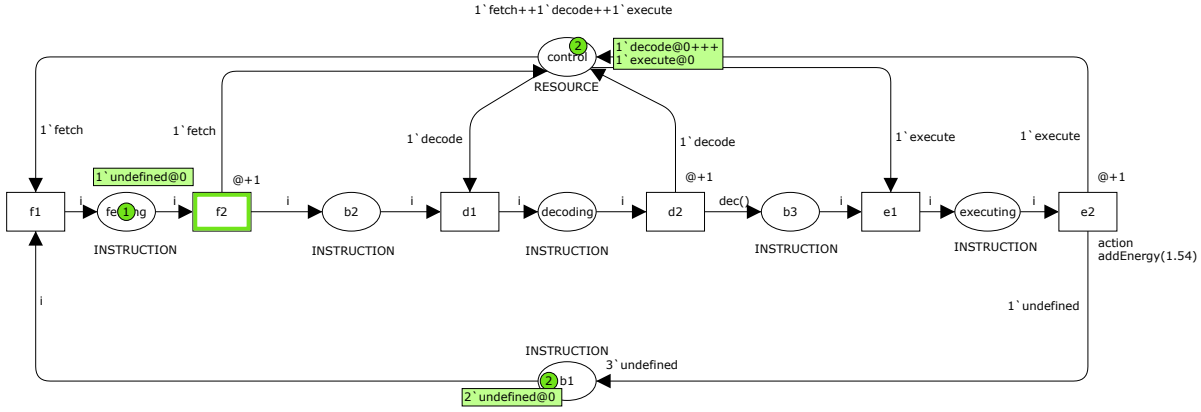

Figura 4.3 Marcação do modelo da Figura 4.2 - instrução é buscada na memória.

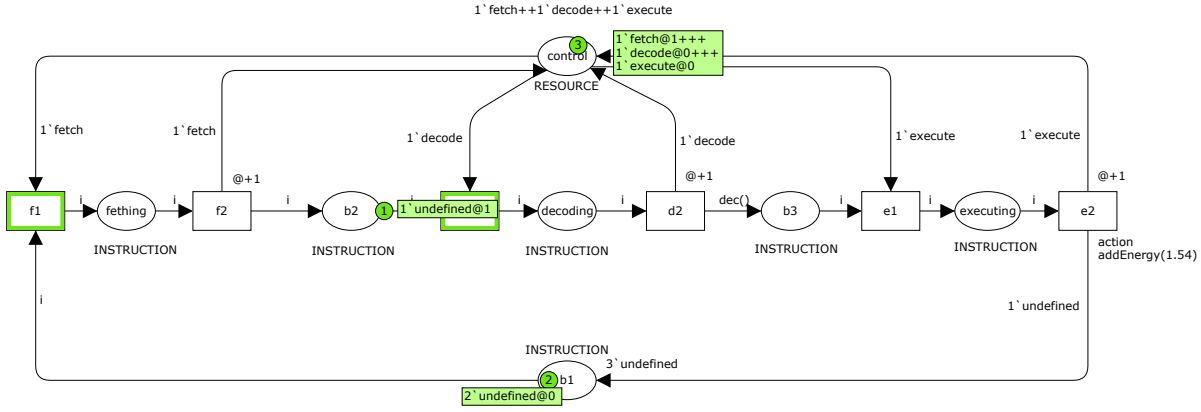

Figura 4.4 Marcação do modelo da Figura 4.2 - fim da atividade de busca na memória *fetch*.

As Figuras 4.3, 4.4, 4.5, 4.6 e 4.7 exibem algumas marcações do modelo para uma melhor compreensão de seu comportamento. Inicialmente, *f1* é a única transição habilitada no modelo. Quando esta transição dispara (ver Figura 4.3), o token com valor *fetch* é removido do lugar *control*, indicando que a primeira unidade funcional está ocupada e que uma instrução está sendo buscada da memória. Esta unidade só é liberada depois que a transição *f2* dispara (Figura 4.4), que corresponde ao fim da atividade nesta unidade. A marcação alcançada após o disparo de *f1* (Figura 4.5), seguido por *d1* (Figura 4.6), indica que no *pipeline* existe uma instrução sendo buscada da memória e outra sendo decodificada. A Figura 4.7 exibe o estado do modelo depois que *d2* é disparada, representando o fim da atividade de decodificação e envio da instrução decodificada para a unidade de execução.

A Figura 4.8 mostra os três *building blocks* desenvolvidos utilizando o modelo mostrado anteriormente para representar cada estágio do *pipeline*. Intuitivamente, é possível observar que os elementos do modelo foram separados e agrupados em módulos. Além das transições *f1* e *f2*, o *building block fetch* (ver Figura 4.9) possui o lugar interno *fetching*; a porta de entrada *b1*; a porta de saída *b2*; e a porta de entrada/saída *control*. Simi-

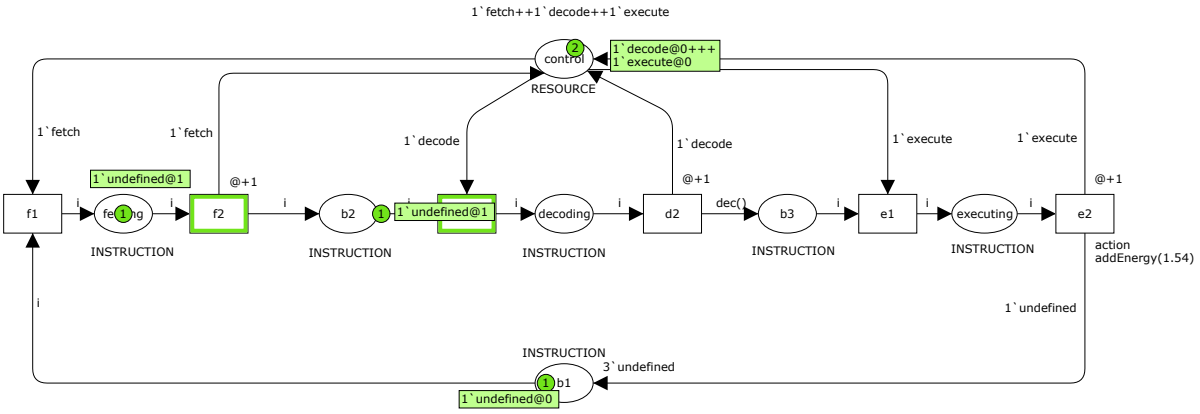

Figura 4.5 Marcação do modelo da Figura 4.2 - segunda sendo buscada na memória.

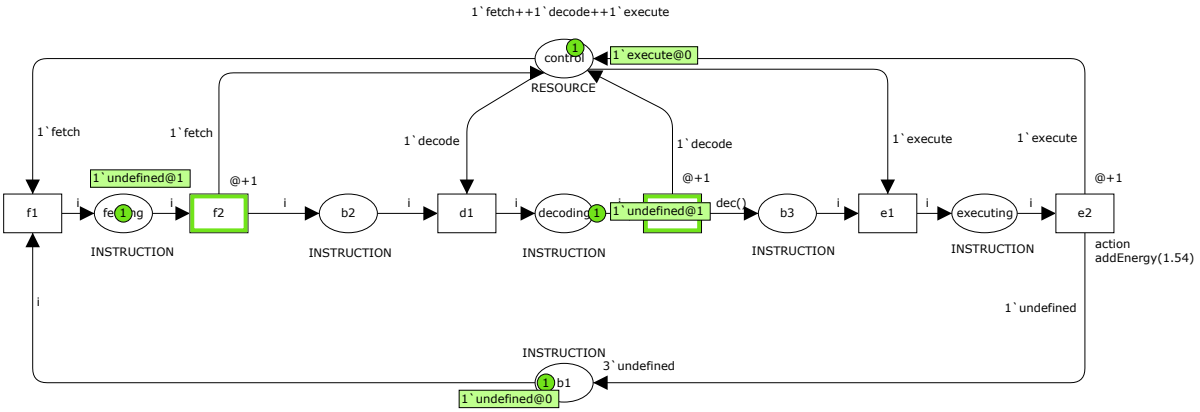

Figura 4.6 Marcação do modelo da Figura 4.2 - instrução começa a ser decodificada.

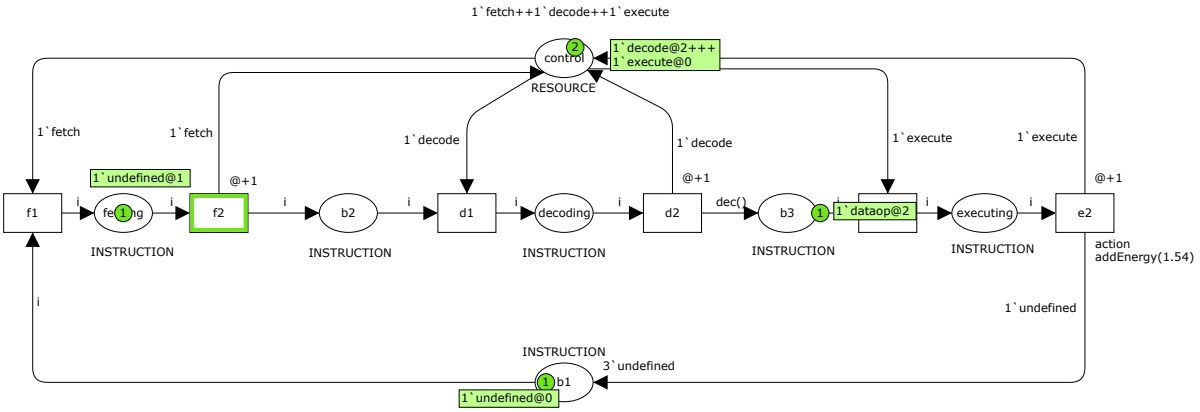

Figura 4.7 Marcação do modelo da Figura 4.2 - instrução termina de ser decodificada.

larmente, além das transições *d1* e *d2* o *building block decode* (ver Figura 4.10) possui o lugar interno *decoding*; a porta de entrada *b2*; a porta de saída *b3*; e a porta de entrada/saída *control*. Por último, o *building block execute* (ver Figura 4.11) possui o lugar interno *executing*; a porta de entrada *b3*; a porta de saída *b1*; a porta de entrada/saída *control*; e as transições *e1* e *e2*.

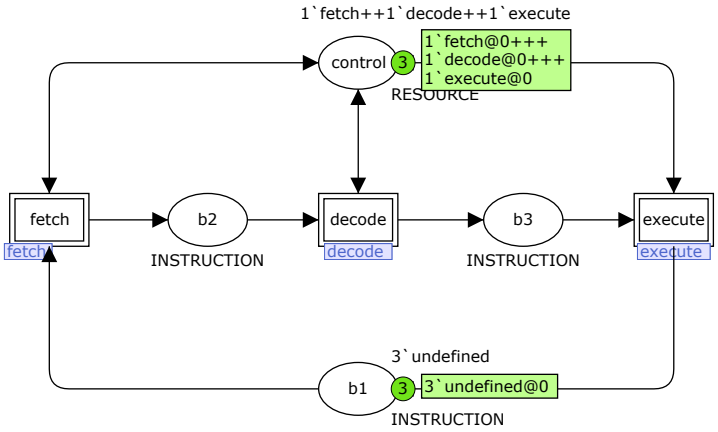

Figura 4.8 *Building blocks*: *fetch*, *decode* e *execute* (visão de alto nível).

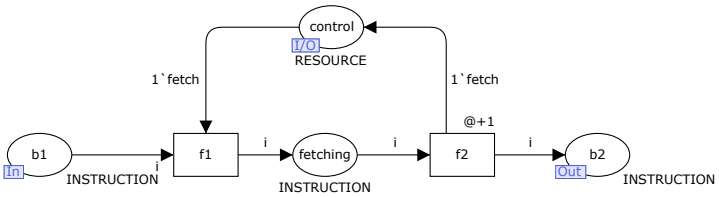

Figura 4.9 *Building block fetch*.

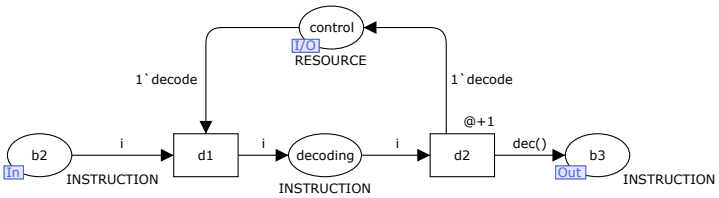

Figura 4.10 *Building block decode*.

4.2 MODELO DO ESTÁGIO DE EXECUÇÃO

Nesta seção, os *building blocks* serão refinados para representar o conjunto de instruções do LPC2106 não contemplado na seção anterior. Com base no consumo de energia e tempo de execução de cada instrução, o conjunto de instruções do microcontrolador foi dividido em quatro classes de instruções, senda elas: (i) operação de dados, (ii) escrita

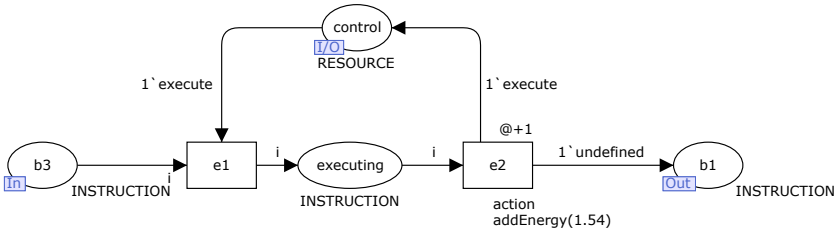

Figura 4.11 *Building block execute.*

na memória, (iii) leitura da memória, (iv) multiplicação. Dado esta classificação inicial, cada classe foi por sua vez subdividida em mais classes para representar as variações que podem ocorrer caso a instrução: (i) realize operações de deslocamento nos registradores, e (ii) escreva no registrador que armazena o contador de programa (i.e., realize um desvio no fluxo do programa). Adicionalmente, uma classe especial de instruções foi definida para representar as bolhas no *pipeline*.

A Listagem 4.2 apresenta as novas definições. O conjunto de cores *INSTRUCTION_TYPE* (linha 3), além de definir o valores que representam as instruções que ainda não foram decodificadas (*undefined*) e as bolhas no *pipeline* (*bubble*), define também os valores que representam as quatro classes de instrução: (i) operação de dados (*dataop*), (ii) escrita na memória (*wmemory*), (iii) leitura da memória (*rmemory*) e (iv) multiplicação (*mul*). A linha 1 da Listagem 4.2 exibe a nova definição para o conjunto de cores *INSTRUCTION*, que define um registro (*record*) contendo os campos *t*, *shift* e *branch*. O campo *t* é do tipo *INSTRUCTION_TYPE* e é usado para determinar a classe da instrução, o campo *shift* (operação de deslocamento) e *branch* (escrita no registrador de contador de programa) são do tipo booleano e são usados para determinar a subclasse da instrução.

No estágio de decodificação, a função *dec* (linha 5 da Listagem 4.2) retorna um valor do tipo *INSTRUCTION* de forma probabilística, tal que, se uma instrução da classe c1 é executada com frequência de 50% no programa em avaliação, esta função retornará um valor que representa a classe de c1 com probabilidade de 50%. Para classificar uma instrução, são gerados três números aleatórios com distribuição uniforme entre 0 e 1. O primeiro número, usado na função *instruction_type* (linhas 13-28 da Listagem 4.2), é utilizado para classificar a instrução em uma das quatro classes. Os outros dois números são usados para definir as subclasses (linhas 30-39 e 41-50 da Listagem 4.2). Em seguida, através de valores adicionados nas linhas 7-11, estes números aleatórios são sucessivamente comparados com a frequência que cada classe de instrução é executada. Neste trabalho, a distribuição de frequência de execução de cada classe é obtida através de uma ferramenta (ver Seção 5.2), que também automaticamente atualiza os valores das linhas 7-11.

Listagem 4.2 Novas declarações para o modelo da Figura 4.8.

```
1  colset INSTRUCTION = record t:INSTRUCTION_TYPE * shift:BOOLEAN * branch:BOOLEAN timed;
2
3  colset INSTRUCTION_TYPE = with mul | rmemory | wmemory | dataop | bubble | undefined;
4
5  fun dec()= {t=instruction_type(), shift=shifter(), branch=wpc()}
6
7  val mulFreq = 0.0;
8  val rmemoryFreq= 0.0;
```

```
 9  val wmemoryFreq = 0.0;
10  val wpcFreq = 0.0;
11  val shiftFreq = 0.0;
12
13  fun instruction_type() =
14  let
15     val choice = uniform(0.0,1.0);
16  in
17     if (choice < mulFreq)
18        then
19           mul
20        else if (mulFreq <= choice andalso choice < rmemoryFreq+mulFreq)
21              then
22                 rmemory
23               else if (rmemoryFreq+mulFreq <= choice andalso choice < rmemoryFreq+mulFreq+wmemoryFreq )
24                  then
25                     wmemory
26                  else
27                      dataop
28  end
29
30  fun shifter() =
31  let
32     val choice = uniform(0.0,1.0);
33  in
34     if (choice < shiftFreq)
35       then
36              true
37       else
38              false
39  end
40
41  fun wpc() =
42  let
43     val choice = uniform(0.0,1.0);
44  in
45     if (choice < wpcFreq)
46        then
47           true
48        else
49           false
50  end
```

Através de expressões de guardas nas transições, o novo modelo do *building block execute* define, para cada classe de instrução, os próximos estados. A Figura 4.12 exibe um trecho do *building block execute*, onde é possível observar que o consumo de energia e tempo de execução dependem do caminho tomado pelo token. Diferentemente do modelo exibido na Figura 4.11, onde apenas a transição *e2* computava o consumo de energia e tempo de execução no *building block execute*, o novo modelo define diversas transições que dependem da classe da instrução para ficarem habilitadas.

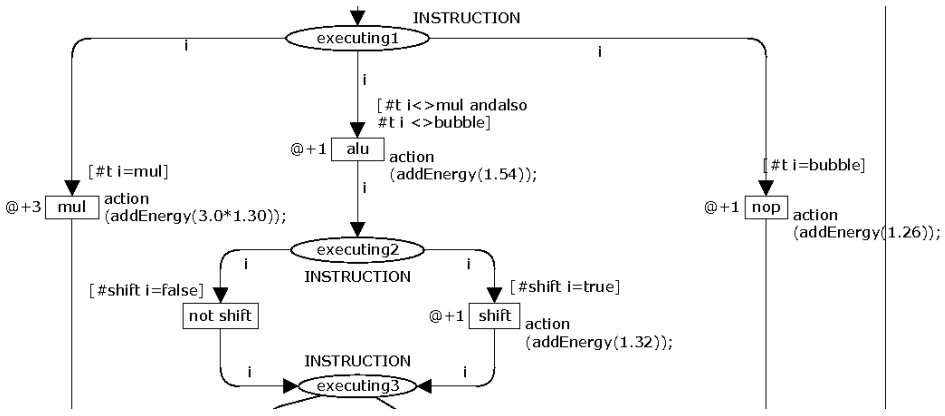

Figura 4.12 Trecho do *building block execute*.

Considere a marcação exibida na Figura 4.13, para o trecho mostrado anteriormente. Esta marcação representa uma instrução em execução que é da classe de operação de da-

dos (ver *t=dataop*), não realiza operação de deslocamento nos registradores (*shift=false*) e não realiza desvio (*branch=false*). Neste momento, caso não existissem as guardas, três transições poderiam estar habilitadas: (i) *mul*, usada para representar operações na unidade de multiplicação, (ii) *alu*, que representa operações na unidade lógica e aritmética e (iii) *nop*, que representa o caso em que a instrução não realiza nenhuma operação, como ocorre nas bolhas do *pipeline*. No entanto, as guardas restringem as possíveis transições habilitadas, e como a instrução é da classe de operação de dados, apenas a transição *alu* está habilitada. A expressão de guarda *[#t i <> mul andalso #t i <> bubble]* na transição *alu* verifica que o campo *t* do valor armazenado na variável *i*, não possui valor *mul* e também não possui valor *bubble*, portanto, não sendo uma instrução de multiplicação e não sendo uma bolha, *alu* é a transição que deve estar habilitada.

A Figura 4.14 apresenta a marcação alcançada após o disparo da transição *alu*. De forma semelhante ao caso anterior, as guardas nas transições: (i) *shift*, usada para representar operações de deslocamento nos registradores e (ii) *not shift*, que representa o caso em que a instrução não realiza deslocamento nos registradores, determinam qual transição deve estar habilitada. Para este caso, como a transição não realiza deslocamento (*shift=false*), a transição *not shift* é a transição habilitada.

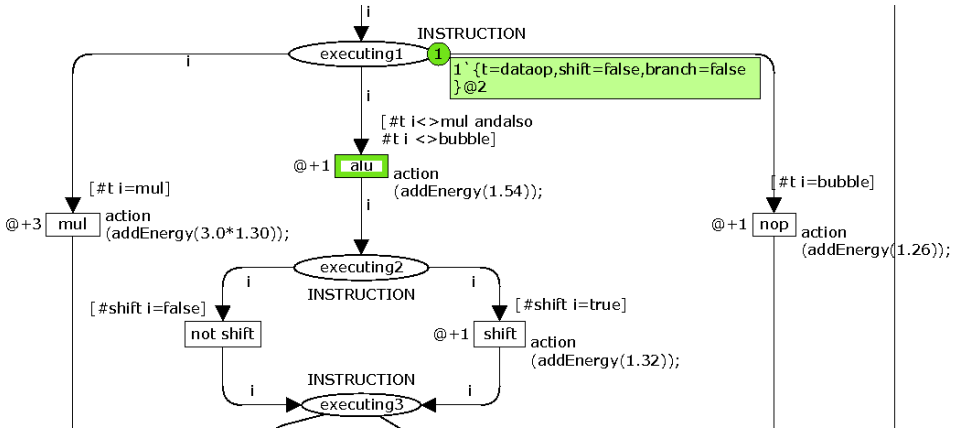

Figura 4.13 Marcação do *building block execute* - operação na unidade lógica e aritmética.

Quando uma instrução de desvio é executada (i.e., quando alguma instrução escreve no contador de programa), o *pipeline* do LPC2106 é esvaziado, gerando bolhas no *pipeline*. A Listagem 4.3 apresenta as novas declarações utilizadas para modelar este comportamento. Os valores *nop* e *undef* (linhas 3 e 4) são formas abreviadas de definir os valores que representam instruções não decodificadas e bolhas, respectivamente.

Listagem 4.3 Novas declarações para representar instruções de desvio.

```
1 colset RESOURCE = with fetch | decode | execute | flush_fetch | flush_decode timed;
2
3 val undef = {t=undefined,shift=false,branch=false};
4 val nop = {t=bubble, shift=false,branch=false};
```

Quando uma instrução deste tipo é simulada, o *building block execute* envia dois tokens com valores *flush_fetch* e *flush_decode* para a porta de entrada/saída *control*. O

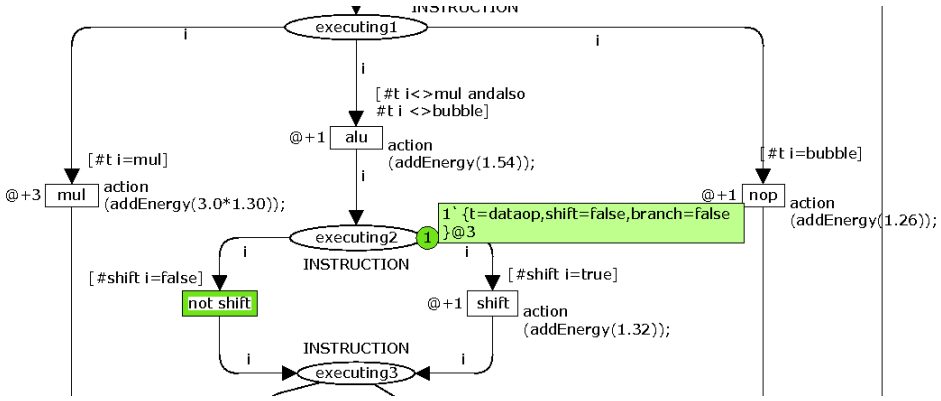

Figura 4.14 Marcação do *building block execute* - fim da operação na unidade lógica e aritmética.

leitor deve notar na linha 1 da Listagem 4.3, que o conjunto de cores da porta de entrada/saída *control*, *RESOURCE*, foi redefinido para abrigar estes dois novos valores. A porta de entrada/saída *control* é usada como interface de comunicação entre o *building block execute* e os *building blocks fetch* e *decode*. A presença de tokens com os valores *flush_fetch* nesta porta informa aos *building blocks decode* e *fetch* que suas instruções devem ser descartadas, pois um desvio foi realizado.

A Figura 4.15 exibe uma marcação para o trecho do *building block execute* que modela o comportamento de desvio. Na marcação exibida, uma instrução que é da classe de operação de dados (ver *t=dataop*), não realiza operação de deslocamento nos registradores (*shift=false*) e realiza desvio (*branch=true*) está em execução. Devido as guardas nas transições *branch* e *not branch*, neste momento, *branch* é a transição habilitada. A Figura 4.16 exibe a marcação após o disparo da transição *branch*, onde o leitor deve notar que dois tokens com valores *flush_fetch* e *flush_decode* são enviados para a porta de entrada/saída *control*.

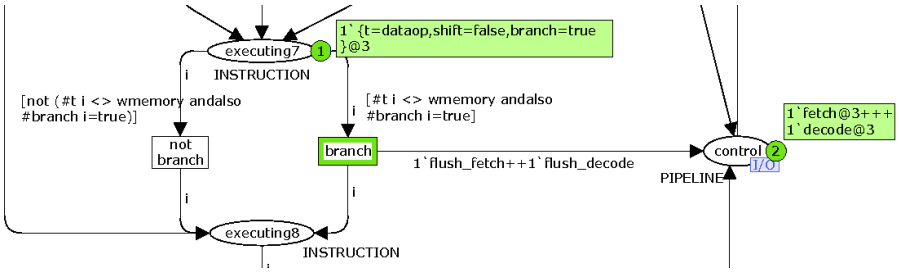

Figura 4.15 Marcação do *building block execute* - operação de desvio.

As Figuras 4.17 e 4.19 exibem marcações para os *building blocks fetch* e *decode*, que foram redefinidos para modelar o comportamento de desvio. Em comparação aos modelos das Figuras 4.9 e 4.10, em ambos os *building blocks* o leitor deve notar uma nova transição chamada *flush*, responsável por esvaziar o *pipeline*.

A marcação da Figura 4.17 exibe uma instrução que acabou de ser decodificada e espera para começar a execução. Como existe um token com valor *flush_decode* na porta de

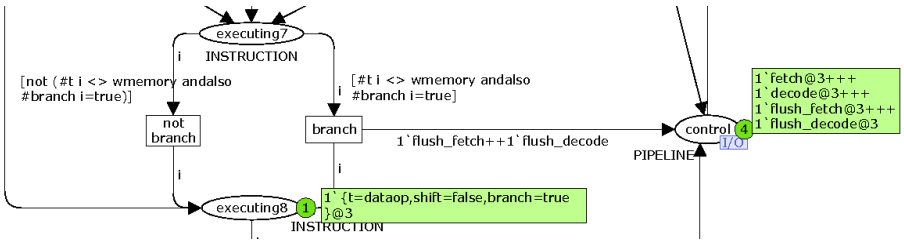

Figura 4.16 Marcação do *building block execute* - fim da operação de desvio.

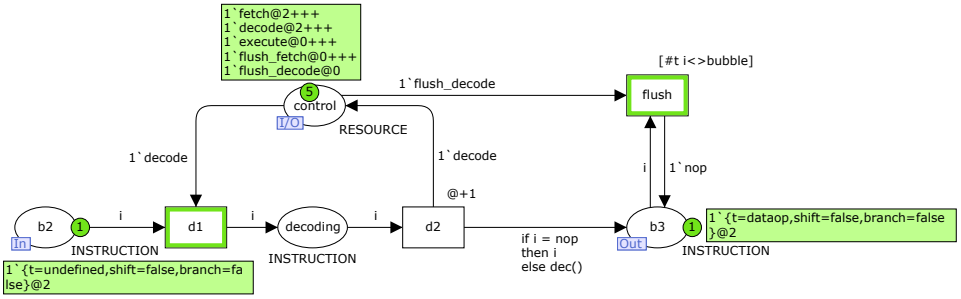

Figura 4.17 Marcação do *building block decode* - esvaziamento do pipeline.

entrada/saída *control*, a transição *flush* está habilitada. A Figura 4.18 exibe a marcação depois que a transição *flush* é disparada, onde a instrução presente no lugar de saída *b3* foi substituída por uma bolha. As bolhas seguem pelo *pipeline* como qualquer outra instrução, no entanto, no estágio de execução elas são descartadas.

De forma semelhante ao caso anterior, a marcação da Figura 4.19 exibe uma instrução que acabou de ser buscada na memória. Como um token com valor *flush_fetch* está presente na porta de entrada/saída *control*, a transição *flush* está habilitada. A Figura 4.20 exibe a marcação após o disparo da transição *flush*. No *building block fetch*, além da substituição, duas novas instruções são colocadas na porta de entrada *b1*, que intuitivamente indica que as instruções que foram substituídas serão reiniciadas.

4.3 MODELO DAS MEMÓRIAS

Nesta seção o modelo das memórias de dado e instrução são apresentados e integrados ao modelo refinado da seção anterior. Como, na arquitetura do LPC2106, as instruções são armazenadas na memória FLASH, o modelo do estágio *fetch* agora está conectado ao modelo da memória FLASH. Acessos à memória FLASH dependem do MAM. Se o dado a ser buscado da memória estiver disponível no MAM (acerto no MAM), não é necessário acessar a FLASH, caso contrário (erro no MAM), um acesso é necessário. A Figura 4.21(a) apresenta a conexão entre os *building blocks fetch* e *flash*, e a Figura 4.21(b) apresenta o *building block flash*. A Listagem 4.5 apresenta as declarações do *building block flash*.

A interface entre o *building block flash* e o *building block fetch* é feita através da portas de saída *flash* e *c2*, e das portas de entrada *end flash* e *c1*. As portas usadas como

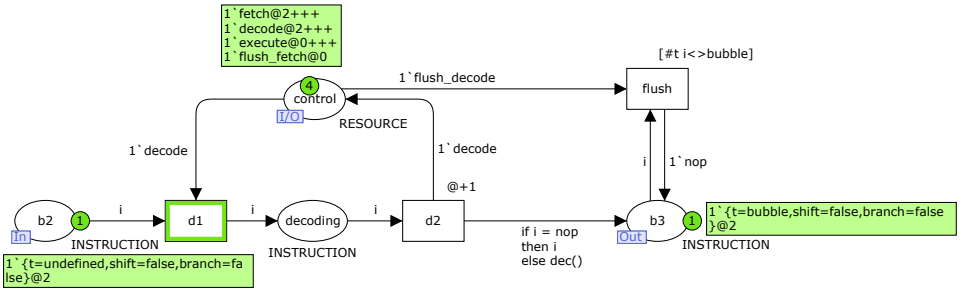

Figura 4.18 Marcação do *building block decode* - fim do esvaziamento do pipeline.

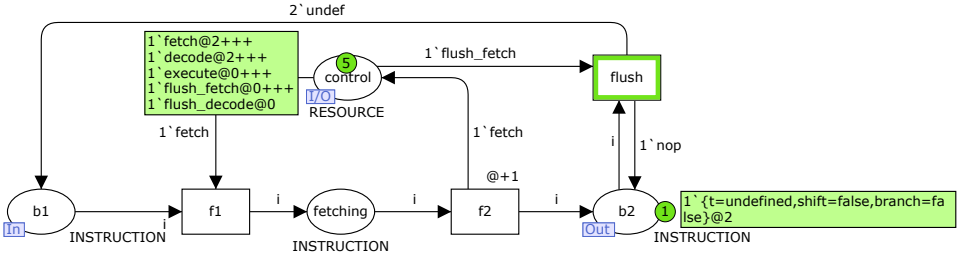

Figura 4.19 Marcação do *building block fetch* - esvaziamento do pipeline.

interface possuem o conjunto de cores *CACHE*, que é definido como tipo booleano (linha 1 da Listagem 4.5). Um token com valor *true* (verdadeiro) nestas portas, representa um acerto no MAM, e um token com valor *false*, indica erro no MAM.

Listagem 4.4 Declarações do *building block flash*.

```
1  colset CACHE = bool timed;
2  var c : CACHE;
3
4  val hitRateFreq = 0.0;
5
6  fun mamaccess() =
7  let
8     val choice = uniform(0.0,1.0);
9  in
10    (choice <= hitRateFreq)
11 end
```

A função *mamaccess* (linhas 6-12 da Listagem 4.5) retorna um valor booleano. Dado a taxa de acerto (i.e., a razão entre o número de vezes em que o dado é encontrado no MAM pelo o número de acessos à memória), esta função gera um número aleatório entre 0 e 1, e em seguida, compara com esta taxa. Caso o número aleatório seja menor que a taxa de acerto, a função retorna verdadeiro (*true*), ou falso (*false*), caso contrário. A taxa de acerto é armazenada no valor *hitRateFreq* (linha 4). As guardas das transições *cache miss* e *cache hit* no *building block flash* determinam que, caso o valor retornado pela função mamaccess seja *true*, a transição *cache hit* fica habilitada, caso seja *false*, *cache miss* será a transição habilitada.

As Figuras 4.22, 4.23, 4.24 e 4.25 apresentam marcações simultâneas dos *building blocks fetch* e *flash memory* para uma melhor compreensão da interação entre estes *building blocks*. A Figura 4.22 representa a situação em que uma instrução está prestes a ser buscada da memória. Depois que *f1* dispara (Figura 4.22) e a instrução começa a ser

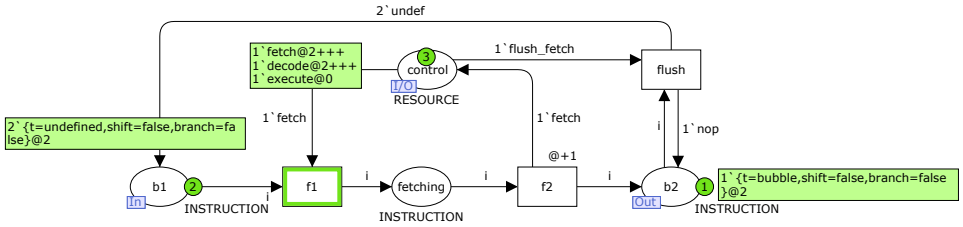

Figura 4.20 Marcação do *building block fetch* - fim do esvaziamento do pipeline.

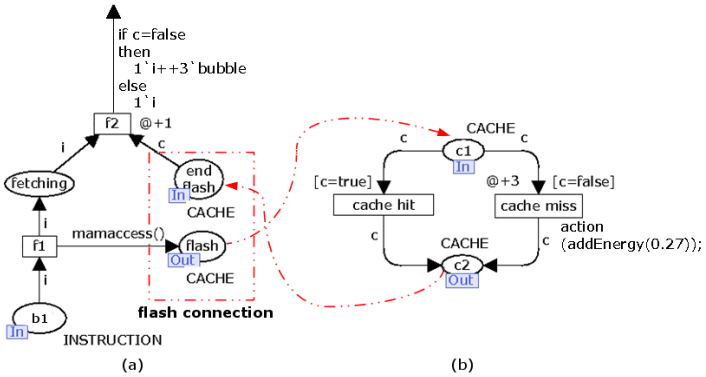

Figura 4.21 (a) Trecho do *building block fetch*. (b) *Building block flash memory*.

buscada da memória, um token com valor *false*, gerado pela função *mamaccess*, é colocado na porta de saída *flash*. Isto significa que a instrução não foi achada no MAM e por isso um acesso à memória FLASH deve ser feito. A Figura 4.24 apresenta a marcação dos dois modelos depois que a transição *cache miss* é disparada. Acessos à memória FLASH param (*stall*) o *pipeline*, gerando bolhas (ver inscrição no arco de saída da transição *f2*). Depois que *f2* dispara (Figura 4.25) e, por conseguinte, a operação de busca na memória termina, quatro tokens são colocados no lugar *b2*. O primeiro token representa a instrução buscada na memória e os outros três representam as bolhas geradas pelo erro no MAM.

A taxa de acerto no MAM depende da aplicação em avaliação. Com o intuito de validar o modelo, simulador de *traces* foi desenvolvido para obter esta informação. O simulador recebe como entrada os endereços das instruções buscadas na memória e retorna a taxa estimada de acerto no MAM.

Dependendo da classe da instrução, a memória RAM pode ser acessada durante o estágio *execute* através de uma operação de leitura ou escrita. A Figura 4.26(a) apresenta a conexão entre o *building block execute* e o *building block ram*, e a Figura 4.26(b) apresenta o *building block ram*. A Listagem 4.5 apresenta as declarações do *building block flash*.

Listagem 4.5 Declarações do *building block flash*.

```
1  colset MEMORY = with read | write timed;
2  var m: MEMORY;
3
4  fun memory(i:INSTRUCTION) =
5  if (#t i=rmemory)
6     then
7          read
```

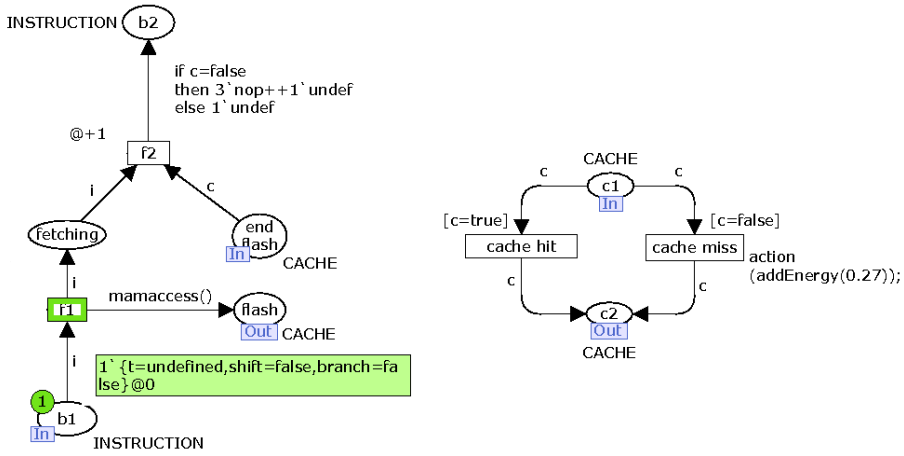

Figura 4.22 Marcação dos *building blocks fetch* e *flash memory* - instrução prestes a ser buscada.

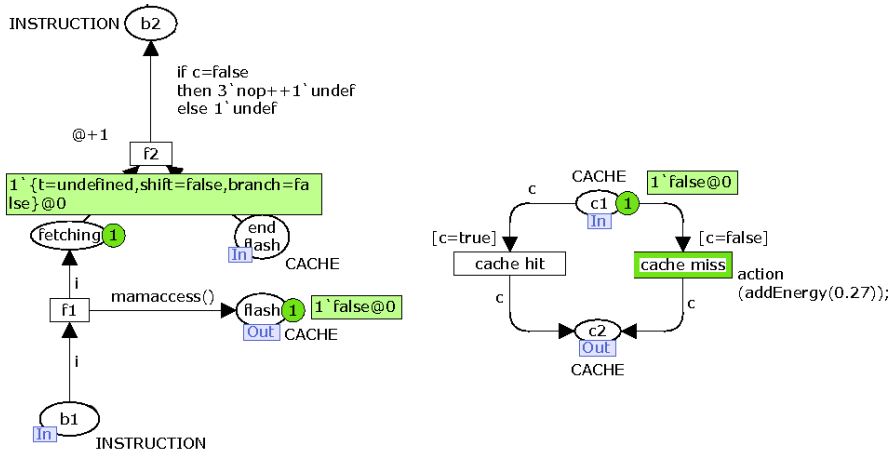

Figura 4.23 Marcação dos *building blocks fetch* e *flash memory* - instrução sendo buscada da memória.

```
8       else
9           write
```

No *building block ram*, as transições *read mem* e *write mem* representam, respectivamente, operações de leitura e escrita na memória RAM. A interface entre o *building block execute* e o *building block ram* é feita através das portas de entrada *end ram* e *m1*, e das portas de saída *rw ram* e *m2*. Estas portas possuem conjunto de cores *MEMORY* (linha 1 da Listagem 4.5), definindo que os tokens nestas portas só podem ter dois valores: *read* ou *write*. Um token com valor *read*, significa uma leitura na RAM, e um token com valor *write*, uma escrita. A função *memory* (linhas 4-9 da Listagem 4.5) recebe como entrada o valor que representa a instrução que acessará a RAM e retorna como saída dois tipos de valores, *read* ou *write*. As guardas das transições *read memory* e *write memory*, determinam que caso o valor retornado seja *read*, a transição *read memory* fica habilitada, caso seja *write*, *write memory* será a transição habilitada.

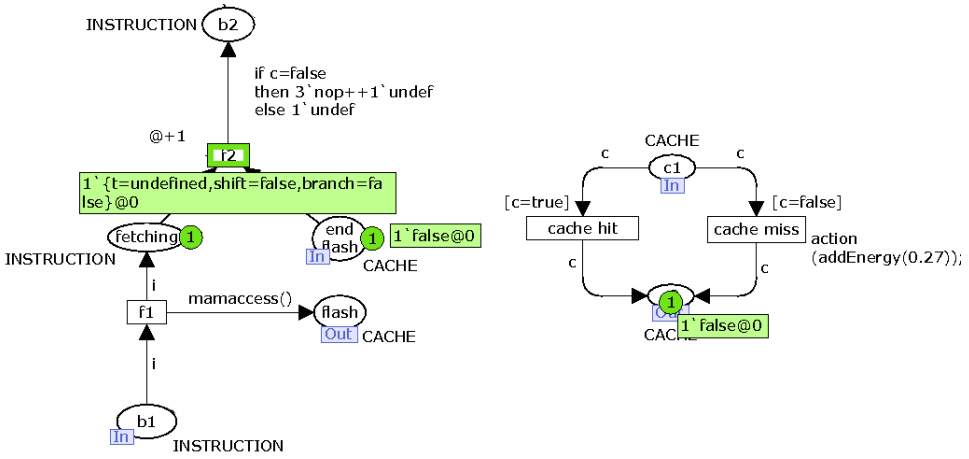

Figura 4.24 Marcação dos *building blocks fetch* e *flash memory* - fim do acesso a memória FLASH.

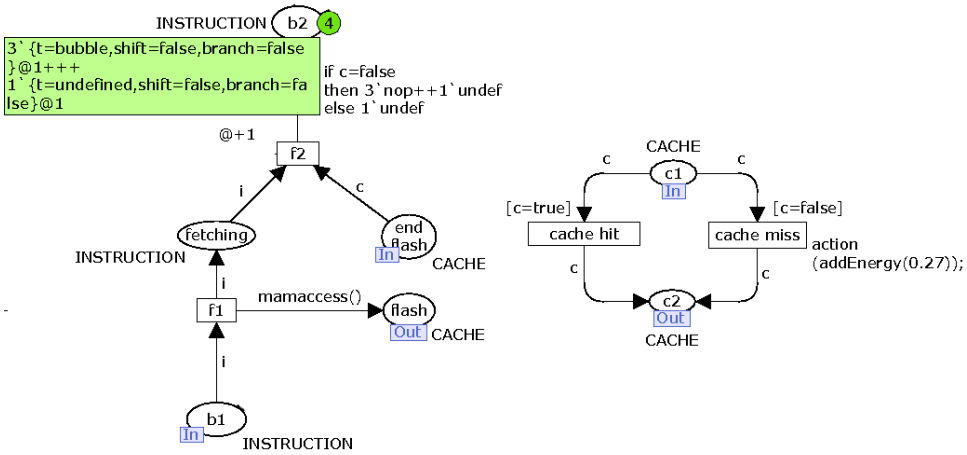

Figura 4.25 Marcação dos *building blocks fetch* e *flash memory* - fim da operação de busca de instrução.

Marcações simultâneas dos *building block execute* e *ram* são exibidos nas Figuras 4.27 e 4.28 para uma melhor compreensão da interação entre estes dois *building blocks*. A Figura 4.27 exibe a situação em que uma instrução da classe que realiza leitura na memória em execução. A Figura 4.28 exibe a marcação após o disparo de *read memory*, representando o fim da leitura na memória.

A Figura 4.29 apresenta uma visão em alto nível dos *building blocks* concebidos para modelar o LPC2106: *fetch*, *decode*, *execute*, *ram* e *flash*.

4.4 AVALIAÇÃO

A avaliação dos modelos na abordagem proposta é feita através de simulação. A ferramenta CPN Tools foi utilizada para definir funções de análise estatística e coletar dados da simulação. O objetivo da simulação é capturar as seguintes métricas: (i) tempo médio de

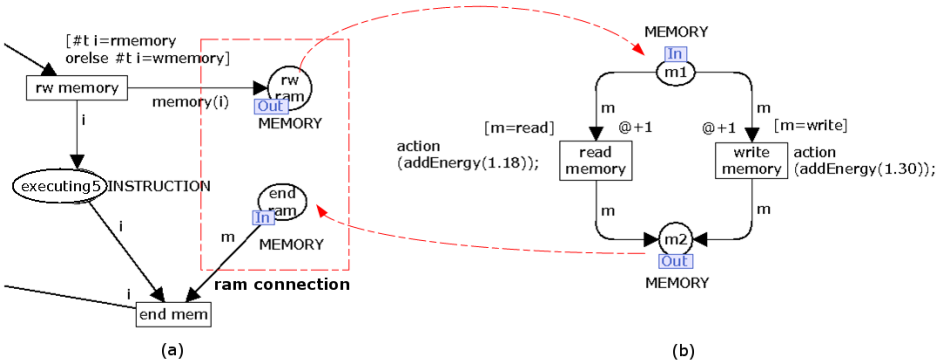

Figura 4.26 (a) Trecho do *building block execute*. (b) *Building block ram memory*.

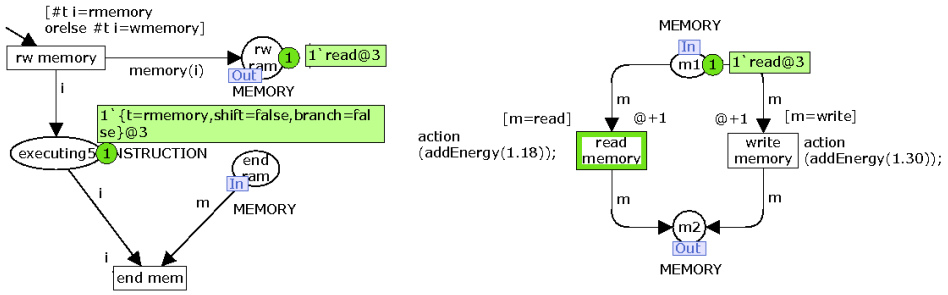

Figura 4.27 Marcação dos *building blocks execute* e *ram* - início de uma leitura na memória.

execução por instrução, e (ii) consumo médio por instrução. Dado estas duas métricas, a frequência de operação do processador e o número de instruções executadas na aplicação, é possível estimar o tempo de execução e consumo de energia de um programa.

Primeiramente, um *breakpoint monitor* [Wel02] foi definido e associado a última transição do *execute building block*. Quando uma instrução termina, esta transição é sempre disparada por todas as classes de instruções, exceto a classe que representa as bolhas no *pipeline*. Quando ela é disparada, dois dados são coletados pelo *breakpoint monitor*: (i) o intervalo do tempo de disparo, i.e., o tempo de simulação atual menos o tempo do último disparo, e (ii) o intervalo de consumo de energia, i.e., o consumo de energia atual menos o consumo de energia do último disparo.

Funções de análise estatísticas foram definidas para que fosse possível construir intervalos de confiança para os dados coletados. Estas funções constroem o intervalo de confiança utilizando o método das médias de lotes explicado na Seção 3.3.

Além de coletar dados, o *breakpoint* verifica o critério de parada da simulação, que testa se o intervalo de confiança para as métricas satisfaz a precisão desejada. Assim, a avaliação adota a abordagem de simulação sequencial, descrita na Seção 3.3, na qual a precisão relativa é sequencialmente medida e comparada a precisão desejada, sempre que novas amostras são coletadas. Este trabalho adotou um nível de confiança de 95% e precisão relativa de 2%. O Apêndice A descreve a função desenvolvida para testar o critério de parada da simulação.

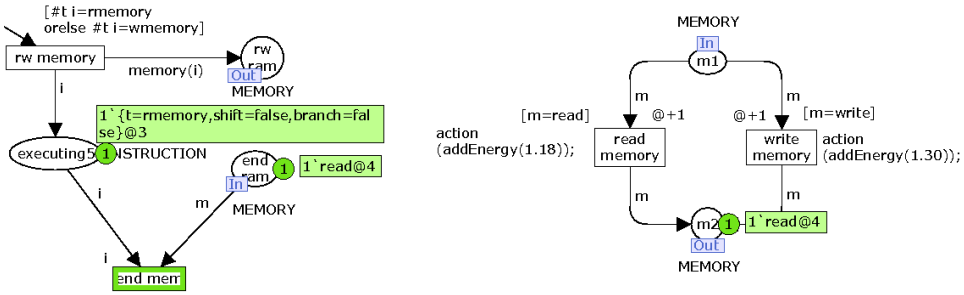

Figura 4.28 Marcação dos *building blocks execute* e *ram* - fim de uma leitura na memória.

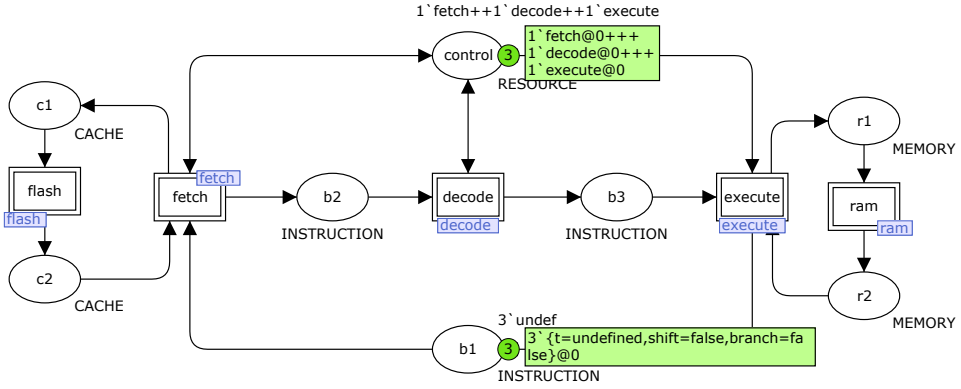

Figura 4.29 Visão em alto nível dos *building blocks* propostos.

4.5 AMBIENTE DE MEDIÇÃO

A Figura 4.30 exibe o ambiente de medição utilizado para caracterizar a plataforma e validar os resultados. A fim de medir potência média, um PC executando a ferramenta AMALGHMA (*Advanced Measurement Algorithms for Hardware Architectures*) [ama, TMS08] é conectado a um osciloscópio Agilent DSO03202A, que por sua vez captura (Canal 2) a corrente consumida através da medição da queda de tensão em um resistor de 1 Ohm (a impedância média do microcontrolador é ordens de magnitude maior que isto). Adicionalmente, o osciloscópio também é conectado a uma porta de I/O do LPC2106 (Canal 1), que é usada para monitorar os tempos de início e fim do código a ser medido. Através dessa conexão, a ferramenta AMALGHMA também consegue estimar o tempo de execução e, por conseguinte, o consumo de energia.

AMALGHMA adota uma série de técnicas estatísticas e de amostragem para obter dados com confiança. Além disso, é importante ressaltar que esta ferramenta foi validada considerando o *datasheet* do LPC2106 [Phi04] e o documento técnico de referência do ARM7TDMI-S [ARM01].

Códigos em assembly foram desenvolvidos para obter os valores de energia utilizados no modelo. Tais códigos estimulam, separadamente, a respectiva unidade funcional do LPC2106. Por exemplo, para capturar a potência média quando ocorrem erros no MAM (i.e., quando o dado buscado não está no MAM), um código em assembly que força *misses*

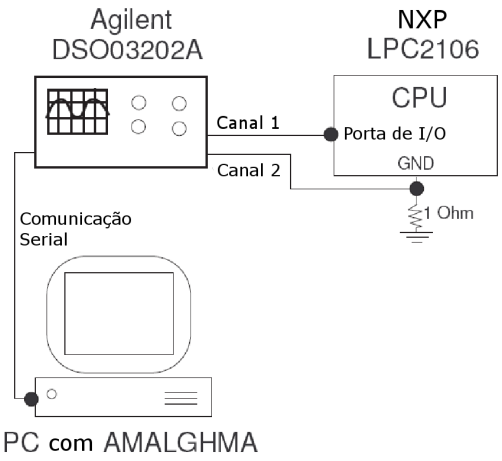

Figura 4.30 Ambiente de medição.

no MAM foi desenvolvido. Em seguida estes códigos são baixados no microcontrolador, executados e medidos.

```
1  while(1) {
2    int i;
3    IOSET = IOPIN | 0x00000080 /* levanta pino */
4
5    /* início do código */
6    __asm {
7          /* código de medição em assembly */
           (...)
8    }
9    /* fim do código */
10
11   IOCLR = (~IOPIN) | 0x00000080;
12   for(i=0; i < 5300; i++); /* guarda */
13 }
```

Figura 4.31 Estrutura de um código de caracterização.

A Figura 4.31 apresenta a estrutura de um código de caracterização. O código apresentado consiste de um *loop* infinito, onde as linhas 3 e 11 escrevem em uma porta de I/O do microcontrolador e são usadas para indicar a janela de medição no osciloscópio. O código assembly a ser medido é inserido a partir da linha 7. A escrita na porta de I/O é feita antes de uma iteração do código e logo após o seu fim, assim a saída no osciloscópio tem o formato uma onda quadrada. A Figura 4.32 exibe uma saída típica, onde é possível observar a janela de medição capturada pelo Canal 1, que indica o trecho a ser medido.

Dada a janela de medição, o tempo de execução do código (T) é medido através da equação:

$$T = (X_f - X_i) \times UT \tag{4.1}$$

Onde X_i e X_f são os pontos de início e fim do nível alto da onda quadrada, e UT é a unidade de tempo adotada no osciloscópio.

Figura 4.32 Tela do Agilent DS0302A durante a medição de um código.

Como a tensão de alimentação do microcontrolador (V_{cc}) e a resistência do resistor (R) são conhecidas. A corrente consumida (I), a potência (P) e o consumo de energia (E) são calculados da seguinte forma:

$$I = V_r/R \tag{4.2}$$

$$P = I \times V_{cc} \tag{4.3}$$

$$E = T \times P \tag{4.4}$$

4.6 CONSIDERAÇÕES FINAIS

Este capítulo introduziu o método proposto para modelar plataformas embarcadas, no qual os *building blocks* construídos para modelar o microcontrolador LPC2106 foram apresentados. No Capítulo 6, um estudo de caso considerando uma arquitetura multi-processada será apresentado.

Adotando uma abordagem probabilística, os modelos propostos são livres de detalhes de baixo nível da arquitetura: eles definem o que deve ser feito, e não como deve ser feito. Adicionalmente, apresentou-se também como a avaliação do modelo é feita e o ambiente de medições utilizado para validar (ver Capítulo 6) e caracterizar a arquitetura.

CAPÍTULO 5

FERRAMENTA PECES

Dado o método de modelagem proposto no capítulo anterior, este capítulo apresenta a ferramenta proposta para avaliação de desempenho e consumo de energia em sistemas embarcados. Adicionalmente, o capítulo também apresenta o método utilizado pela ferramenta para mapear a aplicação no modelo arquitetural.

5.1 MAPEAMENTO

Como mencionado na Seção 4.2, os modelos utilizam a frequência em que cada classe de instrução é executada no programa. Como esta distribuição de frequências é dependente do programa em avaliação, um método apoiado no uso das Cadeias de Markov de Tempo Discreto (DTMC) foi desenvolvido para obter estas frequências.

No método proposto, o Grafo de Fluxo de Controle (*Control Flow Graph* - CFG) do programa é mapeado em uma DTMC irredutível, no qual: (i) cada bloco básico[1] B_i do CFG é mapeado em um estado B_i da DTMC, e (ii) arcos do CFG são mapeados em transições entre estados com probabilidades de transição de acordo com:

$$P(B_i, B_j) = Pr(B_i \, desvia \, para \, B_j) \tag{5.1}$$

que define a probabilidade de executar B_j depois de B_i. Tais probabilidades são obtidas das anotações no código.

A Figura 5.1 mostra um exemplo de código, no qual as anotações são feitas através de comentários. Neste exemplo, a anotação na linha 4 indica que a expressão $x<10$ tem probabilidade de 50% de ser avaliada como verdadeira. A anotação na linha 6 indica que a estrutura iterativa é executada em média 9 vezes.

Para estruturas *while* e *do...while*, as anotações são iguais as da linha 4 do exemplo, no entanto, elas indicam a probabilidade da condição do *loop* ser avaliada como verdadeira. O mesmo se aplica para a estrutura seletiva *switch...case*, na qual as anotações representam as probabilidades de escolha entre as opções. Caso alguma estrutura condicional do código não seja anotada, considera-se que o evento correspondente possui probabilidade de 50% de ocorrer.

Os valores para as anotações podem ser capturados, por exemplo, através: (i) do conhecimento do comportamento dinâmico do código pelo desenvolvedor; (ii) um modelo mais abstrato do sistema, e/ou (iii) técnicas de *profiling*. Diversos cenários de execução podem ser avaliado apenas mudando estes valores.

A Figura 5.2 apresenta a DTMC resultante do mapeamento do exemplo, onde o leitor deve notar uma transição adicional em vermelho do estado 5 para o estado 1 (i.e., do

[1] Um bloco básico é uma sequência de código com apenas um ponto de entrada e um ponto de saída (i.e., sem desvios entrando ou saindo da sequência).

```
1.  int main() {
2.     int x,y
3.
4.     if (x< 10) // <0.5>
5.     {
6.        for(y=0; y< 9 ; y++) { // <9>
7.         x++;
8.        }
9.     } else {  //  <0.5>
10.      x = 0;
11.    }
12.
13. }
```

Figura 5.1 Exemplo de código anotado.

ponto final aplicação para o ponto inicial). Tal transição é adicionada automaticamente para que a DTMC seja transformada em uma DTMC irredutível.

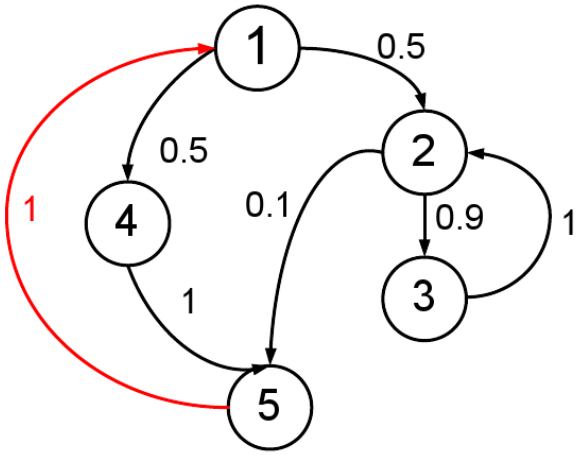

Figura 5.2 DTMC resultante do código anotado da Figura 5.1.

O objetivo em tal mapeamento é obter o número médio de vezes que cada bloco básico é executado (número de visitas). Dado a distribuição estacionária $\pi = (\pi_1, \pi_2, ..., \pi_n)$ da DTMC mapeada, que pode ser obtida através dos métodos numéricos da ferramenta SHARPE, seja $v = (v_1, v_2, ..., v_n)$ o vetor com o número médio de execuções de cada bloco básico $B_1, B_2, ..., B_n$, em que B_n contém o ponto final da aplicação, então v é determinado por (ver Equação 3.2):

$$v = (\frac{\pi_1}{\pi_n}, \frac{\pi_2}{\pi_n}, ..., \frac{\pi_n}{\pi_n}) \tag{5.2}$$

Através do número de vezes que cada bloco básico é executado é possível obter o

número de vezes que cada classe de instrução é executada, e assim, a frequência de execução de cada classe.

5.2 FERRAMENTA PECES

A Figura 5.3 exibe o método de avaliação proposto por este trabalho. Esta figura já foi mostrada no Capítulo 1 e é replicada aqui com mais detalhes (o passo de compilação foi expandido). No lado direito da figura, existe a modelagem da arquitetura que é feita através de um processo de composição, como mostrado no Capítulo 4. Em seguida (lado esquerdo da figura), o código fonte anotado da aplicação é transformado em uma DTMC irredutível, que é avaliado para que sejam obtidas informações sobre a frequência que cada classe de instrução é executada no programa. Esta informações são anotadas no modelo da arquitetura, que por sua vez é avaliado por simulação para que as informações sobre o consumo de energia e desempenho da aplicação sejam obtidas.

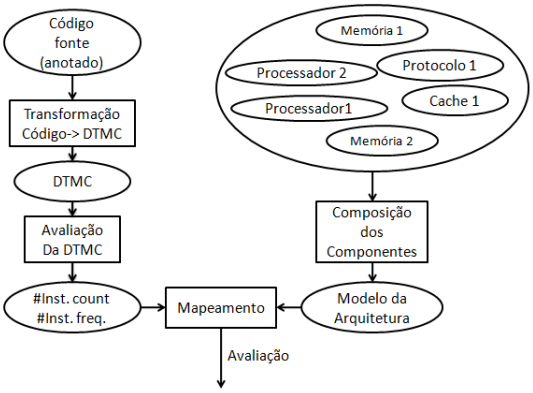

Figura 5.3 Método proposta.

Este trabalho construiu uma ferramenta, denominada PECES (**P**erformance and **E**nergy **C**onsumption **E**valuation of Embedded **S**ystems), para automatizar alguns passos do método. Esta ferramenta (ver Figura 5.4) recebe como entrada o código fonte anotado da aplicação e o modelo da arquitetura (em formato XML compatível com a ferramenta CPN Tools), e retorna como saída o consumo de energia e tempo de execução da aplicação.

Os passos seguidos por PECES para avaliar um código são descritos abaixo:

1. PECES compila o código fonte da aplicação utilizando a opção de geração de arquivo objeto intermediário assembly. GCC (arm-uclibc-gcc [Keib]) foi o compilador utilizado por este trabalho.

2. A ferramenta utiliza o código intermediário assembly do passo anterior para gerar o CFG. O algoritmo de criação do CFG é descrito no Apêndice B.

3. O CFG gerado no passo 2 em conjunto com o código fonte (usado para capturar as probabilidades) é utilizado para gerar a DTMC ergódica. A DTMC é gerada em um formato compatível com a ferramenta SHARPE.

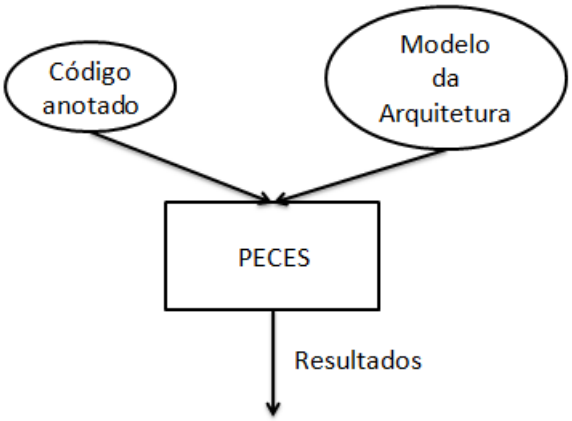

Figura 5.4 Ferramenta PECES.

4. PECES utiliza a ferramenta SHARPE para resolver numericamente a DTMC e obter as probabilidades de estado estacionário.

5. Em seguida, as probabilidades de estado estacionário são utilizadas para obter o número de vezes em que cada bloco básico é executado, e assim, obter a quantidade de vezes em que cada instrução é executada. PECES agrupa instruções de mesma classe e calcula a frequência em que cada classe é executada.

6. A distribuição de frequência é escrita no arquivo XML que representa o modelo da arquitetura.

7. PECES chama a ferramenta Access/CPN [WK09] para iniciar a simulação do modelo arquitetural.

8. Ao final da simulação, PECES utiliza as métricas de tempo médio por instrução e consumo médio por instrução geradas pelo passo anterior, para finalmente retornar os resultados.

9. Adicionalmente, a ferramenta constrói um histograma contendo os resultados relativos às médias dos lotes (ver Seção 4.4) da simulação.

Desenvolvida através da linguagem de programação Python, a ferramenta PECES possui interface textual. A Figura 5.5 exibe um exemplo de avaliação através desta ferramenta. Na linha de comando, o primeiro argumento da chamada é o código em avaliação, e o segundo, o modelo da arquitetura.

5.3 CONSIDERAÇÕES FINAIS

Este Capítulo apresentou a abordagem adotada para capturar a frequência em que cada classe de instrução é executada no programa em avaliação. Tal abordagem mapeia o código da aplicação em uma DTMC que por sua vez é avaliada com o intuito de obter

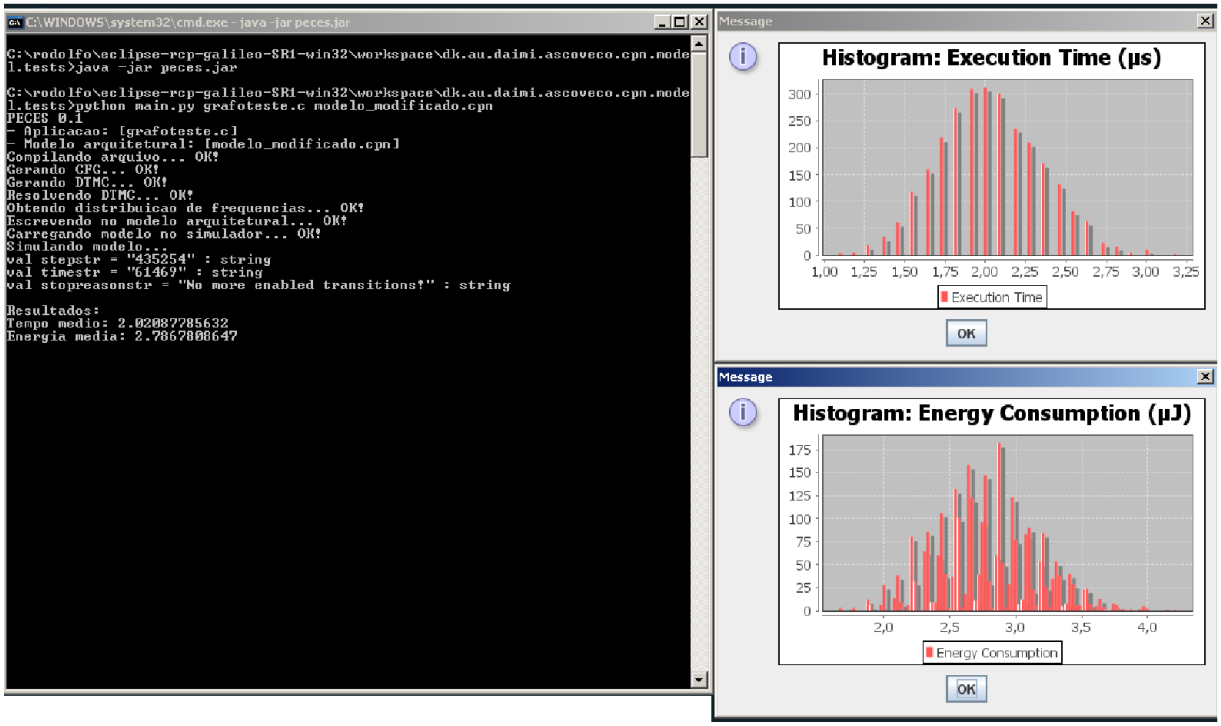

Figura 5.5 Avaliação de um código através da ferramenta PECES.

o número médio de vezes em que cada bloco básico no programa é executado. Através do número de vezes em que cada bloco básico é executado é possível obter o número de vezes em que cada classe de instrução é executada, e assim, a frequência de execução de classe.

Adicionalmente, o capítulo apresentou o método concebido para avaliação e a ferramenta PECES, criada para automatizar o processo de avaliação. Esta ferramenta permite que os detalhes da simulação e do modelo formal da simulação fiquem transparentes ao projetista. Assim, não é necessário que o usuário possua qualquer conhecimento sobre as Redes de Petri Coloridas para utilizar PECES.

CAPÍTULO 6

ESTUDOS DE CASO

Para avaliar o método proposto, alguns estudos de caso foram conduzidos. Inicialmente, estudos de caso considerando a exatidão do método proposto são apresentados. Em seguida, o método é comparado em termos de velocidade com o método proposto em [Cal09]. Depois, estudos de caso considerando possíveis aplicações para o método são apresentados. Por último, um estudo de caso considerando arquiteturas multiprocessadas é introduzido.

Todos os experimentos foram conduzidos em uma máquina com a seguinte configuração: Intel Core 2 Duo 1.67GHz, 2Gb RAM e Windows Vista OS.

6.1 VALIDAÇÃO

Os estudos de caso utilizados para validar o método proposto são compostos pelas aplicações descritas a seguir:

- adpcm, bcnt e fdct: Estas aplicações foram propostas pela Motorola e fazem parte do pacote *Power Stone Benchmark* [MMC00]. Este pacote utiliza algoritmos de diferentes segmentos do mercado de sistemas embarcados como, por exemplo, algoritmos de processamento de sinais, imagem e automação, para caracterizar o consumo de energia de uma arquitetura.

- binarysearch, bubblesort e convolution: Estas aplicações são algoritmos comuns para busca em vetores (binarysearch), ordenação em vetores (bubblesort) e processamento de sinais (convolution).

- oxímetro de pulso [Jun98]: Este experimento consiste em uma aplicação real embarcada. O oxímetro é um equipamento responsável por medir o nível de saturação do oxigênio utilizando um método não invasivo. A Figura 6.1 exibe a estrutura genérica de um oxímetro de pulso, onde é possível observar o Mostrador da Interface com usuário, a unidade microprocessada (denominada Unidade μP na figura) e os elementos analógicos responsáveis pela aquisição e condicionamento do sinal proveniente do meio biológico. O sistema da unidade microprocessada é dividido em três partes distintas, denominadas planos. Os planos são tarefas que executam independentemente uma das outras. Seguindo este conceito, nomeia-se os planos de acordo com a atividade macro que eles desempenham. São elas: (i) Plano 1 - Rotina de Excitação; (ii) Plano 2 - Rotinas de Amostragem e Controle; (iii) Plano 3 - Programa Gerenciador. Neste estudo de caso, considerou-se para avaliação apenas o Plano 1 e o Plano 2, dado que o Plano 3 diz respeito à funções de inicialização do sistema, atendimento aos comandos do usuário entre outras atividades de menor relevância neste contexto.

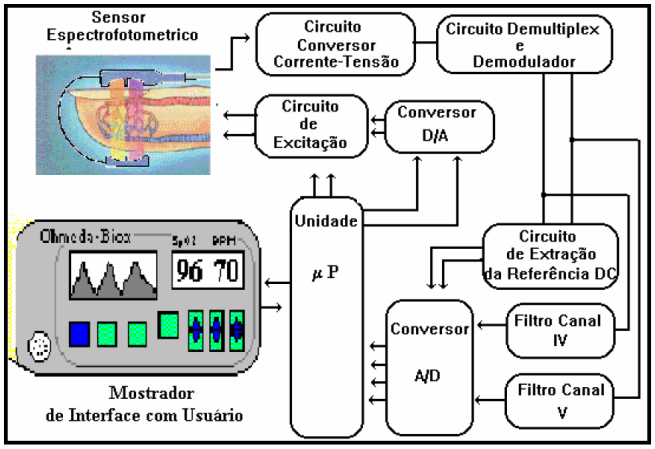

Figura 6.1 Estrutura do Oxímetro de Pulso [Jun98].

Para que o leitor tenha uma idéia da complexidade das aplicações adotadas, a Tabela 6.1 exibe o número de blocos básicos e o número médio de instruções executadas para cada aplicação. O tempo tomado por PECES para obter estes dados através do mapeamento em uma DTMC irredutível é relativamente rápido: menos de 1 segundo para cada aplicação.

Tabela 6.1 Complexidade das aplicações.

	Número de blocos básicos	Número médio de instruções executadas
adpcm	30	433299
bcnt	6	1972
binary search	7	182
bubble sort	7	189207
convolution	6	23575
fdct	4	3478
oxímetro (plano 1)	14	440,9775
oxímetro (plano 2)	17	422
oxímetro (plano 3)	105	121169,225

A Tabela 6.2 e Tabela 6.3 exibem, respectivamente, a estimativa de tempo de execução e consumo de energia para cada estudo de caso e sua comparação com os valores medidos na plataforma real (através da ferramenta AMALGHMA - ver Seção 4.5). O erro médio foi de %3,83 para o tempo de execução e 3,23% para o consumo de energia. O erro máximo foi de %10,41 e %10,03 para o tempo de execução e consumo de energia, respectivamente.

As estimativas do método proposto foram analisadas estatisticamente através da técnica de bootstrap [ET97]. Para realizar tal análise, foram utilizadas os percentuais das diferenças dos dados estimados e medidos para o tempo de execução e consumo

Tabela 6.2 Resultados experimentais: tempo de execução (medido em μs).

	Medido	Estimado	Erro
adpcm	13080,1	12397,3	-5,22%
bcnt	56,1	55,8	-0,53%
binary search	5,8	5,9	1,72%
bubble sort	6138,3	6140,3	0,03%
convolution	1077	964,9	-10,41%
fdct	90,9	93,4	2,75%
oxímetro (plano 1)	11,6	11,9	2,59%
oxímetro (plano 2)	11,7	12,3	5,13%
oxímetro (plano 3)	3357,2	3379	0,65%

Tabela 6.3 Resultados experimentais: consumo de energia (medido em μJ).

	Medido	Estimado	Erro
adpcm	1097,2	1059,4	-3,45%
bcnt	4,6	4,6	0%
binary search	0,5	0,5	0%
bubble sort	5247,6	5172,8	-1,43%
convolution	80,1	76,9	-4%
fdct	8,2	7,7	-6,1%
oxímetro (plano 1)	1,01	0,99	-1,98%
oxímetro (plano 2)	1,07	0,99	-7,48%
oxímetro (plano 3)	257,2	283	10,03%

de energia (coluna 4 das Tabela 6.2 e Tabela 6.3, respectivamente). Um método não paramétrico como bootstrap foi utilizado porque não é possível assumir que os percentuais das diferenças são normalmente distribuídos.

Realizando um experimento bootstrap para a diferença percentual do consumo de energia baseado em 1000 iterações e com um intervalo de confiança de 95%, o seguinte intervalo foi obtido para média da diferença: [-4,37778 ; 1,85] (ver Figura 6.2(a)). Como este intervalo inclui o 0, pode-se inferir com 95% de certeza que estatisticamente não há evidências significativas que refutem a afirmação de que o modelo consegue representar o sistema real adequadamente para a métrica de consumo de energia. Similarmente, realizando o mesmo experimento para a diferença do tempo de execução, o seguinte intervalo foi obtido: [-3,90111 ; 2,22889] (ver Figura 6.2(b)). Como este intervalo também inclui o 0, a mesma afirmativa anterior para o tempo de execução pode ser feita.

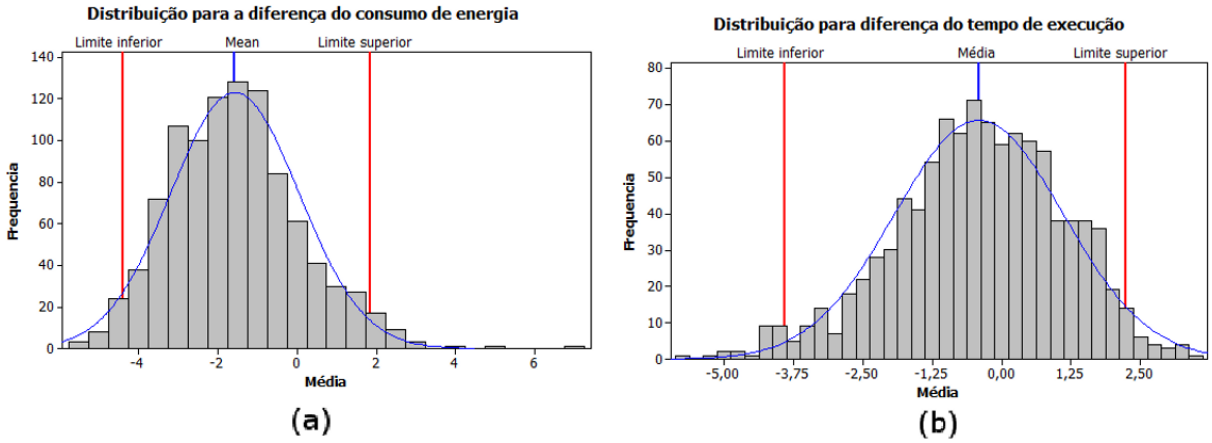

Figura 6.2 Distribuição do bootstrap para a média.

6.2 COMPARAÇÃO ENTRE MÉTODOS

O experimento do oxímetro foi utilizado para comparar, em termos de velocidade, o método proposto com o método apresentado por Callou et al. [CMC+09, Cal09], que também utilizou, para fins de validação, o mesmo microcontrolador usado por este trabalho. A Figura 6.3 apresenta resultados quantitativos da comparação, onde A1 representa a abordagem de Callou et al., e A2, o método proposto por este trabalho. Os resultados mostram que o tempo de simulação nas duas abordagens é aproximadamente igual, exceto para o Plano 3. Neste plano, o método proposto foi 542 vezes mais rápido. A grande diferença é principalmente porque o trabalho descrito em [CMC+09] simula o fluxo de controle da aplicação, portanto, o modelo de simulação assim como o tempo de simulação crescem com o tamanho do código. Por outro lado, no método proposto por este trabalho, o modelo é fixo e a variação ocorre apenas nas anotações relacionadas a frequência em que cada classe de instrução é executada. Isto faz com que o modelo seja escalável com o tamanho do código.

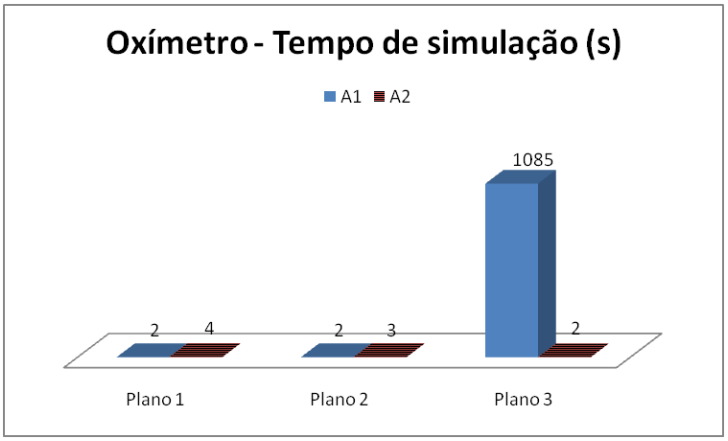

Figura 6.3 Comparação do tempo de simulação.

Uma comparação justa em termos de exatidão das estimativas não pôde ser feita porque o trabalho de Callou et al. não considerou o mecanismo de aceleração da memória (MAM - ver Seção 4.3), i.e., este mecanismo foi desligado durante a caracterização da arquitetura e validação através dos experimentos [Cal09]. A desativação da MAM é pouco utilizada na prática, uma vez que o MAM é responsável por grandes ganhos de desempenho [Mar06]. Em todo caso, os experimentos conduzidos por Callou et al. com o MAM desligado demonstram um erro médio de 4%.

6.3 APLICAÇÕES DO MÉTODO

Otimizações de código como *loop unrolling* [DJ96] e *function inlining* [LM99], têm se mostrado como técnicas de sucesso para otimizar o desempenho de um sistema. Uma aplicação muito útil do método proposto é verificar o efeito que estas técnicas comuns de otimização têm sobre o consumo de energia. O experimento do código bubblesort foi utilizado para demonstrar como tal análise pode ser realizada.

Usando as técnicas de *loop unrolling* e *function inlining*, o código do bubblesort foi otimizado em quatro passos. Partindo do código original, a cada novo passo, otimizações mais agressivas foram incluídas. A Figura 6.4 apresenta os resultados desse experimento. Pode ser observado que ao aplicar tais otimizações, o consumo de energia foi otimizado em 225%. O erro médio para as estimativas foi de 4,38%, mostrando que o método proposto pode ser aplicado com sucesso para analisar otimizações de código considerando o consumo de energia.

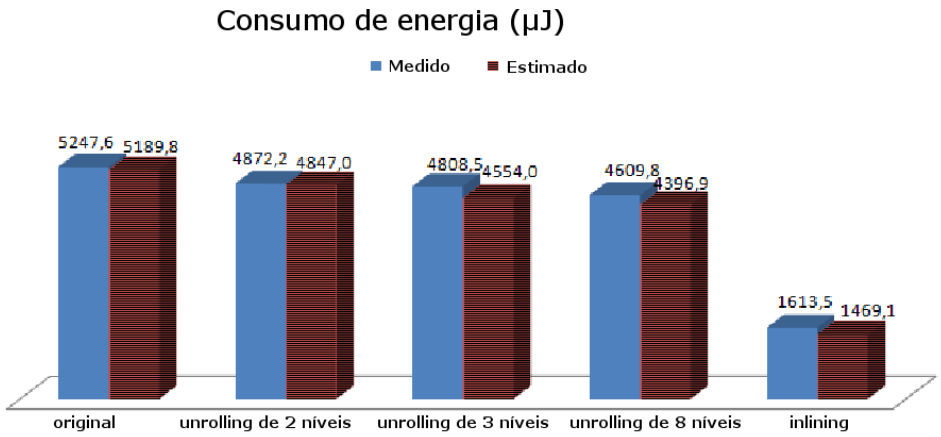

Figura 6.4 Otimizações de código.

O método proposto também é útil quando o interesse do projetista é avaliar cenários de execução do código, tais como: melhor cenário (*best-case*), cenário médio (*average-case*) e pior cenário (*worst-case*). O código do bubblesort foi utilizado para avaliar esta aplicação do método.

O código do bubblesort é exibido na Figura 6.5, onde o leitor deve notar que toda variação de fluxo do código é definida através das estruturas das linhas 6, 8 e 10. O número de iterações das estruturas de controle das linhas 6 e 8 são dependentes apenas do tamanho

```
1  void BubbleSort(int Array[])
2  {
3     int i, j;
4     int k = NUMELEMS-1;
5
6     for(i = 0; i < NUMELEMS; i++)     // <100>
7     {
8        for(j = 0; j < k; j=j+1)       // <4950>
9        {
10          if(Array[j] > Array[j+1])   // <0.5>
11          {
12            swap(Array, j, j+1);
13          }
14       }
15        k--;
16    }
17 }
```

Figura 6.5 Algoritmo do bubblesort.

do array, e portanto, possuem um comportamento determinístico. Por outro lado, a estrutura de controle na linha 10 possui um comportamento probabilístico, dependente do nível de ordenamento do array.

No pior cenário, o array está completamente desordenado, assim a função *swap* será chamada todas as vezes. Tal cenário pode ser avaliado mudando o valor da anotação na linha 10 para 1. No cenário de melhor caso, o array está completamente ordenado. Neste caso, a função *swap* nunca será chamada. Mudando o valor da linha 10 para 0, este cenário pode ser avaliado. Por outro lado, o cenário médio de operação acontece quando o array está parcialmente ordenado. Tal cenário pode ser avaliado anotando o valor 0,5 na linha 10. A Tabela 6.4 exibe os resultados para cada cenário. Os valores estimados para o tempo de execução apresentaram um erro médio de 1,69% e erro máximo de 3,94%. Com relação ao consumo de energia, o erro médio foi de 2,34% e o máximo de 5,24%

Tabela 6.4 Cenários de execução do bubblesort.

	Tempo de execução (μs)			Consumo de energia (μJ)		
	Estimado	Medido	Erro	Estimado	Medido	Erro
melhor cenário	2414.6	2432,5	0.74%	2028.9	2015.6	0.66%
cenário médio	4086.6	4247,8	3.94%	3453.4	3634.4	5.24%
pior cenário	6162.3	6138.3	0.39%	5189.8	5247.6	1.11%

Como informado na Seção 4.3, a taxa de acerto no MAM deve ser dada para que seja possível realizar avaliações com exatidão (este trabalho adotou um simulador de traces para que tal informação fosse obtida, como já informado). No entanto, resultados significativos também podem ser obtidos se consideramos o consumo de energia (ou tempo de execução) em função das variações na taxa de acerto do MAM. A Figura 6.6 apresenta o consumo de energia estimado para o código do bubblesort em função da taxa de acerto no MAM, onde pode ser observado que o consumo de energia cresce a medida que a taxa de acerto diminui.

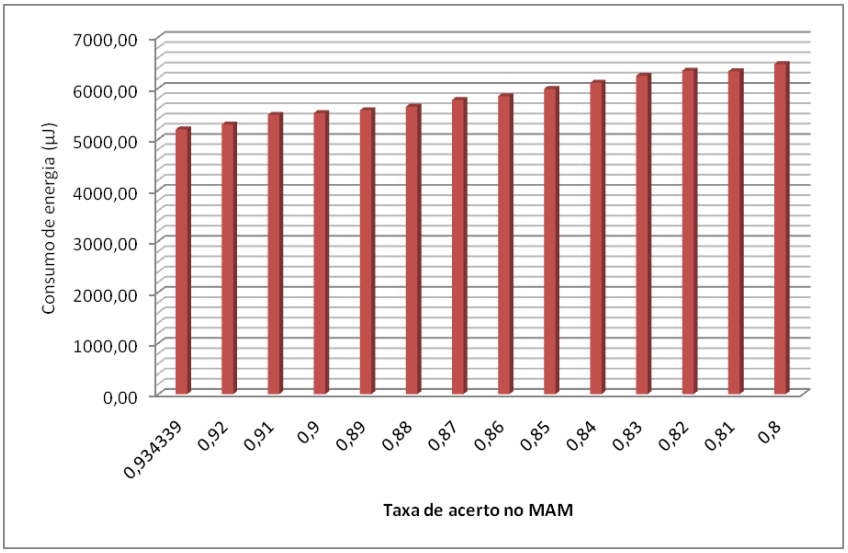

Figura 6.6 Consumo de energia em função da taxa de acerto na MAM para o código do bubblesort.

6.4 MODELANDO ARQUITETURAS MULTIPROCESSADAS

Esta seção apresenta como os *building blocks* podem ser usados para representar arquiteturas mais complexas. Considere a situação exibida na Figura 6.7, onde dois LPC2106 compartilham uma memória externa. A interface da memória externa só suporta uma escrita a cada dois ciclos, ao passo que tal limitação não existe para acessos de leitura. Os pedidos de acesso são colocados em uma fila e processados com política FIFO (*First In-First Out*). É importante lembrar que cada LPC2106 também está conectado a duas memórias internas contendo código e dados.

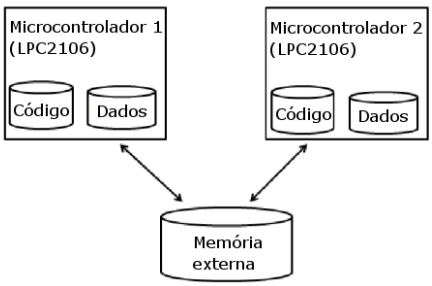

Figura 6.7 Ambiente multiprocessado.

A plataforma descrita acima foi modelada replicando o modelo já apresentado para o LPC2106 e criando um novo *building block* para representar a memória externa. A Figura 6.9 apresenta uma visão em alto nível do modelo proposto para este ambiente e a Figura 6.8 mostra o *buidling block* que representa a memória externa. A Listagem 6.1 exibe as declarações deste novo *building block*.

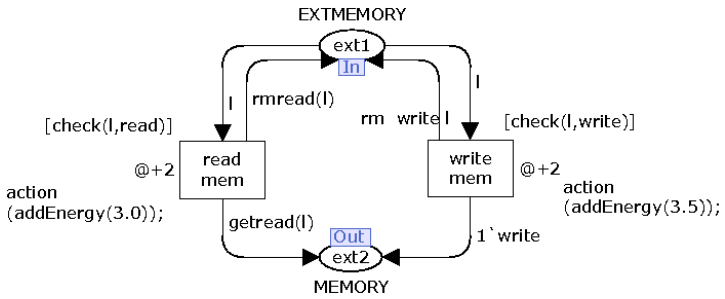

Figura 6.8 Modelo da memória externa.

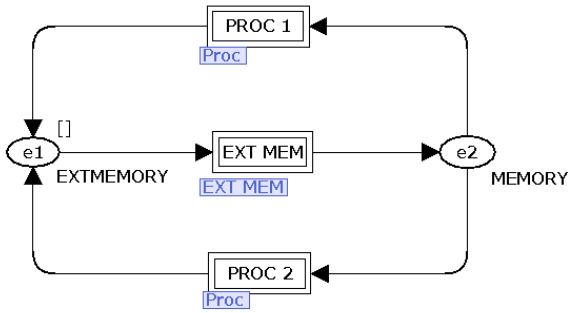

Figura 6.9 Modelo em alto nível para representar a plataforma da Figura 6.7.

Listagem 6.1 Declarações do building block que representa a memória externa.

```
1  colset MEMORY = with read | write timed;
2  colset EXTMEMORY = list MEMORY timed;
3
4  var l: EXTMEMORY;
5
6  fun rmread(x::l) = if x=read then rmread(l) else x::l |
7  rmread([])=[];
8
9  fun getread(x::l) = if x=read then read::getread(l) else [] |
10 getread([])=[];
11
12 fun check(x::l, t) = if x = t then true else false |
13 check([],t) = false
```

A interface entre os modelos que representam os microcontroladores e o *building block* que representa a memória externa é feita através das portas de entrada e saída *ext1* e *ext2*, respectivamente. A porta de entrada *ext1* possui conjunto de cores *EXTMEMORY* (lista que armazena valores do tipo *MEMORY*) e modela a fila de requisições.

As transições *read mem* e *write mem*, modelam as operações de leitura e escrita na memória externa, respectivamente. A função *check*, presente nas guardas destas transições, é utilizada para verificar qual o tipo da próxima requisição na fila. O leitor deve notar, por exemplo, que o uso desta função na guarda da transição *read mem* verifica se a próxima requisição é uma leitura, caso seja, a função retorna verdadeiro, caso contrário, falso.

A função *rmread*, presente no arco (*read mem, ext1*) remove todas as requisições de leitura da fila. Isto modela uma operação de leitura em que várias requisições de leitura

são satisfeitas de uma só vez. Por outro lado, a anotação *rm write l*, no arco (*write mem, ext1*), remove apenas a primeira requisição de escrita da fila. A função *getread* é usada para gerar a mesma quantidade de tokens com valor *read* na porta de saída *ext2* que a quantidade de requisições satisfeitas durante uma operação de leitura. Assim, se por exemplo, existirem duas requisições de leitura na fila e uma operação de leitura é feita (satisfazendo as duas requisições de uma vez), esta função faz com que sejam gerados dois tokens com valor *read* no lugar *ext2*.

Considere a marcação exibida na Figura 6.10, ela representa situação em que uma requisição de escrita e duas requisições de leitura estão na fila. Como a primeira requisição é uma requisição de escrita, apenas a transição *write mem* está habilitada. A Figura 6.11 exibe a marcação após o disparo da transição *write mem*, onde a requisição de escrita é removida da fila e um token com valor *write* é gerado na porta de saída *ext2*. Agora, como a primeira requisição da fila é uma leitura, apenas a transição *read mem* está habilitada. A marcação após o disparo de *read mem* é exibida na Figura 6.12, onde o leitor deve notar que como as duas requisições de leitura foram satisfeitas de uma só vez, as requisições de leitura foram removidas da fila e dois tokens com valor *read* foram adicionados na porta de saída *ext2*. Na marcação atual, como não existem mais requisições na fila, as transições *read mem* e *write mem* estão desabilitadas.

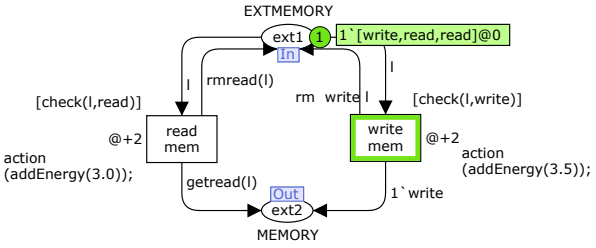

Figura 6.10 Marcação do modelo da Figura 6.8 - fila com uma requisição de escrita e duas de leitura.

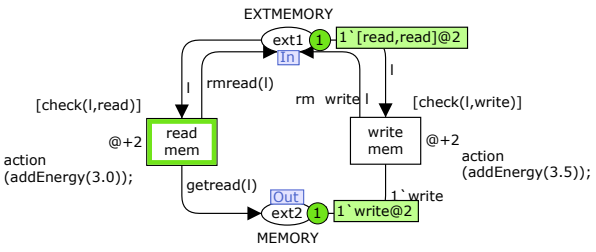

Figura 6.11 Marcação do modelo da Figura 6.8 - operação de escrita.

Este modelo foi avaliado usando o experimento do código adpcm, no qual um código adpcm executa em cada microcontrolador. Assumiu-se que 30% das instruções de memória do código adpcm acessam a memória externa e que não existem dependências de dados entre os dois códigos. A Tabela 6.5 apresenta os resultados para o ambiente com dois microcontroladores descrito acima e os resultados relativos a execução da mesma aplicação

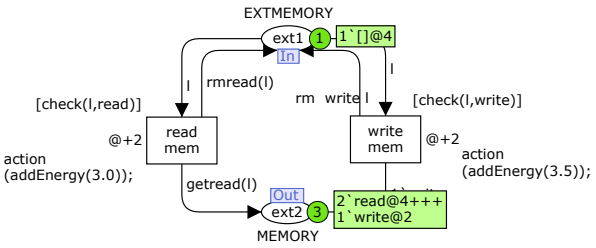

Figura 6.12 Marcação do modelo da Figura 6.8 - operação de leitura.

em apenas um processador (já apresentados nas Tabelas 6.2 e 6.3). Comparando os resultados, é possível observar que o consumo de energia do código adpcm quase dobrou, uma vez que além do consumo adicional da memória externa, dois processadores consomem mais energia que apenas um processador. Por outro lado, o tempo de execução permaneceu quase o mesmo. Na verdade, já que a memória externa introduz um gargalo, existe um pequeno aumento nesse valor. No entanto, é possível observar que a execução de dois adpcm em paralelo é aproximadamente duas vezes mais rápida que a execução de dois adpcm em sequencia.

Tabela 6.5 Resultados da avaliação do ambiente multiprocessado.

	Tempo de simulação (s)	Consumo de energia (μJ)	Tempo de execução (μs)
adpcm (1 microcontrolador)	7	1059,4	12397,3
adpcm (2 microcontroladores)	17	2408,5	13118,4

6.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou estudos de caso para avaliar o método proposto de estimativa consumo de energia e desempenho em sistemas embarcados. Inicialmente, estudos de caso para verificar a exatidão do método proposto foram apresentados, nos quais um erro médio de 4% foi verificado. Adicionalmente, as estimativas foram avaliadas estatisticamente através do método de bootstrap. Os resultados demonstram que não há evidências significativas para refutar a capacidade de representação do modelo frente à plataforma avaliada, o LPC2106. Em seguida, o método proposto foi comparado em termos de velocidade de avaliação com a proposta recente de Callou et al. [Cal09, CMC+09]. Nos experimentos feitos para realizar tal comparação, o método proposto por este trabalho foi até 542 vezes mais rápido. Depois, mostrou-se alguns exemplos de aplicações do método proposto, dentre os quais ressalta-se a possibilidade da avaliação de otimizações de código. Por fim, um exemplo de modelagem de uma arquitetura multiprocessada foi apresentado.

CAPÍTULO 7

CONCLUSÃO

O projeto de sistemas embarcados de baixo consumo tem crescido em importância com a proliferação de dispositivos embarcados operados por bateria. Além de garantir que as funcionalidades estão corretamente implementadas, um dos principais objetivos neste tipo de projeto é reduzir o consumo de energia sem comprometer os requisitos de desempenho. Assim, é essencial dispor, ainda nas fases iniciais de desenvolvimento, de mecanismos que auxiliem de forma rápida e precisa a avaliação de possíveis alternativas de projeto.

Neste contexto, diversas abordagens têm sido propostas ao longo dos últimos anos (ver Capítulo 2). Algumas destas abordagens se baseiam em modelos da arquitetura embarcada para estimar consumo de energia e desempenho. Apesar da boa exatidão das estimativas, o baixo nível dos modelos adotados por estes métodos demanda longo tempo de avaliação, o que restringe a aplicabilidade para códigos grandes. Outra dificuldade é que para realizar as estimativas é necessário ter acesso a detalhes da tecnologia de implementação do hardware, que constituem, normalmente, propriedades intelectuais.

Este trabalho apresentou uma estratégia de modelagem de eventos discretos baseada em Redes de Petri Coloridas (CPN) para estimar consumo de energia e desempenho. O método proposto se baseia arquitetura da plataforma embarcada para realizar as estimativas. No entanto, diferentemente dos métodos existentes que adotam este tipo de abordagem, os modelos propostos possuem alto nível de abstração, permitindo flexibilidade e velocidade. Adicionalmente, os modelos são baseados em dados de consumo de energia e desempenho obtidos através de medições na plataforma, o que faz com que o método proposto possua boa exatidão. A abordagem baseada em medições também permite que estimativa seja feita sem que sejam necessárias descrições de baixo nível da arquitetura, uma vez que os dados de consumo e desempenho são obtidos diretamente da plataforma.

No método proposto, as CPN são utilizadas para modelar o comportamento funcional de processadores e arquiteturas de memória. A construção do modelo arquitetural é feita através de um processo de composição de *building blocks* que representam as unidades funcionais da arquitetura. As operações modeladas nestes *building blocks* possuem anotações de valores para representar o consumo de energia e desempenho demandado pela respectiva operação. Tais valores são obtidos por medição através da ferramenta AMALGHMA, adotada para automatizar o processo de medição. Adicionalmente, por usar uma abordagem probabilística, os modelos são livres detalhes de implementação da arquitetura.

Em seguida, o código que executará na plataforma embarcada é mapeado no modelo arquitetural. O mapeamento se dá através da inserção de anotações relativas à frequência em que determinadas classes de instruções da arquitetura são executadas no código em avaliação. Para obter estas frequências, o código da aplicação é primeiramente mapeado

em uma Cadeia de Markov de Tempo Discreto (DTMC), que por sua vez é avaliada com o intuito de obter o número médio de vezes que cada bloco básico do programa é executado. Através do número de vezes que cada bloco básico é executado é possível obter o número de vezes que cada classe de instrução é executada, e assim, a frequência de execução de classe.

Por último, a avaliação do modelo é feita através de simulação estocástica. A abordagem de simulação adotada se baseia no método das Médias de Lotes para obter resultados com confiança levando em consideração a precisão desejada.

Uma ferramenta, denominada PECES (**P**erformance and **E**nergy **C**onsumption **E**valuation of Embedded **S**ystems), foi criada para automatizar o processo de avaliação. Esta ferramenta permite que os detalhes da simulação e do modelo formal da simulação fiquem transparentes ao projetista. Assim, não é necessário que o usuário possua qualquer conhecimento sobre as Redes de Petri Coloridas para utilizar PECES.

Para fins de validação, este trabalho considerou um microcontrolador baseado na arquitetura ARM. Estudos de caso considerando aplicações de diversos tamanhos e características foram adotados para verificar a eficácia do método proposto. Em comparação aos valores reais medidos na plataforma, as estimativas para os estudos de caso conduzidos apresentaram boa acurácia (erro médio em torno de 4%). Adicionalmente, as estimativas foram avaliadas estatisticamente através do método de bootstrap, mostrando que não há evidências estatísticas significativas para refutar a capacidade de representação do modelo frente à plataforma avaliada.

O método proposto foi comparado em termos de velocidade de avaliação com a proposta recente de Callou et al. [Cal09, CMC+09], que estima consumo de energia e desempenho através da simulação do fluxo de controle do software. Nos experimentos feitos para realizar tal comparação, o método proposto por este trabalho foi até 542 vezes mais rápido.

Além de estudos de caso para validar e avaliar a velocidade do método proposto, outros estudos de caso foram realizados com o intuito de analisar os seguintes cenários de aplicação do método proposto: (i) avaliação do impacto no consumo de energia de otimizações de código; (ii) avaliação de cenários de execução do programa, tais como: melhor cenário (*best-case*), cenário médio (*average-case*) e pior cenário (*worst-case*); (iii) avaliação do impacto da arquitetura de memória sobre as métricas; e (iv) avaliação de arquiteturas multiprocessadas. Os resultados demonstram que, em todas estes cenário de aplicação, o método proposto pode ser usado para garantir um *feedback* rápido e confiável ao desenvolvedor.

7.1 LIMITAÇÕES E TRABALHOS FUTUROS

No decorrer do desenvolvimento deste trabalho, algumas limitações do método foram verificadas. Estas limitações, descritas abaixo, são boas oportunidades para trabalhos futuros:

- **Validação de outras arquiteturas**: Por falta da infra-estrutura necessária, infelizmente o estudo de caso considerando arquiteturas multiprocessadas não foi validado frente uma plataforma real multiprocessada. Assim, pretende-se como

trabalho futuro validar a arquitetura modelada, assim como outras arquiteturas, tais como: arquiteturas VLIW e Superscalar.

- **Ferramenta PECES**: A ferramenta PECES possui apenas interface textual, o que é uma limitação para muitos usuários. Portanto, uma ferramenta com interface gráfica seria uma interessante alternativa como trabalho futuro. Adicionalmente, esta ferramenta só auxilia na avaliação depois que a arquitetura foi modelada. Portanto, pretende-se estender esta ferramenta de forma que ela auxilie de forma gráfica o processo de composição dos *building blocks*.

- **Regras formais de composição**: Este trabalho apresentou uma visão intuitiva de como *building blocks* podem ser compostos. Em trabalhos futuros, pretende-se definir regras formais para esta composição.

REFERÊNCIAS BIBLIOGRÁFICAS

[ama] Amalghma tool. http://www.cin.ufpe.br/∼eagt. 4.5

[AMCN09] Ermeson Andrade, Paulo Maciel, Gustavo Callou, and Bruno Nogueira. Mapping uml sequence diagram to time petri net for requirement validation of embedded real-time systems with energy constraints. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing, 2009*. ACM, 2009. 1.2

[AMG] T. Austin, T. Mudge, and D. Grunwald. The SimpleScalar-Arm Power Modeling Project. http://www.eecs.umich.edu/∼jringenb/power/. 2.2

[And09] E. Andrade. Modelagem e análise de especificações de sistemas embarcados de tempo-real críticos com restrições de energia. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, 2009. 1.2

[ARM01] ARM Limited. *ARM7TDMI-S Technical Reference Manual (Rev. 4)*, 2001. 4, 4.1, 4.5

[art04] Building artemis. Technical report, The High-level Group on Embedded Systems, 2004. 1

[BA97] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997. 2.2

[Ban99] J. Banks. Introduction to simulation. In *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1*, pages 7–13. ACM New York, NY, USA, 1999. 3.3

[BB95] T.D. Burd and R.W. Brodersen. Energy efficient CMOS microprocessor design. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, volume 1060, 1995. 2

[BBR+07] H. Blume, D. Becker, L. Rotenberg, M. Botteck, J. Brakensiek, and TG Noll. Hybrid functional-and instruction-level power modeling for embedded and heterogeneous processor architectures. *Journal of Systems Architecture*, 53(10):689–702, 2007. 2.1

[BBS+03] D. Brooks, P. Bose, V. Srinivasan, M. Gschwind, P. Emma, and M. Rosenfield. Microarchitecture-level power-performance analysis: the powertimer approach. *IBM J. Research and Development*, 47(5), 2003. 2.2

[BCSRM06] CJ Bleakley, M. Casas-Sanchez, and J. Rizo-Morente. Software level power consumption models and power saving techniques for embedded dsp processors. *Journal of Low Power Electronics*, 2(2):281–290, 2006. 1

[BGdMT05] G. Bolch, S. Greiner, H. de Meer, and K.S. Trivedi. *Queueing networks and Markov chains*. Wiley-Interscience, 2005. 3, 3.2, 3.3

[BNYT93] R. Burch, FN Najm, P. Yang, and TN Trick. A Monte Carlo approach for power estimation. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 1(1):63–71, 1993. 2.2

[BR04] R. Bianchini and R. Rajamony. Power and energy management for server systems. *Computer*, pages 68–76, 2004. 1

[BSS+05] A. Bona, M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon. Reducing the complexity of instruction-level power models for VLIW processors. *Design Automation for Embedded Systems*, 10(1):49–67, 2005.

[BTM00] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architecture News*, 28(2):83–94, 2000. (document), 2.2, 2.3

[Cal09] G. Callou. Energy consumption and execution time estimation of embedded system applications. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, 2009. 2.1, 6, 6.2, 6.2, 6.5, 7

[CG99] C. Chakrabarti and D. Gaitonde. Instruction level power model of microcontrollers. In *IEEE International Symposium on Circuits and Systems*, pages 76–79. Citeseer, 1999. 2.1

[Chu04] C.A. Chung. *Simulation Modeling Handbook: A Practical Approach*. CRC Press, 2004.

[CIB98] R.Y. Chen, M.J. Irwin, and R.S. Bajwa. An architectural level power estimator. In *Proceedings of the Power-Driven Microarchitecture Workshop*, pages 87–91, 1998.

[CL08] C.G. Cassandras and S. Lafortune. *Introduction to discrete event systems*. Springer Verlag, 2008. 3

[CMC+09] G. Callou, P. Maciel, E. Carneiro, B. Nogueira, E. Tavares, and M. Oliveira Jr. A Formal Approach for Estimating Embedded System Execution Time and Energy Consumption. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, page 388. Springer, 2009. 2.1, 6.2, 6.5, 7

[Con03] A.P.C.A.P. Conversion. Determining total cost of ownership for data center and network room infrastructure, 2003. 1

[DJ96] J.W. Davidson and S. Jinturkar. Aggressive Loop Unrolling in a Retargetable Optimizing Compiler. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 59–73. Springer-Verlag London, UK, 1996. 6.3

[DLCD00] A. Dhodapkar, C.H. Lim, G. Cai, and W.R. Daasch. TEM2P2EST: A Thermal Enabled Multi-model Power/Performance ESTimator. In *Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 112–125. Springer-Verlag London, UK, 2000. 2.2

[DR98] J. Desel and W. Reisig. Place/Transition Petri Nets. *Lectures on Petri Nets: Advances in Petri Nets.*, page 122, 1998. 3.1.1

[ET97] B. Efron and R.J. Tibshirani. *An introduction to the bootstrap*. Chapman & Hall, 1997. 6.1

[FFY05] J.A. Fisher, P. Faraboschi, and C. Young. *Embedded computing*. Elsevier, 2005. 1

[FMS06] S. Friedenthal, A. Moore, and F. Steiner. *OMG Systems Modeling Language (OMG SysML) Tutorial*. INCOSE Intl. Symp, 2006. 1.2

[Fur00a] S. Furber. *Arm System-On-Chip Architecture*. Addison-Wesley, 2000.

[Fur00b] S.B. Furber. *ARM system-on-chip architecture*. Addison-Wesley Professional, 2000. (document), 4.1, 4.1

[GDKW92] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of average switching activity in combinational and sequential circuits. In *Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 253–259. IEEE Computer Society Press Los Alamitos, CA, USA, 1992. 2.2

[H+05] A. Helmerich et al. Study of Worldwide Trends and R&D Programmes in Embedded Systems, 2005. (document), 1, 1.1

[HJS91] P. Huber, K. Jensen, and R.M. Shapiro. Hierarchies in Coloured Petri Nets. In *Proceedings on Advances in Petri nets 1990*, pages 313–341. Springer-Verlag New York, Inc. New York, NY, USA, 1991.

[HZDS95] C.X. Huang, B. Zhang, A.C. Deng, and B. Swirski. The design and implementation of PowerMill. In *Proceedings of the 1995 international symposium on Low power design*, page 110. ACM, 1995. 2.2

[IRF08] M.E.A. Ibrahim, M. Rupp, and H.A.H. Fahmy. Power Estimation Methodology for VLIW Digital Signal Processor. In *Proceedings of the Asilomar08 conference on signals, systems and computers, Asilomar, CA, US*, 2008.

[Jen92] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods, and practical use*. Springer, 1992.

[Jen97] K. Jensen. A Brief Introduction to Coloured Petri Nets. In *Tools and Algorithms for the Construction and Analysis of Systems: Third International Workshop, TACAS'97, Enschede, The Netherlands, April 2-4, 1997: Proceedings*. Springer, 1997.

[JK09] K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modeling and Validation of Concurrent Systems*. Springer-Verlag New York Inc, 2009. 1.1, 3.1, 3.1.2

[JML06] Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. Estimating the worst-case execution energy of embedded software. In *Proceedings of the Twelfth Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 81–90, 2006. 1

[JNM+06] M.N.O. Junior, S. Neto, P. Maciel, R. Lima, A. Ribeiro, R. Barreto, E. Tavares, and F. Braga. Analyzing Software Performance and Energy Consumption of Embedded Systems by Probabilistic Modeling: An Approach Based on Coloured Petri Nets. In *Petri Nets and Other Models of Concurrency (ICATPN 2006)*, volume 4024, pages 261–281. Springer, 2006. 2.1

[Jun98] Meuse Nogueira Oliveira Junior. Desenvolvimento de um protótipo para a medida não invasiva da saturação arterial de oxigênio em humanos - oxímetro de pulso. Master's thesis, Departamento de Biofísica e Radiobiologia, Universidade Federal de Pernambuco, Agosto 1998. (document), 6.1, 6.1

[keia] Keil software version 3.33. https://www.keil.com.

[Keib] Keil. Compilador gcc. https://www.keil.com/demo/eval/arm.htm. 1

[Kri05] Ravi Krishnan. Future of embedded systems technology. Technical report, BCC Research Group, June 2005. 1

[KS76] J.G. Kemeny and J.L. Snell. *Finite markov chains*. Springer, 1976.

[KTSN98] B. Klass, D.E. Thomas, H. Schmit, and D.F. Nagle. Modeling inter-instruction energy effects in a digital signal processor. In *Power-Driven Microarchitecture Workshop*, 1998. 2.1

[LK99] Averill M. Law and David M. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Higher Education, 1999.

[LM99] R. Leupers and P. Marwedel. Function inlining under code size constraints for embedded processors. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 253–256. IEEE Press, 1999. 6.3

[LSJM01] J. Laurent, E. Senn, N. Julien, and E. Martin. High Level Energy Estimation for DSP Systems. In *Proc. Int. Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, pages 311–316, 2001. 2.1

[LWD02] M. Lorenz, L. Wehmeyer, and T. Dr
"ager. Energy aware compilation for DSPs with SIMD instructions. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, pages 94–101. ACM New York, NY, USA, 2002. 2.1

[Mar06] Trevor Martin. *Introduction to the LPC2000*. Hitex (Uk) Ltd., 2006. 6.2

[MLC96] P.R.M. Maciel, R.D. Lins, and P.R.F. Cunha. *Introdução às Redes de Petri e Aplicações*. X Escola de Computação, Campinas, SP, 1996.

[MMC00] A. Malik, B. Moyer, and D. Cermak. A low power unified cache architecture providing power and performance flexibility (poster session). In *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 241–243. ACM New York, NY, USA, 2000. 6.1

[mod] Modeling of distributed and concurrent systems - modcs. http://www.modcs.org. 1.2

[MTMH97] R. Milner, M. Tofte, D. Macqueen, and R. Harper. *The definition of standard ML: revised*. The MIT Press, 1997. 3.1.2

[Mud01] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52–58, 2001. 2, 2

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. (document), 1.1, 3.1, 3.1

[Nag75] L.W. Nagel. *Spice 2: A computer program to stimulate semiconductor circuits*. University of California Berkeley, 1975. 2.2

[NKN+02] S. Nikolaidis, N. Kavvadias, P. Neofotistos, K. Kosmatopoulos, T. Laopoulos, and L. Bisdounis. Instrumentation Set-up for Instruction Level Power Modeling. In *Integrated circuit design: power and timing modeling, optimization and simulation: 12th International Workshop, PATMOS 2002, Seville, Spain, September 11-13, 2002: proceedings*, page 71. Springer Verlag, 2002. 2.1

[NMT+09] Bruno Nogueira, Paulo Maciel, Eduardo Tavares, Ermeson Carneiro, Gustavo Callou, Ricardo Massa, Rodolfo Ferraz, and Bruno Montenegro. Performance and Energy Consumption Evaluation of Embedded Applications: A Method Based on Platform's Behavioral Model. In *Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing*, pages 135–142. IEEE Computer Society, 2009. 1.1

[OJ06] M. Oliveira Júnior. *Estimativa do Consumo de Energia Devido ao Software: Uma abordagem em Redes de Petri Coloridas*. PhD thesis, Centro de Informática, Universidade Federal de Pernambuco, 2006. 2.1

[Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Rheinisch-Westfälisches Institut f. instrumentelle Mathematik an d. Univ., 1962. 3.1

[Phi04] Philips Electronics. *NXP LPC2104, LPC2105, LPC2106 Data Sheet*, 2004. 1.1, 4, 4.1, 4.5

[PKG02] D. Ponomarev, G. Kucuk, and K. Ghose. AccuPower: An accurate power estimation tool for superscalar microprocessors. In *Proceedings of Design, Automation and Test in Europe Conference*, 2002. 2.2

[Rib07] A. Ribeiro. Estimativa de consumo de energia de código ansi-c para sistemas embarcados: Uma abordagem baseada em simulação estocástica. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, 2007. 2.1

[RJ98] J.T. Russell and M.F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Proc. Int. Conf. Computer Design*, pages 328–333, 1998. 2.1

[RWL+03] A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Applications and theory of Petri Nets 2003: 24th international conference (ICATPN 2003)*, pages 450–462. Springer Verlag, 2003. 3.1.2

[SBN05] M. Schneider, H. Blume, and TG Noll. Power estimation on functional level for programmable processors. *Advances in Radio Science*, 2:215–219, 2005. (document), 2.1, 2.2

[SC01] A. Sinha and A.P. Chandrakasan. JouleTrack: a web based tool for software energy profiling. In *Proceedings of the 38th conference on Design automation*, pages 220–225. ACM New York, NY, USA, 2001. 2.1

[SLJM04] E. Senn, J. Laurent, N. Julien, and E. Martin. Algorithmic level power and energy optimization for DSP applications: SoftExplorer. In *IEEE International Symposium on Image/ Video Communications (ISIVC)*, 2004. 2.1

[Ste09] W.J. Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton Univ Pr, 2009. 3.2

[Tav09] Eduardo Antônio Guimarães Tavares. *Software Synthesis for Energy-Constrained Hard Real-Time Embedded Systems*. PhD thesis, Centro de Informática, Universidade Federal de Pernambuco, 2009. 1.2

[TL98] V. Tiwari and M.T.C. Lee. Power analysis of a 32-bit embedded microcontroller. *VLSI DESIGN-LANGHORNE-*, 7(3):225–242, 1998. 2.1

[TMS08] E. Tavares, P. Maciel, and B. Silva. Modeling hard real-time systems considering inter-task relations, dynamic voltage scaling and overheads. *Microprocessors and Microsystems*, 32(8):460–473, 2008. 4.5

[TMSO08] E. Tavares, P. Maciel, B. Silva, and M.N. Oliveira. Hard real-time tasks' scheduling considering voltage scaling, precedence and exclusion relations. *Information Processing Letters*, 2008. 1.2

[TMW94] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards softwarepower minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994. 2.1

[TMWTCL96] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee. Instruction level power analysis and optimization of software. *The Journal of VLSI Signal Processing*, 13(2):223–238, 1996. 2.1

[TP98] C.S.D.C.Y. Tsui and M. Pedram. Gate-level power estimation using tagged probabilistic simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1099–1107, 1998. 2.2

[Tri08] K.S. Trivedi. *Probability & Statistics with Reliability, Queuing and Computer Science Applications*. Wiley India Pvt. Ltd., 2008.

[VG02] F. Vahid and T. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, New York, USA, 2002. 1

[VKI+00] N. Vijaykrishnan, M. Kandemir, MJ Irwin, HS Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 95–106. ACM, 2000. 2.2

[Wel02] Lisa Wells. *Performance Analysis Using Coloured Petri nets*. PhD thesis, University of Aarhus, July 2002. 1.1, 4.4

[WK09] M. Westergaard and L.M. Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In *Proceedings of the 30th International Conference on Applications and Theory of Petri Nets*, page 322. Springer, 2009. 7

[ZHD+07] P. Zipf, H. Hinkelmann, L. Deng, M. Glesner, H. Blume, and TG Noll. A Power Estimation Model for an FPGA-Based Softcore Processor. In *International Conference on Field Programmable Logic and Applications, 2007 (FPL 2007)*, pages 171–176, 2007. 2.1

[Zim07] A. Zimmermann. *Stochastic discrete event systems: modeling, evaluation, applications*. Springer-Verlag New York Inc, 2007. 3

APÊNDICE A

CRITÉRIO DE PARADA

Durante a simulação, sempre que novas amostras são coletadas o critério de parada é testado. A função booleana criada para testar o critério de parada é exibida na Listagem A.1.

Esta função inicialmente verifica se o número de lotes atuais é maior que o número máximo de lotes, caso seja, a simulação para. O número máximo de lotes adotado foi de 10000. Portanto, mesmo que a simulação não atinja a precisão desejada, a simulação deve parar se o número de lotes for maior que 10000.

Em seguida, a função testa se a precisão desejada foi atingida para o consumo de energia e tempo de execução. Caso a precisão tenha sido atingida para ambas as métricas, a simulação para. Caso contrário, é calculado o número de novos lotes necessários para que tal critério seja atingido. Assim, a precisão relativa só é comparada novamente quando o número de novos lotes for igual ao número necessário de lotes calculado anteriormente.

Listagem A.1 Declarações da função que testa o critério de parada.

```
1
2  fun checkEnd() =
3  (
4  let
5    val ener=average(!energyList);
6    val tim=average(!timeList);
7    val energyError=standardError(!meanEnergyList);
8    val timeError=standardError(!meanTimeList);
9  in
10   if (!nReplicationExec>nMaxReplication) then(
11     writeFile("\nStopped by the max number of Replication\n");
12     true
13   )else
14   if (!nIterExec < nIter andalso !nReplicationExec<=(!nReplication))
15    then(
16      inc nIterExec;
17      false
18    )
19   else(
20      if (!nReplicationExec <= (!nReplication)) then(
21         (* DEBUG writeFile(Int.toString (!nReplicationExec) ^ "   " ^ Int.toString (!nReplication)); *)
22         addMeanData(ener,tim);
23         nIterExec:=0;
24         energyList:=nil;
25         timeList:=nil;
26         nReplicationExec:=(!nReplicationExec)+1;
27         false
28      )
29      else(
30         (*Checks if founded error is less than the desire error*)
31         if energyError< (average(!meanEnergyList) * relPreEner) andalso timeError<(average(!meanTimeList) * relPreTime)
   then
32           true(*Finish the simulation*)
33         else(
34           (*Calc the new number of replication needed*)
35             timeNewReplication:=relPreReplication(!meanTimeList, relPreTime);
36             energyNewReplication:=relPreReplication(!meanEnergyList, relPreEner);
37             writeFile("\n TimeRep: " ^ Int.toString(!timeNewReplication) ^
38                       "\n EnergyRep: " ^ Int.toString(!energyNewReplication) ^ "\n");
39             (*Choose the biggest number of replications*)
40             if (!energyNewReplication)>(!timeNewReplication) then
41               nReplication := !energyNewReplication
42             else
43               nReplication := !timeNewReplication;
44             energyList:=nil;
45             timeList:=nil;
46             nIterExec:=0;
47             false
```

```
48              )
49          )
50      )
51  end
52  );
```

APÊNDICE B

ALGORITMO DE GERAÇÃO DO CFG

O módulo de PECES que realiza a transformação do código Assembly em uma CFG depende repertório do conjunto da arquitetura alvo. Para este trabalho, o conjunto de instruções do LPC2106 foi utilizado, todavia, a mudança para outra arquitetura (família Intel x86, por exemplo) pode ser facilmente implementada, bastando apenas trocar as expressões regulares que identificam os tipos de instruções de cada classe de instrução.

Dado um código em assembly, a Figura B.1 exibe o algoritmo de geração dos blocos básicos que irão compor o CFG.

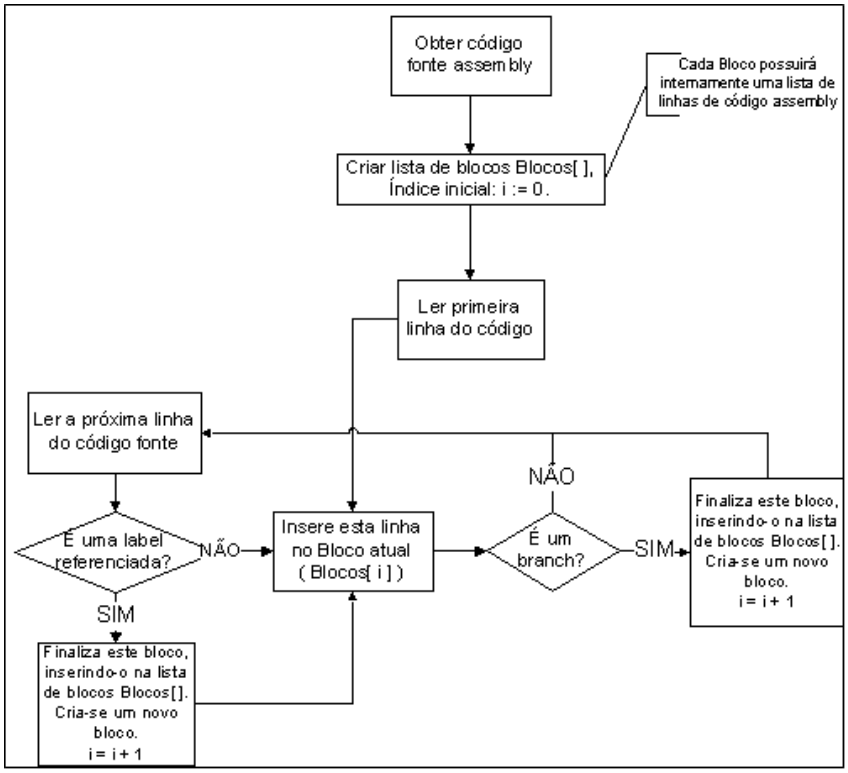

Figura B.1 Algoritmo de criação dos blocos básicos.

Depois de criar os blocos básicos, o próximo passo é criar as arestas do CFG. A Figura B.2 exibe o algoritmo de criação das arestas, que já são associadas a probabilidades (capturadas através das anotações no código fonte) para facilitar o processo de geração da DTMC.

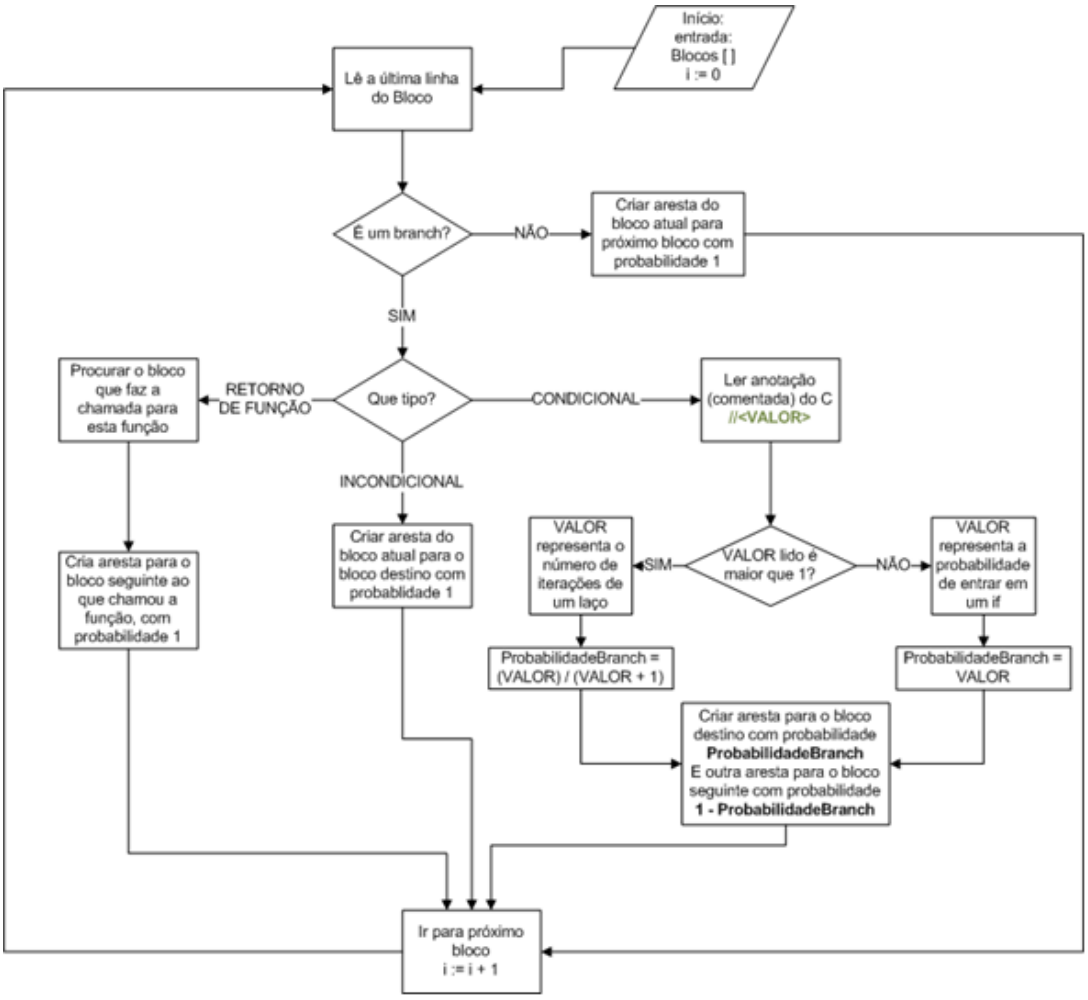

Figura B.2 Algoritmo de criação das arestas do CFG.