ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

UTILIZANDO TÉCNICAS DE COMPILAÇÃO PARA OTIMIZAÇÃO DA ANÁLISE DO CONSUMO DE ENERGIA

Trabalho de Conclusão de Curso

Engenharia da Computação

Emanuel de Aragão Pessôa
Orientador: Prof. Dr. Ricardo Massa Ferreira Lima

Recife, novembro de 2006

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

UTILIZANDO TÉCNICAS DE COMPILAÇÃO PARA OTIMIZAÇÃO DA ANÁLISE DO CONSUMO DE ENERGIA

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Emanuel de Aragão Pessôa
Orientador: Prof. Dr. Ricardo Massa Ferreira Lima

Recife, novembro de 2006

UNIVERSIDADE
DE PERNAMBUCO

Emanuel de Aragão Pessôa

UTILIZANDO TÉCNICAS DE COMPILAÇÃO PARA OTIMIZAÇÃO DA ANÁLISE DO CONSUMO DE ENERGIA

Resumo

A popularização dos dispositivos móveis portáteis (telefones celulares, tocadores de áudio e vídeo digital etc) tem motivado diversas pesquisas sobre a otimização do consumo de energia destes dispositivos. Mais do que a vantagem de economia financeira, economia de consumo de energia é uma grande vantagem competitiva já que proporciona maior autonomia aos dispositivos e maior aceitação de seus usuários.

Existem diversas ferramentas que permitem que os projetistas otimizem a arquitetura e implementação dos seus dispositivos de *hardware*, através de simulação dos seus circuitos. Entretanto, estudos demonstram que o consumo de energia é, também, afetado pela dinâmica do *software* em execução no dispositivo. Equipamentos que têm projeto de *hardware* e *software* integrados são mais eficientes de que os que são projetados para propósito geral.

Dentre as inúmeras técnicas para análise do consumo de energia nos *softwares*, há uma baseada em redes de Petri coloridas que apresenta resultados com um rigor formal adequado. A técnica consiste no mapeamento das instruções de um dado processador em um modelo CPN. Os programas submetidos à análise serão também convertidos em redes de Petri coloridas com base no modelo CPN de instruções do processador. A rede que representa o programa é analisa pelo *CPN-Tools* e gera informações que nos permitem criar um perfil de consumo de energia e apresentá-lo aos projetistas.

Oliveira Jr. propôs uma extensão desse modelo de forma a considerar que os diversos fluxos de execução do programa não podem ser analisados de forma igualitária. Ele propôs uma abordagem probabilística em que o projetista deve atribuir, no código assembly, probabilidade a cada instrução de desvio, descrevendo assim o cenário de execução em que a aplicação será analisada.

Este trabalho apresenta um estudo sobre o modelo proposto por Oliveira Jr e propõe uma extensão do mesmo, visto que naquele os projetistas devem aplicar anotações probabilísticas no código assembly e neste, a proposta é que tais anotações sejam feitas em código de linguagem C.

Abstract

The popularization of the portable mobile devices (cellular phones, digital audio and video players etc) motivated several researches on power consumption otimization for these devices. Despite the monetary economy, lower energy consumption is a great competitive advantage since provides more autonomy to the devices and more acceptance of its users.

There are some tools that allow designers to optimize the architecture and implementation of its hardware devices through simulation of its circuits. However, studies demonstrate that energy consumption is also affected by the dynamics of software execution in the device. Equipments that have integrated hardware and software project are more efficient of that ones that are projected for general purpose.

Amongst the techniques for energy consumption analysis in software, one is based in Colored Petri Nets and presents adequate formal results. The technique consists on mapping the processor instructions into a CPN-model. The programs submitted to the analysis will also be converted into Colored Petri Nets. The net that represents the program is analyzed by CPN-Tools and generates information that allow to create a consumption energy profile and to present it to the designers.

Oliveira Jr proposed an extension of this model in a way it considers that the several execution flows of the program cannot be analyzed as being the same. It considers a probabilist approach in that the designer must apply, in the assembly code, a probabilistic notation to each conditional instruction, thus describing the execution scenario where the application will be analyzed.

This work describes the model proposed by Oliveira Jr and presents an extension: in that one the designers must apply probabilistic notations in the assembly code and in our approach, the proposal is that such notations are made in C language.

Sumário

Índice de Figuras

Tabela de Símbolos e Siglas

PCAF - *Power Consumption Analysis Framework*
SDCC - *Small C Device Compiler*
HPC - H*igh-Performance Computing*
CPU - *Central Process Unity*
DV(F)S - *Dynamic Voltage (and Frequency) Scaling*
CMOS - *Complementary Metal-Oxide-Semiconductor*
CPN - *Colored Petri Nets*
CPN-ML - *Colored Petri-Nets Markup Language*
XML - *eXtensible Markup Language*
AST - *Abstract Syntatic Tree*
WCET - *Worst Case Execution Time*

Agradecimentos

Aos meus pais e irmãos por sempre terem dado todo o suporte para que eu chegasse até aqui. Agradeço também pela confiança em todos esses anos que estive fora de casa.

Aos amigos da Escola Politécnica que sempre me deram força e tornaram esses anos de convivência tão humanos. Um curso de Engenharia não seria possível sem vocês. Aos pais dos meus amigos da POLI pelas orações, pelos anseios e pelo acolhimento.

Aos meus amigos dos tempos de colégio por não deixarem nossa amizade desaparecer, mesmo nos momentos em que eu não fui cuidadoso com ela. A distância não consegue apagar momentos tão bons que passamos juntos nem diminuir o desejo de mais.

Ao Professor Ricardo Massa pelo apoio e compreensão nesse período, por ter sido tão ético e companheiro em alguns momentos difíceis da produção desse texto e pelo exemplo de profissional que é.

Ao Professor Meuse Oliveira pela co-orientação, não oficial, deste trabalho, pela inspiração e motivação de sempre.

Aos dedicados professores do nosso departamento pelo conhecimento repassado e pela inspiração. Uma grande parte do profissional que eu sou hoje se deve aos exemplos que vocês me deram.

Aos meus padrinhos Nancy Bezerra e Francisco de Assis, minha tia Sônia Amorim e meu amigo Vinícius Jucá por terem me acolhido em suas casas em momentos tão importantes para minha formação profissional.

Aos amigos da Rede Wireless por fazerem do nosso cotidiano, no trabalho, uma verdadeira escola. Eu tenho aprendido muito com vocês. Obrigado também pelo suporte nessa reta final do curso e por nunca terem me pressionado para colocar as atividades acadêmicas em segundo plano.

Aos demais amigos que acompanharam alguma etapa da produção desta monografia ou o decorrer do curso. A paciência e o apoio de vocês foi algo fora de série.

E a todos que passaram ou estão presentes em minha vida. Obrigado por terem contribuído para eu me tornar a pessoa que sou hoje.

Capítulo 1

Introdução

Motivados pela crescente demanda por dispositivos eletrônicos portáteis (aparelhos celulares, computadores de mão, tocadores de áudio e vídeo digital entre outros), os grandes fabricantes de *hardware* têm incentivado os estudos na área de consumo de energia, como forma de aumentar a autonomia de seus equipamentos. Outro fator importante é a necessidade de economia financeira causada por aparelhos que possuem consumo de energia menor. Além disso, o consumo de energia tem impacto direto no tempo de vida útil dos dispositivos.

Essa preocupação é claramente notada em dispositivos eletrônicos com alta popularidade, como o *iPod* geração 6, da *Apple Computers* [1], que teve a duração da sua bateria incrementada de 20 para 24 horas, em relação ao seu antecessor. A *Intel* [2] também tem se mostrado bastante atenta às necessidades de baixo consumo de energia. O *Intel Pentium M* é provido da tecnologia *SpeedStep*, que controla o consumo de energia do processador, habilitando um consumo maior para aplicações que precisam de maior capacidade de processamento e diminuindo-o quando não necessário.

As novas tendências de tecnologia para o desenvolvimento de sistemas portáteis têm necessitado de estratégias para diminuir o consumo de energia. Para alcançar os melhores resultados, os projetistas dispõem de ferramentas que auxiliam na otimização de suas alternativas sem a necessidade de implementação física do dispositivo e simulações de seus circuitos. Essas ferramentas permitem alcançar vários níveis de complexidade e exatidão.

Mesmo com otimizações de *hardware*, o impacto causado pela execução de *software* é também crucial, visto que o consumo de energia é afetado pela dinâmica do comportamento do programa em execução [3]. Então, a análise da estrutura de dados gerada pelo compilador e seus impactos são importantes para mensurar o consumo de energia em um dispositivo, como está abordado em [4].

Uma das técnicas utilizadas para estudar o comportamento de consumo de energia é através do mapeamento das instruções primitivas do processador, que são executadas pelo *software*, em redes de Petri coloridas [5]. Cada instrução do processador teria uma rede de Petri colorida associada a ela, assim, após o processo de compilação, teremos uma rede complexa correspondente ao programa em análise. A análise da rede de Petri é realizada pelo PCAF (*Power Consumption Analysis Framework*) [6] e fornece ao projetista informações valiosas sobre trechos críticos da implementação, no que diz respeito ao consumo de energia. Através dessas análises, é

possível, ainda, gerar uma estatística sobre os valores médios da energia consumida em cada cenário de execução.

O modelo proposto por Oliveira Jr. [6] apresenta uma evolução dessa técnica, ao considerar que os diversos fluxos de execução do programa não podem ser analisados de forma equalitária, pois cada cenário de execução gera um perfil de consumo diferente. Esse modelo é uma abordagem probabilística que deve ser aplicada pelo projetista a cada instrução de desvio, descrevendo assim seu cenário de execução. Este modelo foi desenvolvido para ajudar desenvolvedores de código de máquina.

O presente trabalho apresenta uma extensão para o modelo descrito no parágrafo anterior, de forma a auxiliar também os projetistas de código de alto nível.

1.1 Objetivos do Trabalho

Nosso projeto possui as seguintes metas a serem alcançadas:

- Explorar a técnica de análise probabilística do consumo de energia. Essa técnica cria perfis de consumo de energia baseado na ocorrência de específicos cenários de execução, eliminando a necessidade de realização de testes com valores de inicialização diversos.
- Definir e implementar extensão para o modelo de análise proposto por Oliveira Jr., para que este possa ser utilizado por projetistas que desenvolvem seus sistemas em linguagem de alto nível (neste caso, linguagem C).
- Criar um ambiente integrado para análise de consumo de energia.

1.2 Metodologia

Nosso grande desafio é encontrar uma maneira fácil e eficiente para estender o modelo proposto por Oliveira Jr. de forma a considerar a visão do projetista de sistemas embarcados – propriamente de sistemas desenvolvidos em linguagem C – sobre o fluxo de execução do programa. Essa abordagem, que considera informações úteis informadas pelo projetista, gera um perfil de consumo de energia[1] a partir do qual pode-se optar por uma melhor distribuição de *hardware/ software* nos segmentos de código crítico do sistema, tais como: a re-implementação de um trecho de código, a implementação da funcionalidade crítica em *hardware*, entre outras.

Para estender o trabalho de Oliveira Jr. de forma a contemplar a análise de código em linguagem C, procederemos da seguinte forma:

- **Estudo do mapeamento de código C para código assembly –** Como iremos adotar a mesma abordagem de anotações probabilísticas para cada instrução de desvio condicional, deveremos entender o mapeamento de código C para o código assembly realizado pelo compilador adotado, o SDCC (*Small C Device Compiler*) [7] – já que este mapeamento é um pouco diferente de compilador para compilador – para que possamos inserir anotações no código assembly a partir das anotações informadas pelo projetista no código de alto nível.

[1] Perfil de consumo de energia deve ser entendido como informações, em forma de tabelas e gráficos, que permitem que o projetista analise o consumo do programa nos mais diversos cenários de execução.

- **Criação de um *parser* para validação do arquivo gerado pelo SDCC –** Criaremos um *parser* que irá ler as informações de cabeçalho do arquivo C e adicioná-las ao arquivo assembly. Nosso *parser* será também responsável pela inserção de anotações probabilísticas padrões ao lado das instruções de desvio condicionais que não foram mapeadas pelo SDCC.
- **Validação do Modelo** – Validaremos nosso modelo através de comparação com os resultados apresentados no trabalho de Oliveira Jr. Nossa preocupação será obter perfis de consumo equivalentes aos obtidos naquele trabalho, utilizando a nossa extensão.

1.3 Organização do Trabalho

O presente trabalho está estruturado com os seguintes capítulos:

- **Capítulo 2 –** Neste capítulo apresentamos o cenário atual de desenvolvimento de dispositivos eletrônicos, os obstáculos enfrentados por esse setor e as soluções que já vêm sendo largamente utilizadas. Nosso foco serão os dispositivos portáteis, que são o objeto de estudo desse trabalho, mas as considerações apresentadas poderão ser estendidas para as demais áreas. Esse capítulo aumenta a nossa motivação nos estudos para redução de consumo de energia em dispositivos móveis.
- **Capítulo 3 –** Como o foco do nosso trabalho é otimização da análise do consumo de energia através de *softwares*, neste capítulo apresentamos diversas ferramentas úteis na compreensão e desenvolvimento de ferramentas que satisfaçam os nossos objetivos.
- **Capitulo 4 –** Mostramos, além das nossas implementações, um estudo completo do ambiente de análise proposto por Oliveira Jr. e estendido aqui. Nesse capítulo explicaremos cada etapa do processo de análise, desde a preparação do código-fonte da aplicação a ser estudada até à geração de resultados, focando nas entradas disponibilizadas e insumos produzidos em cada uma dessas fases.
- **Capítulo 5 –** Utilizamos este capítulo para validar o nosso modelo, através de um estudo de caso aplicado a um conhecido algoritmo de ordenação.
- **Capítulo 6 –** Neste capítulo apresentamos as conclusões e as contribuições deste trabalho; bem como os trabalhos relacionados e trabalhos que serão desenvolvidos no futuro.

Capítulo 2

Consumo de Energia e Linhas de *Hardware* Emergentes

Por muito tempo, a comunidade da área de computação de alto desempenho (HPC – *high-performance computing*) encarou desempenho como sinônimo de velocidade. O fundador da Intel, Gordon Moore, constatou que a capacidade de processamento dos computadores dobra a cada período de 18 a 24 meses (conjectura de Moore), mantendo os custos constantes [8]. Isso quer dizer que dentro de, aproximadamente, dois anos, um *chip* com o dobro do processamento será comprado pelo mesmo preço que é encontrado nas prateleiras hoje em dia.

Mas, junto com o número de transistores, aumentava também o consumo de energia deste *chip*. À energia dissipada pelo processamento do *chip*, se junta a energia consumida pelo resfriamento contínuo que se faz necessário, o que aumenta substancialmente o custo de manutenção de um sistema de computação em funcionamento.

A lei de Moore está em vigor há mais de 30 anos e a maioria dos pesquisadores acredita que deve durar pelo menos mais cinco gerações de processadores (e outros aspectos da tecnologia digital como chips de celulares, discos rígidos e até velocidade de conexão com a Internet).

De qualquer forma, já podemos observar algumas mudanças nos dispositivos computacionais, motivadas em parte pela crescente distribuição de dados e serviços, proporcionada pela Internet e, em parte, pela conseqüente necessidade de dispositivos portáteis para atender a tal demanda. Acompanhando tal revolução, tem-se focado mais na vida útil e tempo de operação destes dispositivos do que em seu alto poder de processamento.

Os computadores modernos de uso geral tiveram seu poder de processamento aumentado de tal forma que subutilizamos esse recurso. De fato, para a simples atividade de navegação na *web*, execução de músicas e digitação desse texto, é necessário menos de 10% do poder de processamento total do computador (já contando o processamento gasto para o funcionamento do sistema operacional), conforme o gerenciador de tarefas do sistema operacional. Isso não seria nenhum problema se energia fosse um recurso abundante e barato.

Nesse novo cenário, não é incomum que as maiores empresas de *hardware* estejam desenvolvendo micro-processadores especializados em determinada tarefa, com poder de processamento menor que um processador tradicional, mas que execute uma mesma atividade de forma tão satisfatória quanto o outro. Há ainda diversas pesquisas na área de *super-threading* (ou *hyper-threading*, em termos da Intel), que permite a uma CPU (*Central Process Unity*) processar

várias *threads* paralelas simultaneamente, alterando de uma para outra sempre que um processo entrar em estado de espera.

Como motivação para o restante do trabalho, este capítulo apresenta, de forma sucinta, duas novas tendências no desenvolvimento de *hardware* para dispositivos portáteis. Nosso objetivo é mostrar que as empresas estão investindo em pesquisas para diminuir o consumo de energia de seus dispositivos e que nossa abordagem – otimizar a análise do consumo de energia através de *software* – é pertinente em meio às mudanças.

2.1 SoC – *System on a Chip*

Uma alternativa à perspectiva de se ter bilhões de transistores num único processador vem da necessidade de integração das funções que estão fora do *chip*, reduzindo o tamanho, custo e consumo de energia do sistema como um todo. Há também o fato de que a redução dos custos de comunicação entre os diversos elementos do sistema aumenta o desempenho do sistema como um todo comparado com o ganho obtido apenas com o aumento de processamento.

A tecnologia *System on a Chip* consiste no agrupamento de todos (ou grande parte dos) circuitos eletrônicos necessários para a construção de um sistema num único circuito integrado. Por exemplo, um SoC para um dispositivo de detecção de ruído poderia incluir um receptor de áudio, um conversor de áudio analógico para digital, um microprocessador, memória e um controlador da lógica de entrada e saída de dados – tudo num único *chip*.

A integração de diversas funções é bem comum em dispositivos portáteis, onde o consumo de energia e seus impactos são características essenciais. Hoje, já conseguimos encontrar dispositivos que contenham uma variedade de funções, incluindo suporte para funções gráficas, de vídeo, de áudio, de disco rígido e rede, por exemplo. Os processadores de alto desempenho não são utilizados porque sua capacidade é diminuída quando colocado junto com dispositivos que possuem outra tecnologia no mesmo *die*[2], e seu uso seria apenas um fator de elevação dos custos de fabricação. A integração de diversos tipos de componentes requer então a habilidade de pôr junto tecnologias diferentes sem causar queda no desempenho, além de garantir que a organização destes leve a um alto desempenho e baixo consumo de energia. A Figura 1 mostra um controlador embarcado desenhando baseado na arquitetura *IBM CoreConnect* usando a tecnologia *Blue Logic* [9]. Mesmo tendo um processador com desempenho menor que os processadores de propósito geral, o sistema mostrado na figura tem um melhor desempenho por unidade de área e por *watt* dissipado.

[2] *Die* se refere à parte do material semicondutor no qual um dado circuito será fabricado.

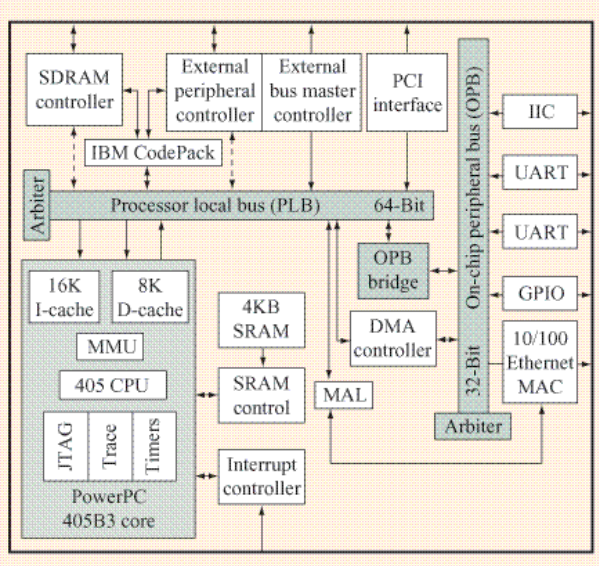

Figura 1. Diagrama de blocos de um sistema embarcado com várias funções integradas no mesmo chip.

Apesar das dificuldades encontradas, é provável que o número de projetos utilizando esse tipo de integração continue a crescer. E enquanto for atrativo, financeiramente, pôr funções sofisticadas juntas na mesma placa, a maioria das aplicações continuará tirando vantagem da redução de tamanho e consumo de energia. Além disso, os avanços oferecidos por SoC em termos de custo, área, consumo de energia e modularidade poderá gerar um grande reuso de componentes, inclusive em processadores de propósito geral.

2.2 DV(F)S – *Dynamic Voltage (and Frequency) Scaling*

Outra técnica já bastante utilizada por computadores portáteis é a DV(F)S (*Dynamic Voltage - and Frequency - Scaling)* [10], que consiste na exploração das características dos *hardwares* dos processadores visando à redução do consumo de energia, através do controle de voltagem e freqüência do processador.

DVS tenta alcançar um equilíbrio entre desempenho e tempo de vida da bateria levando em consideração duas características importantes da maioria dos sistemas computacionais: (1) a taxa de processamento disponível é muito maior que a média do processamento necessário pela maioria das aplicações; e (2) os processadores são baseados em lógica CMOS *(Complementary Metal-Oxide-Semiconductor).* A primeira característica significa efetivamente que um alto desempenho é necessário apenas em uma pequena fração do tempo, enquanto no restante do tempo, para atividade de baixo desempenho, um processador de baixo consumo seria suficiente. Nós podemos alcançar o baixo desempenho simplesmente diminuindo a freqüência de operação do processador quando a velocidade total não for necessária. DVS vai além disso e também associa escalas de voltagem de operação adequadas a cada freqüência. Isto é possível graças à lógica CMOS, usada na maioria dos micro-processadores atualmente, que tem dependência de uma voltagem máxima de operação por freqüência, então quando a freqüência é reduzida, o processador pode operar com a voltagem menor oferecida.

Ao contrário do que acontece com os dispositivos móveis, a noção de economia do consumo de energia é nova para a comunidade de HPC. Há duas razões claras para isso. Primeiro, as características computacionais encontradas nos sistemas embarcados e sistemas móveis diferem notadamente daquelas encontradas em dispositivos com alto poder computacional. Como resultado, os algoritmos para economia de energia que funcionam bem para uso interativo em *laptops*, por exemplo, podem falhar quando se trata de aplicações científicas [11]. Segundo, consumo de energia é necessário por razões distintas. Em sistemas embarcados e móveis, a economia no consumo de energia é necessária para aumentar o tempo de vida útil da bateria, enquanto nos sistemas de alto desempenho, é necessária para reduzir os custos operacionais com energia e resfriamento, bem como forma de aumentar a confiabilidade dos sistemas.

2.3 Resumo

As seções anteriores nos mostram que estamos muito próximo do retorno da computação com propósitos específicos (é claro, que o cenário em que esse retorno acontecerá é outro completamente diferente daquele de décadas atrás). De fato, um dispositivo portátil com código otimizado, por exemplo, pode executar uma função igualmente bem a um computador pessoal moderno, possuindo um processador muito mais lento. Por exemplo, um tocador de *mp3* pode executar sua atividade melhor do que uma CPU de propósito geral, com grande poder de processamento.

Pesquisadores acreditam que dentro de alguns anos o aumento de freqüência das CPU's será bem modesto e que os dispositivos com processamento especializado tomarão conta do mercado. Os atuais processadores de uso geral se tornarão, então, coordenadores das atividades dos pequenos processadores de propósito específico.

A área de desenvolvimento de *software* sofrerá um impacto radical para atender as demandas por baixo consumo de energia. Em vez de um software que controlará todo o hardware, sem preocupação com o custo operacional disto, teremos softwares especializados em cada dispositivo de baixo consumo.

Nos próximos capítulos apresentaremos uma técnica de análise de consumo de energia que será aplicada aos nossos *softwares*.

Capítulo 3

Softwares para otimização do Consumo de Energia: Fundamentação

No capítulo anterior analisamos diversas formas de aumento de desempenho dos processadores modernos levando em consideração o consumo de energia como um fator crucial para a sua aceitação comercial.

As linhas de pesquisa para otimização de consumo de energia, quase sempre, seguem a linha de pesquisa em *hardware* ou em *software*, sendo poucas as que consideram esses dois universos. De fato, as pesquisas em *hardware* consideram sempre os piores casos, em que o código não é otimizado e totalmente dependente, e tenta lidar com essas restrições para propor modelos mais eficientes. Do outro lado, as pesquisas em *software* lidam com o fluxo de execução dos programas e características de processadores de propósito geral, normalmente determinando trechos de implementação críticos e deixando na mão do desenvolvedor/ projetista a solução a ser adotada. As Figuras 2 e 3 fazem uma comparação entre o tempo de execução de uma ação num programa em dois sistemas operacionais diferentes, rodando numa mesma máquina. Os testes foram realizados num MacBook Pro e os sistemas operacionais utilizados foram o Windows XP Pro e o Mac OS. É valido notar que o Mac Os se sai melhor nos testes principalmente por ser um sistema operacional projetado para o *hardware* em uso.

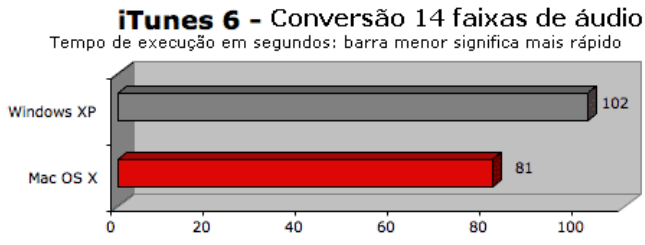

Figura 2. Tempo de conversão de uma faixa de aúdio no programa iTunes, no Windows Xp e no Mac OS.

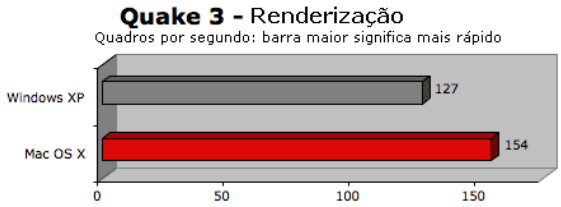

Figura 3. Comparação do desempenho no Windows Xp e no Mac OS da execução do jogo Quake.

De fato, a produção de *software* e *hardware* sob demanda casa perfeitamente com nossos objetivos de alto desempenho e baixo consumo de energia. Entretanto não podemos esquecer que o mercado é muito heterogêneo e sua maior parte é composta por empresas especializadas em determinada atividade (não é comum encontrar empresas especializadas em desenvolvimento de *software* para MacBooks, por exemplo). O grande desafio é a criação de *hardware/ software* de propósito geral e que atendam aos objetivos de desempenho e consumo almejados.

Segundo [3], o impacto causado pela execução de um *software* é também crucial num cenário que visa ao menor consumo de energia, já que este é afetado pela dinâmica do comportamento do programa em execução. Então, a análise da estrutura de dados gerada pelo compilador e seus impactos são importantes para mensurar o consumo de energia em um dispositivo.

Este capítulo apresenta uma fundamentação teórica em redes de Petri (que são a base das simulações utilizadas para mensuração do consumo de energia), bem como algumas ferramentas que as utilizam para realizar análises do consumo. A seção 3.4 apresenta a ferramenta gerada a partir dos resultados apresentados em [6] que faz uso de um modelo probabilístico para realizar análise de consumo de energia em códigos assembly.

3.1 Redes de Petri

Essa seção é baseada em conteúdo encontrado em [5].

As Redes de Petri, propostas por C. A. Petri em 1962, são representações matemáticas de sistemas que apresentam ferramentas de análises e corretude poderosas. Além disso, as Redes de Petri têm ganhando bastante espaço por possuir uma visualização gráfica, como a apresentada na Figura 4. As Redes de Petri são formadas por dois componentes básicos: transições e lugares. Os lugares correspondem às variáveis de estado e as transições às ações realizadas pelo sistema.

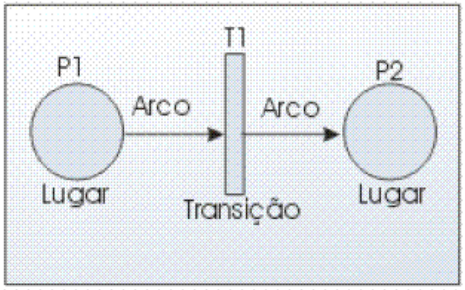

Figura 4. Modelo de uma Rede de Petri simples e alguns de seus componentes.

Numa rede de Petri, os lugares são representados pelos círculos e representam o estado do sistema através de marcas denominadas *tokens* (e representadas por um círculo preto dentro do lugar). A presença dos *tokens* num lugar representa pré-condição para que uma transição seja disparada. A transição, representada por retângulos, caracterizam o dinamismo da rede; quando certas pré-condições são satisfeitas (existe um número de *tokens* necessário no lugar correspondente) as transições vão sendo disparadas. O disparo das transições é responsável pela criação e destruição de recursos/ marcas (*tokens*) na rede.

Um terceiro elemento que compõe as Redes de Petri são os arcos, responsáveis por ligar lugares a transições e vice-versa. Temos a associação de um peso a cada arco (grau do arco) de forma a indicar quantas marcas devem ser consumidas para que o sistema execute uma transição, mudando seu estado. De forma geral, a mudança de posição das marcas caracteriza a mudança de estado da rede através de um disparo. Vale salientar que, ao habilitar uma transição, esta não vai obrigatoriamente executar. Isto é o que dá o caráter não determinístico das redes.

A habilitação de uma transição se dá quando os lugares de entrada têm um número de marcas maior ou igual aos pesos dos arcos de saída, como indicado na Figura 5(a). Caso a transição seja disparada, será adicionado aos lugares de saída um número de marcas igual aos pesos dos arcos de saída (da transição), o que pode ser observado na Figura 5(b).

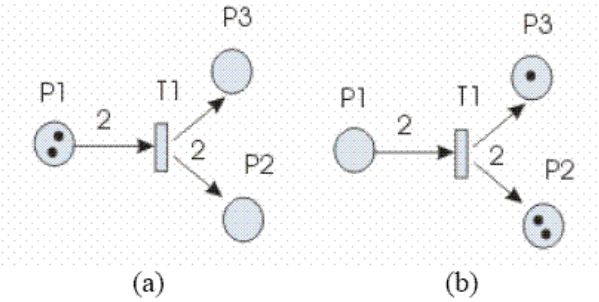

Figura 5. Habilitação e disparo de uma transição

Matematicamente, uma Rede de Petri é definida por uma quíntupla (P, T, I, O, M0) onde:

- P = {p1, p2, p3, ...,pm} é o conjunto dos lugares da rede, sendo que ‘m’ representa o identificador do lugar da rede;
- T = {t1, t2, t3, ... ,tn} é o conjunto das transições da rede. Da mesma forma, ‘n’ representa o identificador da transição;
- I:(P,T) é a função que define o número de arcos de entrada das transições. Por exemplo, para dizer que existem k arcos saindo do lugar pm e chegando a transição tn poderíamos representar desta forma: I(pm, tn) = k;
- O:(T,P) por outro lado, representa o conjunto de arcos que saem das transições e chegam nos lugares da rede.
- M0 é a marcação inicial da rede, definida pela distribuição das marcas nos lugares. M = (M(p1), M(p2), M(p3), ..., M(pj)), onde M(pj) define a quantidade de marcas no lugar pj, sendo que pj Î P.

3.1.1 Análise de Redes de Petri

Redes de Petri são ferramentas matemáticas e gráficas capazes de modelar processos, executando paralelamente em um sistema discreto, representando com clareza relações de causa e efeito entre esses. Uma grande vantagem das redes de Petri é a possibilidade de realizar tanto análises matemáticas, para a verificação de propriedades do sistema modelado, quanto simulações [5].

A seguir são apresentadas algumas características desse tipo de rede que permitem tais análises:

Alcançabilidade (*reachability*): possibilidade de se atingir determinada marcação através da execução de finitas transições, a partir de uma marcação inicial. Em sistemas computacionais, essa característica é importante para se determinar se estados indesejáveis podem ser executados. Na Figura 6 vemos que a partir de uma marcação inicial M(0,1,0,0) podemos alcançar a marcação M(1,0,0,0), mas não podemos alcançar M(0,0,0,0), por exemplo.

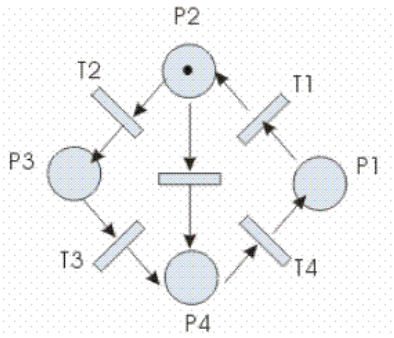

Figura 6. Análise de Alcançabilidade

Limitação (*boundedness*): serve para verificar se existem lugares na rede que acumulam marcas infinitamente. Em termos práticos, poderíamos utilizar esta característica para verificar se a capacidade de um *buffer* num sistema de comunicação cliente-servidor é suficiente. Determina-se a capacidade de acumular marcas num determinado lugar usando a notação n-limitada, onde n é o número máximo de marcas naquele lugar. Na Figura 7, os lugares P1, P3 e P4 são 1-limitados.

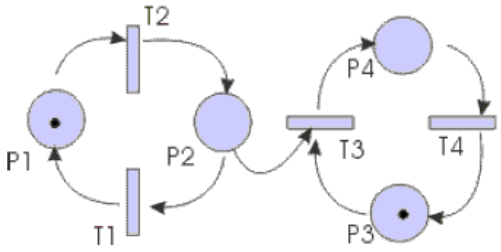

Figura 7. Análise de Limitação.

Segurança (*safeness*): análise especial de limitação em que todos os lugares da rede são 1-limitado, independente do estado em que a rede se encontra.

***Liveness*:** serve para checar se todas as transições presentes na rede podem ser disparadas em algum estado específico. Quando uma transição não pode ser disparada, dizemos que ela está

"morta". Fazendo uma analogia com programas de computadores, uma transição morta corresponde a uma condição, no trecho de código, que nunca será executada e, portanto, está apenas representando um "lixo" dentro do código. Na Figura 8(a) podemos garantir que todas as transições são *live* enquanto na Figura 8(b), a transição T0 é uma transição "morta".

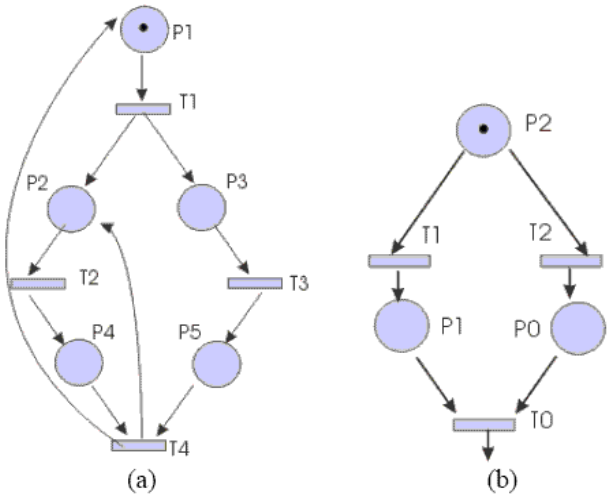

Figura 8. (a) Rede com transições *live*; (b) Rede com transições "mortas".

Persistência: essa propriedade verifica se, para um par de transições habilitadas, o disparo de uma transição não desabilita a outra. Em sistemas computacionais, uma aplicação dessa análise seria a busca de efeitos colaterais causados pela execução de alguma ação.

Reversibilidade: averigua se é possível através de uma dada marcação, retornar a uma marcação específica através de um número finito de transições. A rede apresentada na Figura 9 é reversível, pois todas as marcações são acessíveis a partir de outras.

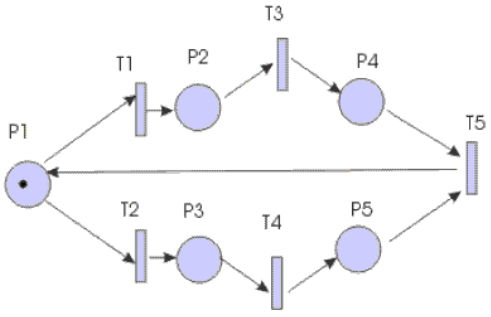

Figura 9. Rede Reversível

Justiça (*fairness*): o conceito de justiça está ligado ao fato de uma transição não ficar executando sozinha por tempo indeterminado. A justiça garante que cada transição irá executar um número limitado de vezes, e outras transições irão poder executar também. Nos sistemas computacionais, encontramos justiça nas operações de leitura/ escrita de dados na memória pro vários processos, por exemplo. A rede na Figura 10 apresenta-nos um caso de justiça onde há relação de um para um no disparo das transições.

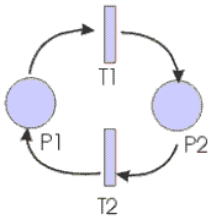

Figura 10. Análise da Justiça

3.1.2 Redes de Petri Coloridas

As Redes de Petri Coloridas surgiram da necessidade de se modelar grandes sistemas, que era uma atividade muito complexa utilizando Redes de Petri tradicionais. A idéia por traz das redes de Petri coloridas é unir a capacidade de representar sincronização e concorrência das redes de Petri com o poder expressivo das linguagens de programação. Através dessa união, sistemas cujo estudo anteriormente era impraticável tornaram-se passíveis de estudo.

Rede de Petri Colorida é um modelo de mais alto nível que considera tipos de dados abstratos e hierarquia. Esse modelo de rede é utilizado por três razões relacionadas. Primeiro, um modelo CPN (*Colored Petri Nets*) é uma representação formal de um sistema. Criando um modelo, pode-se investigar o sistema antes de construí-lo, o que é uma vantagem óbvia, especialmente em sistemas onde erros de modelagem podem gerar problemas de segurança e/ ou demandar muito tempo para correção. Segundo, o comportamento de um modelo CPN pode ser analisado através de simulação, o que corresponde à execução e depuração de um programa, ou através de métodos mais formais de análises, o que representa a verificação de programas. Finalmente, pode-se admitir que o processo de criação de uma descrição e execução de análises de um modelo aumentam drasticamente o entendimento do sistema modelado.

A principal diferença das CPN para as redes tradicionais é o fato de que os *tokens* que marcam os lugares possuem um valor associado, deixando de ser apenas uma pré-condição para que uma transição aconteça. Estes valores são então utilizados em expressões que são calculadas durante a avaliação de transições, bem como durante o disparo destas.

As CPN são compostas por três elementos, que podem ser visualizados na Figura 11:

- *estrutura*: corresponde à estrutura de lugares, transições e arcos de uma rede de Petri ordinária. A diferença aqui é que cada lugar possui um tipo de dado associado. O conjunto de valores que esse dado pode assumir é chamado de conjunto de cores
- *declarações*: tipos e variáveis utilizadas na rede.
- *inscrições*: anotações associadas aos elementos da rede.

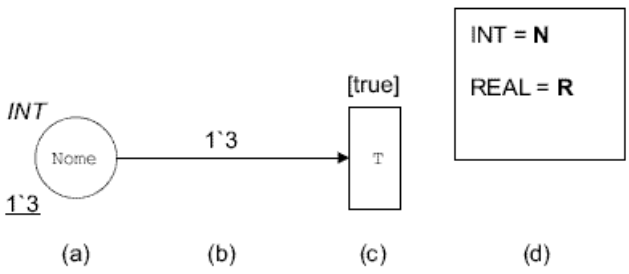

Figura 11. Exemplo de uma Rede de Petri Colorida

3.2 *CPN-Tools*

CPN-Tools [12] é uma ferramenta de alto nível para edição, análise e simulação de redes de Petri Coloridas. Ela implementa praticamente todas as funcionalidades contidas na especificação das CPN junto com algumas melhorias, principalmente na interface da aplicação.

Outras facilidades encontradas no *CPN-Tools* dizem respeito a mensagens indicativas de erros e de dependência entre os elementos de uma rede, por exemplo. A ferramenta conta também com um analisador de sintaxe e geração de código que é ativado quando uma rede está sendo criada. Um simulador mais rápido e eficiente também foi integrado, permitindo a análise de redes temporizadas ou não. É possível também gerar e analisar estados intermediários completos e parciais, bem como um estado padrão; a partir desses estados pode-se analisar propriedades como de alcançabilidade.

A ferramenta *CPN-Tools* foi construída em cima de três princípios de projeto:

- *reification*, que significa que todos os objetos (mesmo os abstratos) serão tratados como entidades físicas reais;
- polimorfismo, o que garante que comandos sejam aplicados a tipos pertencentes a uma mesma classe;
- reuso, oferece a possibilidade de reutilizar instâncias previamente criadas em outros módulos.

A Figura 12 mostra a interface do *CPN-Tools*. O exemplo do modelo descreve como dois tipos diferentes de processos estão compartilhando três tipos diferentes de recursos.

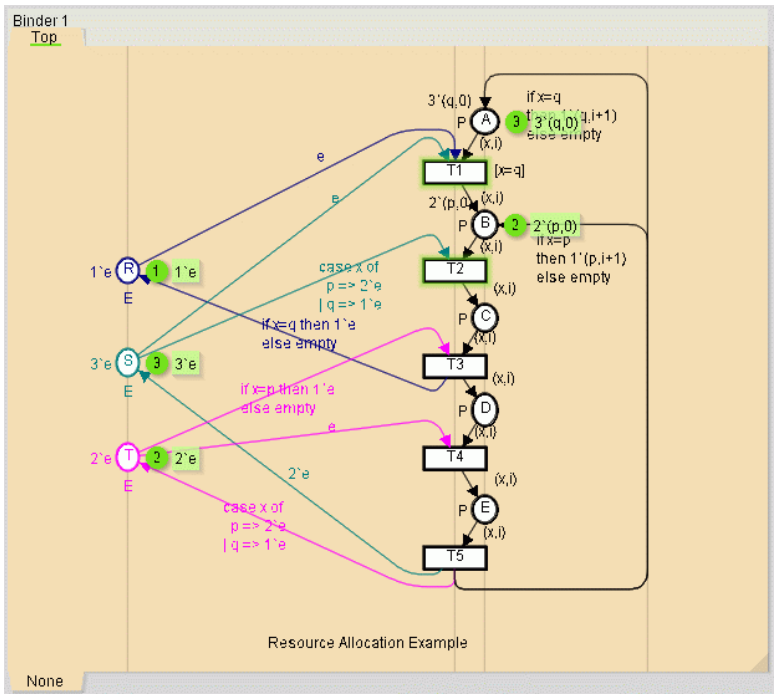

Figura 12. Exemplo de Rede de Petri Colorida no *CPN-Tools*.

3.3 PCAF (*Power Cost Analysis Framework*)

Foi proposto em [13] uma nova abordagem para análise de consumo de energia de *software*. Essa abordagem explora as redes de Petri coloridas como uma linguagem de descrição formal. Através de simulação é possível obter padrões comportamentais que caracterizam determinados consumos de energia.

O modelo consiste na associação de uma CPN a cada instrução pertencente ao conjunto de instruções do processador. Então, cada programa pode ser modelado como uma CPN maior, onde cada estado do processador é modelado como um lugar, cada contexto do programa como uma estrutura de dados com um *token* e as instruções como transições que consomem esses *tokens*. O comportamento da instrução é determinado por um código CPN-ML (*Colored Petri-Nets Markup Language*) associado ao analisador.

Essa nova abordagem modela os processadores apenas através de seu conjunto de instruções. Modelos baseados em suas unidades funcionais teriam que ser providos de uma série de informações detalhadas sobre o consumo do *hardware* e das tecnologias associadas. Essas informações não estão disponíveis freqüentemente para os projetistas. Já os modelos baseados em instruções podem ser facilmente implementados através de dados coletados por simples medições físicas [14].

O PCAF é, então, um arcabouço que permite integrar, através do EZPetri [15], o modelo definido acima com as ferramentas de simulação apropriadas, no caso o *CPN-Tools*. Ele implementa funções responsáveis pela identificação de trechos de consumo de energia críticos no código do programa permitindo análises consistentes.

O PCAF é constituído de três módulos de integração principais, ilustrados na Figura 13:

- compilador binário-CPN: responsável pela transformação do código assembly na Rede de Petri Colorida que representa o programa. O compilador espera duas entradas: o código de máquina e o modelo de instruções CPN. O modelo de instruções CPN é o mapeamento da arquitetura do processador em redes de Petri Coloridas correspondentes. A partir desse modelo, o compilador binário-CPN cria o modelo correspondente ao código de máquina fornecido. O modelo é gerado num formato XML apropriado aceito pelo *CPN-Tools*.
- CPN-Tools: executa as funções de análise e oferece como retorno perfis de consumo do programa, *patches* de consumo e *clusters* de consumo, que serão melhor explicados no capítulo referente à análise de resultados.
- Ambiente "*hidden-tools*": implementa funções que identificam os *patches* e *clusters* criados, bem como os perfis de consumo e apresenta a informação para o usuário em forma de gráficos e tabelas, dentro do ambiente do Eclipse [16].

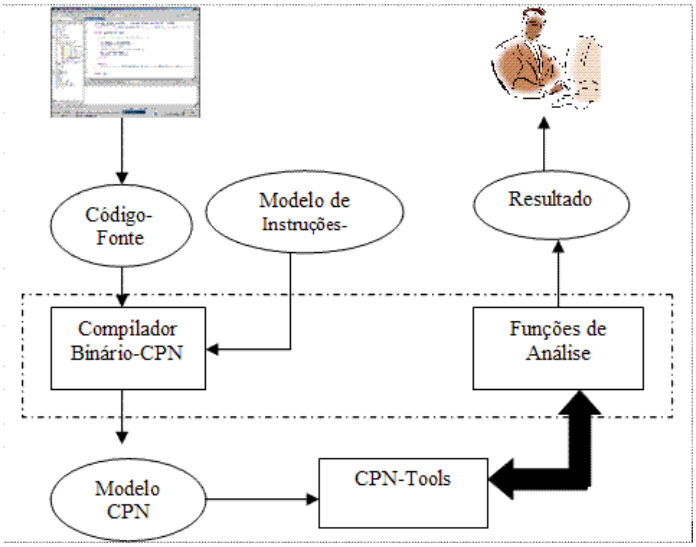

Figura 13. Arquitetura do PCAF

3.4 *CPN-Probabilistic Modeling*

A ferramenta *CPN-Probabilistic Modeling* é uma extensão do PCAF e implementa às funções necessárias à análise baseada em cenários de execução (determinados através de valores de probabilidades de execução) propostas em [6] e estendida neste trabalho.

Descrevemos aqui os principais módulos que compõem a ferramenta:

- editor de código: onde poderão ser escritos ou modificados (anotados com respectivas probabilidades) os códigos em linguagem assembly.
- *assembler*: compilador de código assembly. A compilação gera, além do arquivo binário, um arquivo .LST que contém informações sobre as posições de memória de cada instrução no código. Este arquivo é importante para nós, pois através dele conseguimos obter as posições de memória das instruções de desvio condicional que foram anotadas.
- *LST-Parser*: tradutor de arquivos .LST em uma estrutura de arquivo .XML (contendo as anotações probabilísticas) que será utilizada pelo compilador binário-CPN modificado. O arquivo .XML é chamado de *XML-Prob*.
- compilador binário-CPN modificado: continua sendo responsável pela transformação do código assembly no modelo de rede de Petri colorida que representa o programa. Uma diferença é que em vez de duas entradas (código assembly e modelo de instruções CPN do processador) , ele recebe três entradas (recebe também o XML-Prob) e gera o modelo baseado nessas informações adicionais.
- *CPN-Tools*: assim como no PCAF, o *CPN-Tools* executa as funções de análise e oferece como retorno perfis de consumo do programa, *patches* de consumo e *clusters* de consumo.
- Ambiente "*hidden-tools*": implementa uma interface entre os resultados gerados pelo *CPN-Tools* e o usuário, através de gráficos, como o da Figura 14.

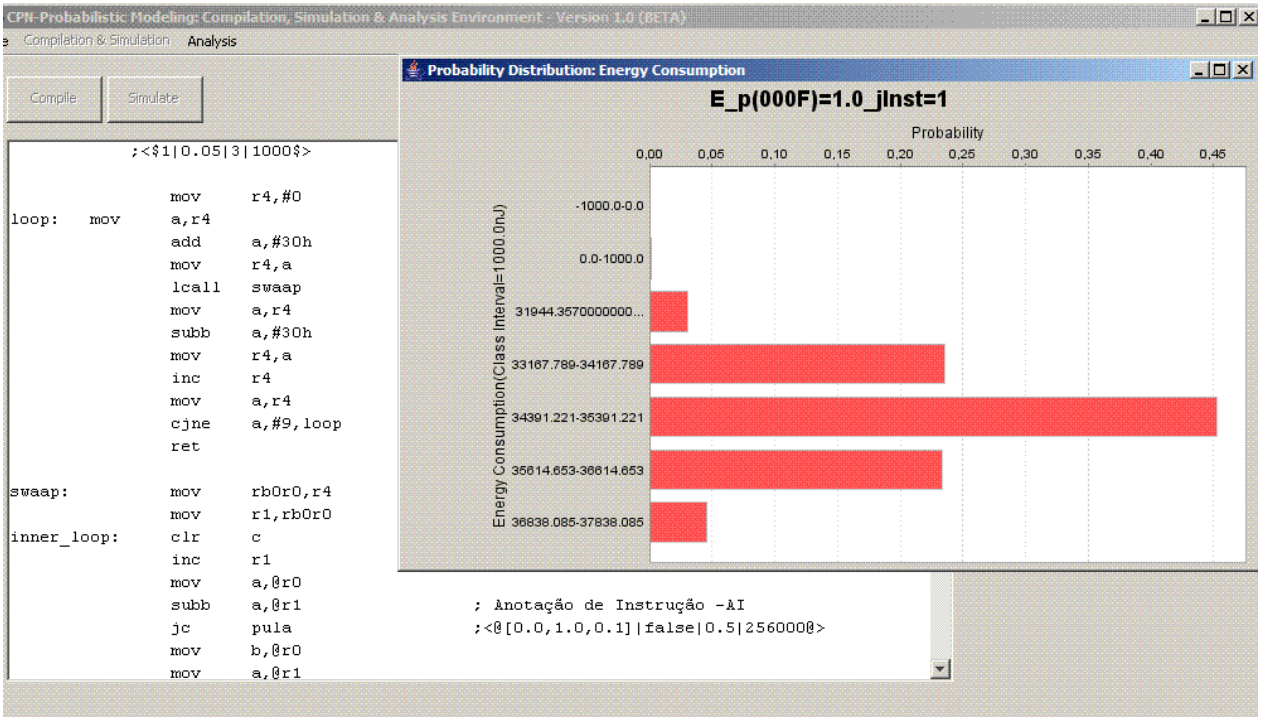

Figura 14. Gráfico que representa a distribuição de probabilidade de uma dada instrução.

O gráfico da Figura 14 apresenta a distribuição probabilística de consumo de energia para um programa, dependendo do comportamento da instrução alocada na posição de memória 000F (o mapeamento de instruções e suas respectivas alocações podem ser encontrados no arquivo XML-Prob). Segundo o gráfico, há uma probabilidade de aproximadamente 45% de que o consumo de energia esteja entre 34.391nJ e 35.391nJ.

3.5 Resumo

As seções anteriores mostraram como o uso de redes de Petri coloridas pode ajudar na análise de sistemas. A utilização desse modelo é motivada por ser uma representação formal do sistema, por permitir análises através de simulações e por aumentar o entendimento do sistema, através de sua modelagem.

Uma das técnicas utilizadas para estudar o comportamento de consumo de energia é através do mapeamento das instruções primitivas do processador, que são executadas pelo *software*, em redes de Petri coloridas. Cada instrução do processador teria uma rede de Petri colorida associada a ela, assim, após o processo de compilação, teremos uma rede complexa correspondente ao programa em análise. A análise da rede de Petri é realizada pelo PCAF e fornece ao projetista informações valiosas sobre trechos críticos da implementação, no que diz respeito ao consumo de energia. Através dessas análises, é possível, ainda, gerar uma estatística sobre os valores médios da energia consumida em cada cenário de execução.

Uma limitação dessa abordagem é o fato de que essas análises não levam em consideração os diversos fluxos de execução do programa, analisando de forma igualitária a execução de instruções condicionais, mesmo que o projetista tenha informações empíricas relativas à probabilidade de execução de cada ramo do desvio condicional. A inclusão dessa informação no

modelo da rede de Petri colorida provê uma análise mais próxima do real. O Capítulo seguinte apresenta uma extensão para esse modelo probabilístico.

Capítulo 4

Analisando o Consumo de Energia: uma Abordagem Probabilística

No capítulo anterior foram apresentadas algumas ferramentas computacionais que podem ser utilizadas na análise de consumo de energia através de redes de Petri coloridas. As ferramentas apresentadas permitem também fazer relações entre consumo de energia, trechos críticos de implementação e tempos de execução.

A otimização de consumo de energia está intimamente relacionada à exploração do melhor uso de certos parâmetros de projeto de *hardware*, como consumo de energia e desempenho. Uma atividade muito importante para obtenção de dados que ajudem em tais otimizações é a definição dos cenários de execução da aplicação (leia-se cenário de execução como conjunto de eventos que acontecem no programa) desde o pior até o melhor caso.

Uma abordagem clássica é apontada em [17] e faz a modelagem dos cenários de execução através de vetores de teste, nos quais os conjuntos de eventos e dados são informados para que se possa realizar uma simulação. Esses vetores de teste são submetidos à execução de trechos de código padrão e, com os resultados, é possível estimar o consumo de energia médio da aplicação em um domínio específico.

Uma limitação dessa abordagem é o fato de que essas análises não levam em consideração os diversos fluxos de execução do programa, analisando de forma igualitária a execução de instruções condicionais, mesmo que o projetista tenha informações empíricas relativas à probabilidade de execução de cada ramo do desvio condicional.

Ainda, segundo [17], a exploração do projeto implica na avaliação de código para uma aplicação específica. A definição de vetores de teste que cubram uma série de possibilidades para um certo processador, não é uma tarefa simples. Logo, faz-se necessária uma outra abordagem que permitisse o controle de diversos fluxos de execução do programa para cobrir uma grande variedade de dados e cenários.

A abordagem usando redes de Petri coloridas é motivada por alguns fatores:

- formalismo;
- representação gráfica dos programas a serem analisados;
- capacidade de modelagem hierárquica, permitindo vários níveis de abstração;

- ferramentas (descritas no capítulo anterior) de simulação e validação dos modelos propostos.

A contribuição desse trabalho é, além de analisar o modelo proposto em [6], estendê-lo para o uso com linguagens de programação de alto nível (no nosso caso, a linguagem C). Agora, as notações probabilísticas que determinam os cenários de execução do programa serão inseridos e analisados no código C, o que facilitará o trabalho dos projetistas.

Este capítulo descreve em detalhes a arquitetura do modelo utilizado, incluindo informações sobre seu mecanismo de funcionamento. É apresentado também o SDCC e a forma como este compilador foi alterado para implementar o modelo proposto. Por fim, a validação do modelo servirá para demonstrar o potencial do nosso modelo probabilístico.

4.1 Modelo Proposto

Com base no modelo ilustrado na Figura 15, esta seção dará uma idéia geral do modelo aqui proposto (extensão do original proposto em [6]). Cada módulo da figura será descrito independentemente e relacionado com os outros através de suas saídas, que também serão analisadas. A contribuição desse trabalho está destacada na figura.

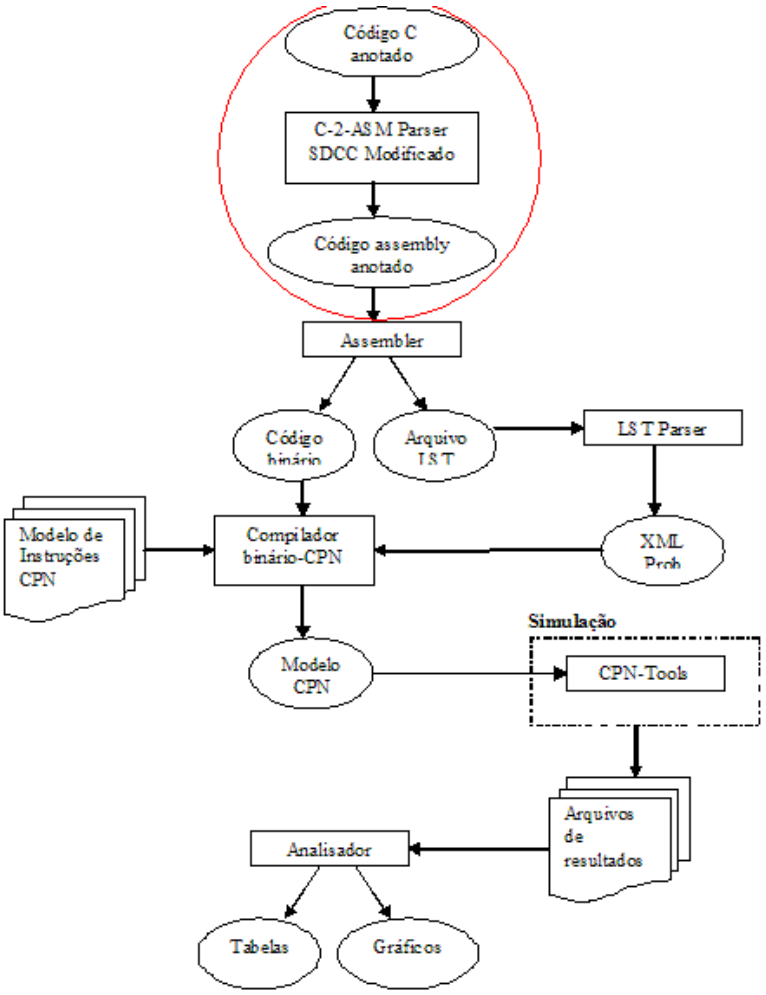

Figura 15. Arquitetura do Modelo Probabilístico de Análise de Código.

4.1.1 *C-2-ASM Parser* e o SDCC Modificado

A implementação do modelo exposto na seção anterior será através da alteração do SDCC (*Small Device C Compiler)* [7] de forma que este passe a reconhecer as notações probabilísticas no código da linguagem C, ao lado das instruções de desvio condicional.

O SDCC é um compilador C de código aberto que tem como alvo, entre outros, o microcontrolador 8051. Ele é suportado em plataformas Microsoft Windows ou Linux, porém as implementações serão realizadas e testadas nesta última, de forma a ser mais coerente com a ideologia de código aberto da comunidade que mantém o compilador. Ele é compilado pelo GCC *(GNU Compiler Colletion)* e faz uso das ferramentas auxiliares, lex [18] e bison [19], para geração dos analisadores léxico e sintático, respectivamente.

Na seção 4.2 será descrito o processo de alteração do SDCC para que este entenda as anotações no código C e as insira no código assembly gerado de forma correta. As anotações a que se refere o texto são as probabilidades que indicam os possíveis cenários de execução para a aplicação, a indicação se a execução da instrução é determinística etc.

O *C-2-ASM Parser* é a ferramenta responsável por algumas atividades fundamentais para o nosso ambiente:

- criação de um arquivo .asm intermediário, a partir de um arquivo C com anotações sobre probabilidade de execução de cada instrução de desvio. Esse arquivo é gerado através de chamada de sistema ao SDCC modificado e consegue associar probabilidades às intruções condicionais presentes no código C (alguns desvios condicionais provenientes de otimizações ficam sem anotação, nesse ponto do processo);
- definição de uma probabilidade padrão para as instruções que não foram mapeadas durante o processo de compilação. Essa etapa se faz necessária até que o compilador consiga mapear um valor de probabilidade para todas as instruções de desvio presentes do código de máquina. De forma a não interferir nos resultados das simulações, o valor de probabilidade associado a essas instruções é de 50%.

Anotação em Código

As anotações geradas pelo *C-2-ASM Parser* no código servem para preencher parâmetros da rede de Petri colorida que representa o modelo probabilístico e os critérios de parada durante a simulação. Há dois tipos de anotações:

ANOTAÇÕES DE CABEÇALHO

Dizem respeito aos critérios de parada durante a simulação da rede. A sua estrutura é como segue:

```
<$ ConfInterv | Emax | TarMet | Ns $>
```

onde:

- *ConfInterv* é um identificador de confiabilidade da simulação. A confiabilidade da simulação pode assumir seis valores:
  - 90% (`ConfInterv = 1`);
  - 92% (`ConfInterv = 2`);

  - 94% (`ConfInterv = 3`);
  - 96% (`ConfInterv = 4`);
  - 98% (`ConfInterv = 5`);
  - 99% (`ConfInterv = 6`).
- *Emax* representa o erro máximo tolerado;
- *TarMet* o critério de parada que deve ser utilizado para a simulação. Pode assumir algum dos valores seguintes:
  - potência (`TarMet = 1`);
  - consumo de energia (`TarMet = 2`);
  - tempo de execução (`TarMet = 3`).
- *Ns* é o número mínimo de ciclos de simulação que serão executados. É interessante salientar que Ns tem prioridade sobre TarMet, pois visa garantir que a simulação tenha dados suficientes antes de finalizar.

ANOTAÇÕES DE INSTRUÇÕES

As anotações de instruções estão relacionadas com os parâmetros de simulação associados ao comportamento probabilístico da instrução. A sua estrutura é como segue:

```
<@ [Pi; Pf ; Ps] | sweepmodefree | fixprob | wloop @>
```

onde:

- [Pi; Pf; Ps] é o vetor de varredura que indica os valores de probabilidades inicial, final e sua variação para a instrução;
- *sweepmode-free* indica se o vetor de varredura está ativado ou não. Se a instrução tiver comportamento dependente de valores probabilísticos, esta variável deve conter o valor falso (`sweepmodefree = false`), indicando que a simulação deve explorar os intervalos de probabilidades no cálculo dos valores de consumo de energia e tempo de execução de cada instrução. A combinação de todos os valores capturados através da varredura no vetor de probabilidade provê uma visão geral sobre todos os cenários de execução da instrução, desde o pior ao melhor caso;
- *fixprob* é o valor de probabilidade de uma instrução executar. Assume valores no intervalo [0.0, 1.0], onde 0.0 é a condição em que a condição nunca executa e 1.0 é aquela que indica que ela sempre vai executar. Instruções determinísticas são modeladas atribuindo-se o valor de *fixprob* para 1.0. Esses valores são informados pelo projetista de acordo com dados de outras ferramentas ou de observações empíricas;
- *wloop* é o valor que restringe o número de vezes consecutivas que uma instrução pode executar. Esse parâmetro permite a modelagem de instruções repetitivas-iterativas (*loops*) e impede que uma instrução probabilística execute sempre da mesma forma.

Ao inserir anotações nos nossos códigos, devemos determinar corretamente os intervalos de probabilidades que permitem simulações consistentes. Devemos também ajustar o nível de confiabilidade da nossa simulação para valores toleráveis de acordo com a aplicação. Um valor muito alto para confiabilidade (valor muito baixo de erro) leva a simulações muito mais demoradas. Ainda, os critérios de parada devem ser bem ajustados, de modo que não se obtenha simulações incompletas (e assim com valores não úteis) ou simulações muito demoradas.

4.1.2 Assembler

Compilador do código assembly. A compilação gera além do arquivo binário, um arquivo .LST em formato ASCII, que contém um mapeamento dos endereços de memória e suas respectivas instruções. O arquivo .LST contém também os comentários que havia no código assembly, preservando assim as anotações.

4.1.3 *LST-Parser*

O *LST*-Parser é responsável pelo processamento do arquivo .LST e extração dos parâmetros de cada instrução que foi anotada, conforme vemos na Figura 16. Ele gera um arquivo XML, chamado XML-Prob, que carrega as informações das anotações feitas no código.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <probabilidades>
    <Header Conf_Interval="1" ErrorMax="0.05" N_iter_Min="1000" Par="3" />
    <elemento probabilidade="[1.0,1.0,0.1]" instrucao="a,#9,loop" Endereco="000F" Iteracoes="9" FixProb="1.0" DeterInst="true" />
    <elemento probabilidade="[0.0,1.0,0.1]" instrucao="" Endereco="001B" Iteracoes="256000" FixProb="0.5" DeterInst="false" />
    <elemento probabilidade="[1.0,1.0,0.1]" instrucao="a,#39h,inner_loop" Endereco="0024" Iteracoes="5" FixProb="1.0"
      DeterInst="true" />
  </probabilidades>
```

Figura 16. Arquivo XML-Prob gerado através de um código que continha uma anotação de cabeçalho e três anotações de instruções.

Na Figura 16, vemos que há o mapeamento dos parâmetros de três instruções e suas anotações. Por exemplo, a instrução que foi mapeada na posição de memória 001B é não determinística e possui vetor de varredura variando de 0.0 a 1.0, permitindo no máximo 256mil repetições consecutivas. Num processo de “compilação reversa”, podemos extrair, através desse arquivo, os comentários que foram inseridos no código:

Anotação de Cabeçalho: `<$ 1 | 0.05 | 2 | 1000 $>`

Anotações de Instruções:
`<@ [1.0; 1.0; 0.1] | true | 1.0 | 9 @>`
`<@ [0.0; 1.0; 0.1] | false | 0.5 | 256000 @>`
`<@ [1.0; 1.0; 0.1] | true | 1.0 | 5 @>`

4.1.4 Compilador Binário-CPN

O compilador binário-CPN é responsável por construir um arquivo de modelo de redes de Petri coloridas através de suas entradas: o código binário da aplicação, o arquivo XML-Prob gerado pelo *LST-Parser* e o arquivo de mapeamento de instruções em CPN. O modelo gerado, um CPN-ML, é um arquivo em formato XML que serve de entrada para o *CPN-Tools*.

Assim, cada bloco simples de código é modelado com uma transição simples na rede de Petri colorida. Adicionalmente, instruções de laço são implementadas como *delays*, tais como *self-loop*, são também modelados como uma transição única, como podemos comprovar na Figura 17. Esta operação aumenta a velocidade da simulação.

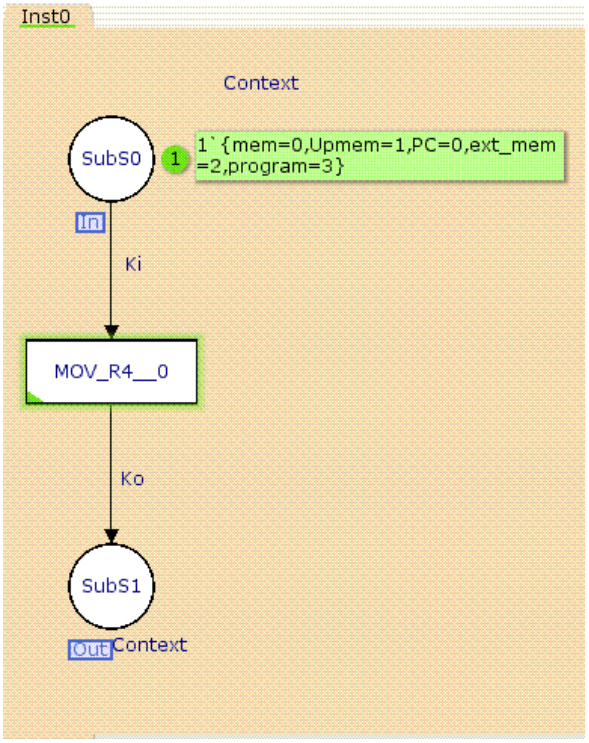

Figura 17. Representação da Instrução MOV como uma transição simples.

4.1.5 *CPN-Tools*

Na Seção 3.2 já foram apresentados os conceitos básicos da ferramenta *CPN-Tools*. Aqui, apresentaremos de forma simplificada a maneira através da qual esta ferramenta realiza suas análises.

A partir de um código, um modelo de uma rede de Petri colorida é construído com base em um conjunto de instruções CPN. Esse conjunto de instruções CPN descreve a arquitetura do processador em uso. O comportamento de cada instrução é descrito por um código CPN-ML, associado com cada transição. O código CPN-ML é responsável pela execução das funções que dão informações precisas sobre o consumo de energia. Durante a simulação, são guardadas informações sobre o consumo de energia da instrução, contexto do processador (memória e registradores), além de informações sobre variáveis de probabilidade tais como número repetido de ciclos.

No modelo proposto, o comportamento das instruções de desvio é removido do modelo de instruções CPN (apenas instruções de desvio incondicional mantém o seu comportamento original). Para essas instruções, um modelo de comportamento probabilístico foi implementado.

O ciclo de execução e as informações sobre consumo de energia são mantidos. O novo comportamento implementado contém uma série de funções para manipulações estatísticas e suporte a critérios de parada. Para essa nova abordagem, o compilador binário-CPN é responsável pela inserção dos parâmetros referentes ao modelo probabilístico, bem como por informações

referentes ao critério de parada; essas informações são obtidas através das anotações no código assembly.

O *CPN-Tools* gera uma série de arquivos contendo o consumo médio estimado para cada instrução a cada simulação. O vetor que contêm os dados do consumo médio estimado para a instrução é chamado de Vetor de Consumo.

4.1.6 Analisador de Resultados

Os resultados parciais produzidos pelo *CPN-Tools* são processados por um analisador e gera gráficos, tabelas e cenários de execução. A distribuição de probabilidades representando o tempo de execução, a energia consumida e a potência dissipada são avaliadas para cada cenário durante a simulação. Com esses dados, o analisador constrói um perfil de consumo de acordo com a freqüência das instruções de uma dada classe.

O analisador de resultados do *CPN-Probabilistic Modeling* apresenta uma série de gráficos interessantes, entre eles – e o que é objeto de nosso estudo – o gráfico de distribuição probabilística do consumo de energia.

4.2 Implementações

Esta seção apresenta as decisões de projeto tomadas e as implementações realizadas para obtermos o código assembly anotado, conforme Subseção 4.1.2, a partir de um código C para plataforma 8051.

4.2.1 Mapeamento de Código C em Código Assembly

Nesta seção serão descritos brevemente os mecanismos de tradução da linguagem C para a linguagem assembly, utilizadas pelo SDCC. Nosso objetivo aqui é explanar como blocos de estruturas em C se apresentam quando compilados em linguagem de máquina.

Serão apresentados exemplos para as estruturas condicionais mais simples (que foram abordadas nesse projeto). O entendimento de tais processos de compilação nos ajuda na marcação das probabilidades de cada instrução, durante a alteração do compilador SDCC.

Os códigos apresentados a seguir foram comentados, depois de compilados, para facilitar o entendimento.

Estrutura if-else

A tradução da estrutura if-else do C para a linguagem de máquina consiste na inicialização dos registradores apropriados, seguido de uma instrução jc (*jump if carry is set*) que desvia para a posição de memória indicada logo após a instrução condicional ou continua executando o código a partir da instrução seguinte. A instrução jc corresponde às relações < e >, no código C. Os trechos de código abaixo mostram um exemplo da estrutura if-else num código C e sua correspondente num código assembly:

C:

```
if (x < y)
```

```
{
      (...) //executa o IF
}
else
{
      (...) //executa o ELSE
}
```

ASSEMBLY:

```
; inicializa registradores
      mov   a,_main_x_1_1
      subb  a,#0x05
      mov   a,(_main_x_1_1 + 1)
      xrl   a,#0x80
      subb  a,#0x80
; verifica se deve executar o desvio
      jc    00106$
; executa as ações referentes ao ELSE e retorna
      (...)
      ret
; caso tenha desviado, executa as ações referentes ao IF e retorna
00106$:
      (...)
      ret
```

Estrutura if-elseif-else

De forma semelhante à estrutura if-else, a estrutura if-elseif-else acrescenta apenas uma instrução de desvio incondicional depois de cada instrução de comparação, exceto na última, quando o else deve ser executado. Deve-se notar que quando existe a operação de comparação == a instrução gerada será cjne (*carry and jump if not equal*) e o desvio incondicional será jz (*jump if accumulator zero*). Para as demais operações de comparação, a instrução de desvio será sjmp (*short jump*). O trecho de código exemplifica o comando:

C:

```
if (x < y)
{
      (...)
}
else if (x == y)
{
      (...)
}
else
{
      (...)
}
```

ASSEMBLY:

```
; inicializa registradores para verificar se x < y
      mov   a,_main_x_1_1
      subb  a,#0x05
      mov   a,(_main_x_1_1 + 1)
      xrl   a,#0x80
      subb  a,#0x80
; verifica se deve executar o desvio
```

```
      jc    00111$
; senão, desvia para trecho do código que testa próxima condição
      sjmp  00107$
; caso tenha desviado, executa as ações referentes ao IF e retorna
00111$:
      (...)
      ret
; inicializa registradores para verificar condição x == y
00107$:
       mov  a,_main_x_1_1
      cjne  a,ar2,00109$
      mov   a,(_main_x_1_1 + 1)
      cjne  a,ar3,00109$
; caso não tenha desviado, executa as ações referentes ao ELSE e retorna
      (...)
      ret
; executa ações referentes ao ELSE-IF
00109$:
      (...)
      ret
```

Estrutura while

A tradução da estrutura while também é bem parecida com o que foi visto até agora: primeiro há inicialização dos registradores e acumuladores, depois uma instrução de desvio (jc ou cjne) verifica se deve executar o corpo do laço e depois retornar através de um desvio incondicional sjmp para o início da estrutura ou se deve prosseguir com a execução do código, caso a condição não seja verdadeira. O código a seguir demonstra essa situação:

C:

```
while (x < y)
{
      x++;
}
```

ASSEMBLY:

```
; início do laço
00101$:
; inicialização de registradores
      mov   a,r2
      subb  a,r4
      mov   a,r3
      xrl   a,#0x80
      mov   b,r5
      xrl   b,#0x80
      subb  a,b
; verifica se deve executar o corpo do laço WHILE
      jc    00109$     ;<@90@>
; caso a condição não seja satisfeita, sai do laço e continua a execução.
      sjmp  00103$
; executa as instruções contidas no corpo do laço
00109$:
      (...)
; retorna ao início do laço através de um desvio incondicional
      sjmp  00101$
; código posterior ao laço
```

```
00103$:
      (...)
```

Estrutura while(1)

Caso particular da estrutura while; aqui, haverá um desvio incondicional para o início do laço até que uma instrução de `break` seja executada. Vejamos a seguir:

C:
```
while(1)
{
      (...)
}
```

ASSEMBLY:
```
; início do laço
00102$:
; executa instruções internas ao laço
      (...)
; desvio incondicional para o início do laço
      sjmp  00102$
```

Estrutura for

Assim como as demais estruturas, o código para a estrutura for segue uma estrutura parecida com a das demais estruturas: primeiro há inicialização dos registradores, depois há instrução de desvio que verifica se o corpo do laço será executado e instrução de desvio incondicional, caso a instrução anterior não seja verdadeira. Antes de executar as instruções contidas no corpo do laço, há ainda alteração dos valores das variáveis de controle do laço, caso existam. Depois de executar as instruções do corpo do laço, há um desvio incondicional que leva o fluxo de execução de volta para o início da estrutura. O código a seguir mostra a tradução de um laço típico:

C:
```
for(i=0; i<10; i++)
{
      (...)
}
```

ASSEMBLY:
```
; início do laço FOR
00104$:
; inicialização dos registradores
      mov   a,r4
      subb  a,#0x0A
      mov   a,r5
      xrl   a,#0x80
      subb  a,#0x80
; instrução que verifica se deve executar o corpo do laço
      jc    00112$
; desvio incondicional para o trecho do código posterior ao laço
      sjmp  00102$
; executa corpo do laço
```

```
00112$:
00105$:
; altera o valor da variável de controle do laço: i++
      inc   r2
      cjne  r2,#0x00,00113$
      (...)
; executa corpo do laço
00113$:
      (...)
; instrução de desvio incondicional que retorna ao início do laço
      sjmp  00104$
; código posterior ao laço
00102$
      (...)
```

Casos especiais: conjunções e disjunções

Um caso especial de tradução é realizado quando há uma conjunção (representado por && em código da linguagem C) ou uma disjunção (representado por || na linguagem C) como argumento da instrução condicional a ser verificada. Nesses casos, cada condição individual da conjunção/disjunção é representada por um bloco de código específico, como apresentado no trecho de código seguinte:

C:

```
if (x > 3 && y < 10 || y < 5)
{
      y++;
}
```

ASSEMBLY:

```
; inicializa registradores para verificar a condição x > 3
      mov   a,#0x03
      subb  a,_main_x_1_1
      mov   a,#(0x00 ^ 0x80)
      mov   b,(_main_x_1_1 + 1)
      xrl   b,#0x80
      subb  a,b
; verifica se a condição é verdadeira, em caso positivo, verifica a próxima
condição da conjunção
      jc    00111$
; caso a instrução não seja verdadeira, executa a instrução correspondente à
disjunção
      sjmp  00104$
; inicializa registradores para verificar a condição y < 10
00111$:
      mov   a,r2
      subb  a,#0x0A
      mov   a,r3
      xrl   a,#0x80
      subb  a,#0x80
; verifica se a expressão já pode ser considerada como verdadeira ou se
depende do resultado da disjunção
      jnc   00104$
; executa as instruções no interior da estrutura
      Sjmp  00113$
```

```
; inicializa registradores para verificar a condição y < 5
00104$:
      mov   a,r2
      subb  a,#0x05
      mov   a,r3
      xrl   a,#0x80
      subb  a,#0x80
; verifica se deve executar as instruções do interior da estrutura
      jc    00113$
; avança sem executar as instruções da estrutura
      sjmp  00106$
; executa instruções do interior da estrutura
00113$:
      (...)
; continua execução do programa
00106$
      (...)
```

4.2.2 Anotações na Linguagem C

Na primeira etapa de compilação do código C para o código assembly, devemos inserir anotações nas instruções condicionais de forma a indicar a probabilidade de execução de cada uma delas. Consideraremos algumas abordagens de forma a explicitar a decisão pela utilização da solução adotada aqui.

Uma primeira abordagem seria inserir a anotação probabilística na forma de comentário (mesmo padrão usado no código assembly) imediatamente antes da instrução à qual o valor se refere. Esse padrão foi inutilizado por duas razões básicas. A primeira é que todos os comentários são eliminados do programa pelo analisador léxico. Logo os valores seriam perdidos na primeira etapa da compilação. Mesmo desenvolvendo um meio de recuperar tais valores (alterando a forma como comentários são tratados pelo analisador léxico ou criando comentários especiais para a instrução) o padrão ainda seria descartado por um segundo motivo; já que a transformação da linguagem C para a linguagem assembly não gera o mesmo número de blocos de código. Um único bloco alto-nível pode ser dividido em diversas partes quando levadas para um nível inferior. Assim, o compilador precisaria de informações de como fazer a distribuição do valor de probabilidade entre os diversos blocos gerados.

No processo de tradução da linguagem C para assembly é interessante notar que há divisão de um bloco de instrução em mais de um bloco no código gerado quando a condição de execução da instrução é formada por uma conjunção (ou associação ou disjunção). Assim, para cada condição deve haver um valor de probabilidade associado. Logo, devemos alterar a definição dos não terminais referentes às expressões relacionais e de igualdade (<, <=, >, >=, = =, !=).

A abordagem adotada consiste na inserção junto aos comandos condicionais, que controlam a execução das instruções de laço, da anotação probabilística correspondente, como explicado na subseção 4.1.1.

Essa nova abordagem impõe algumas restrições à sintaxe da gramática original; a mais severa delas é que toda expressão relacional e de igualdade deve vir acompanhada de uma anotação, já que se tornarmos opcional a inserção do símbolo, teremos um problema de *binding*. Por exemplo, se o uso da probabilidade for condicional, no trecho de código

```
while (x > 5 <@[0.0;1.0;0.1]|true|0.5|100@>
 && y < 20 <@[0.0;1.0;0.1]|true|0.7|100@>)
```

seria impossível saber se "`<@[0.0;1.0;0.1]|true|0.7|100@>)`" se refere à expressão anterior "`y < 20`" ou à expressão geral "`x > 5 <@[0.0;1.0;0.1]|true|0.5|100@> && y < 20`. Com a obrigatoriedade de inserção da anotação, temos referência sempre à instrução imediatamente anterior.

4.2.3 *C-2-ASM Parser*

Aplicativo que integra todas as atividades da compilação do arquivo-fonte C para o arquivo-fonte na linguagem assembly, no formato desejado.

O *C-2-ASM Parser* executa duas tarefas principais:

- usa o SDCC para transformar o arquivo C em arquivo assembly (esse arquivo ainda não é o final que será utilizado);
- lê a anotação de cabeçalho do arquivo C e adiciona ao arquivo assembly final.
- lê o arquivo assembly intermediário e o copia para o final, acrescentando uma anotação probabilística padrão às intruções de desvio condicional que não estão anotadas.

A Figura 18 mostra o *C-2-ASM Parser* em execução.

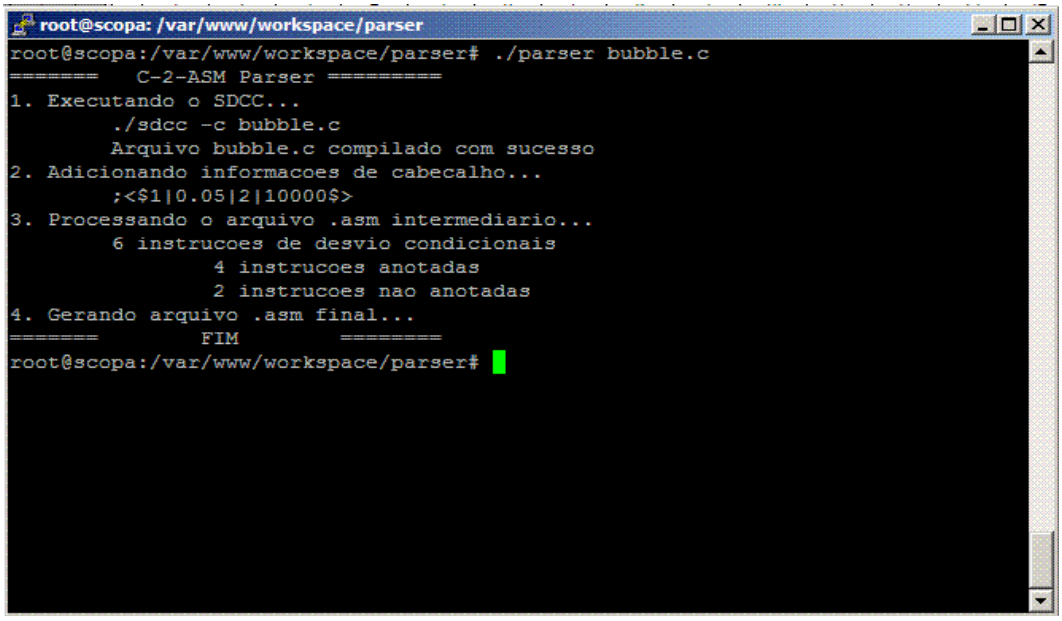

Figura 18. *C-2-ASM Parser* em execução.

4.2.4 Alteração do SDCC

Esta seção descreve as atividades básicas de um compilador moderno e como o comportamento dessas atividades foi modificado para atender às necessidades já mencionadas nas seções anteriores. Os conceitos básicos das etapas de compilação apresentados aqui foram retirados de [20].

Analisador Léxico

O Analisador léxico é a primeira etapa de um compilador.

Análise léxica (*scanning*) é o processo de análise dos caracteres que formam as palavras de um programa para verificar se a palavra está bem formada. Neste caso, são produzidos símbolos representativos de grupos de palavras pertecentes a uma determinada categoria léxica. Estes símbolos são chamados de *tokens*. Esses *tokens* podem ser manipulados por um *parser* na próxima etapa do processo de compilação.

A análise léxica é a forma de verificar determinado alfabeto para uma linguagem de programação. Quando analisamos uma palavra, podemos determinar se existe na linguagem algum símbolo correspondente à seqüência de caracteres encontrada.

Além de identificar *tokens*, é função de um analisador léxico:

- contar as linhas de um programa;
- contar a quantidade de caracteres em um arquivo;
- tratar espaços;
- eliminar comentários.

O SDCC utiliza, também, a análise léxica para eliminar os símbolos que não pertencem à sua gramática. Assim, a inclusão do token “@”, por exemplo, seria simplesmente eliminado pelo SDCC. A maioria dos compiladores, em casos de não reconhecimento de um símbolo como pertencente à sua gramática, exibiria um erro para o usuário.

A alteração que será realizada no analisador léxico será o reconhecimento das instruções de cabeçalho e de instruções, conforme vemos a seguir, no arquivo SDCC.lex:

```
...
"<@["{D}+"."{D}+";"{D}+"."{D}+";"{D}+"."{D}+"]|"("true"|"false")"|"{D}+"."{D}+
"|"{D}+"@>"
      { count(); return(check_notation(1)); }

"<$"{D}"|"{D}"."{D}+"|"{D}"|"{D}+"$>"
      { count(); return(check_notation(2)); }

...
static int check_notation(int type)
{
      sprintf (yylval.yychar, "%s", yytext);
      if (type == 1)
      {
            return(INST_NOTATION);
      }
      else
      {
            return(HEADER_NOTATION);
      }
}
...
```

Analisador Sintático

Análise sintática analisa a estrutura de um texto (que foi dividido em *tokens*, pelo analisador léxico), através de um conjunto de regras sintáticas bem definidas – as quais chamamos de gramática da linguagem – com o objetivo de obter unidades estruturais organizadas.

O analisador léxico, explicado na etapa anterior, é também responsável pela criação de uma árvore que contém as informações sintáticas de um determinado programa. Essa árvore será utilizada pelo analisador semântico na próxima etapa do processo de compilação.

Para que o valor das anotações seja passado para o analisador semântico, e continue por todo processo de compilação, é necessário a extensão da estrutura que representa a nossa árvore sintática abstrata. O código a seguir, no arquivo SDCCast.h, mostra como realizamos tal alteração e inserimos um campo que guardará o valor das anotações de cada instrução de desvio:

```
/* expression tree   */
typedef struct ast
 {

  ASTTYPE type;
  unsigned decorated:1;
  unsigned isError:1;
  unsigned funcName:1;
  unsigned rvalue:1;
  unsigned lvalue:1;
  unsigned initMode:1;
  unsigned reversed:1;
  int level;
  int block;
  int seqPoint;

  union
   {
      value *val;
      sym_link *lnk;
      struct operand *oprnd;
      unsigned op;
   }
  opval;

  /* union for special processing */
  union
   {
      char *inlineasm;
      char probability[100];
      literalList *constlist;
   ...
   }
  values;
   ...
```

No arquivo SDCC.y, o analisador sintático preenche a árvore sintática. Foram acrescentadas novas produções à gramática de forma que seja reconhecida a anotação de cabeçalho no início do arquivo. Foram alteradas também as expressões relacionais e de igualdade, para que o analisador reconheça as produções correspondentes às anotações da instrução.

```
  ...
  %token <yychar> IDENTIFIER TYPE_NAME INST_NOTATION HEADER_NOTATION
  %token <val> CONSTANT STRING_LITERAL
  ...
  %type <sym> inst_notation header_notation file_tmp
  %type <sym> identifier declarator  declarator2 declarator3 enumerator_list
enumerator

  ...
  %start file_tmp

  %%
  file_tmp
      : header_notation file
      ;

  file
      : external_definition
      | file external_definition
      ;
  ...

  | relational_expr '<' shift_expr inst_notation {
      ast *ex;
      ex = (port->lt_nge ?
            newNode('!',newNode(GE_OP,$1,$3),NULL) :
            newNode('<', $1,$3));
      sprintf(ex->values.probability, "%s", ((char*)symbolVal($4))+1);
      $$ = ex;
  }
  | relational_expr '>' shift_expr inst_notation {
      ast *ex;
      ex = (port->gt_nle ?
             newNode('!',newNode(LE_OP,$1,$3),NULL) :
             newNode('>',$1,$3));
      sprintf(ex->values.probability, "%s", ((char*)symbolVal($4))+1);
      $$ = ex;
  }
...
```

Abstract Syntatic Tree (AST)

Uma árvore sintática abstrata é uma árvore finita e rotulada, em que os nós internos representam operadores de um programa de computadores, e as folhas desses nós representam os operandos. AST são usadas como um estrutura intermediária que será transformada em um outro programa. A diferença básica entre uma árvore sintática e uma *parse tree* é que esta não descarta nenhum valor, enquanto aquela mantém apenas os nós que afetam a sua estrutura semântica. Um exemplo da omissão gerada pelas AST são os parênteses, que são dispensáveis devido à estrutura da árvore.

Para passar o valor da probabilidade adiante no processo de compilação, precisamos estender a estrutura que define o código intermediário. O arquivo SDCCicode.h descreve o tipo *operand* que é usado na criação do código intermediário:

```
/* typedef for operand */
```

```
typedef struct operand
 {
  OPTYPE type;                /* type of operand */
  unsigned int isaddr:1;      /* is an address   */
  unsigned int aggr2ptr:1;
  unsigned int isvolatile:1;  /* is a volatile operand */
  unsigned int isGlobal:1;    /* is a global operand */
  unsigned int isPtr:1;       /* is assigned a pointer */
  unsigned int isGptr:1;      /* is a generic pointer  */
  unsigned int isParm:1;      /* is a parameter        */
  unsigned int isLiteral:1;   /* operand is literal    */

  int key;
  char probability[100];
  union
   {
    struct symbol *symOperand;
    struct value *valOperand;
    struct sym_link *typeOperand;
   }
  operand;

  bitVect *usesDefs;
  struct asmop *aop;
 }
operand;
```

Para passar o valor da probabilidade da AST para a estrutura *operand*, precisamos apenas alterar a função ast2iCode(), do arquivo SDCCicode.c:

```
  ...
  case '>':
  case '<':
  case LE_OP:
  case GE_OP:
  case EQ_OP:
  case NE_OP:
   {
    operand *leftOp, *rightOp;

    leftOp  = geniCodeRValue (left , FALSE);
    rightOp = geniCodeRValue (right, FALSE);

      operand *result;
      result = geniCodeLogic (leftOp, rightOp, tree->opval.op);
      sprintf(result->probability, "%s", tree->values.probability);
   return result;
   }
   ...
```

Código Intermediário

O gerador de código intermediário será acionado quando o programa for analisado léxica, sintática e semanticamente, e estiver correto do ponto de vista das três análises citadas. Neste ponto do projeto, finda-se a parte de análise e tem início o processo de síntese. Para essa geração,

deve-se percorrer a árvore em profundidade (*depth first)* para que o código seja gerado das folhas para os nós.

Para exemplificar, apresentamos a seguir um exemplo de como acontece a síntese para algumas instruções de desvio. O arquivo a ser modificado é o mcs51/gen.c, função *genIfxJump*:

```
/*-----------------------------------------------------------------*/
/* genIfxJump :- will create a jump depending on the ifx           */
/*-----------------------------------------------------------------*/
static void
genIfxJump (iCode * ic, char *jval, operand *left, operand *right, operand
*result)
{
 symbol *jlbl;
 symbol *tlbl = newiTempLabel (NULL);
 char *inst;

 D(emitcode (";     genIfxJump",""));

 /* if true label then we jump if condition
  supplied is true */
  char probability[100];
  if (IC_TRUE (ic)) {
      jlbl = IC_TRUE (ic);
        if (strcmp (jval, "a") == 0){
            inst = "jz";
            sprintf(probability, "%s", result->probability);
        }
        else{
            if (strcmp (jval, "c") == 0){
                  inst = "jnc";
            }
            else{
                  inst = "jnb";
            }
            sprintf(probability, "%s", prob_complement(result->probability));
        }
  }
  else{
      /* false label is present */
      jlbl = IC_FALSE (ic);
        if (strcmp (jval, "a") == 0){
            inst = "jnz";
            sprintf(probability, "%s", prob_complement(result->probability));
        }
        else{
            if (strcmp (jval, "c") == 0){
                  inst = "jc";
            }
            else{
                  inst = "jb";
            }
            sprintf(probability, "%s", result->probability);
        }
    }
  if (strcmp (inst, "jb") == 0 || strcmp (inst, "jnb") == 0)
    emitcode (inst, "%s,%05d$     ;%s", jval, (tlbl->key + 100), probability);
  else
```

```
    emitcode (inst, "%05d$     ;%s", tlbl->key + 100, probability);
  freeForBranchAsmop (result);
  freeForBranchAsmop (right);
  freeForBranchAsmop (left);
  emitcode ("ljmp", "%05d$", jlbl->key + 100);
  emitcode ("", "%05d$:", tlbl->key + 100);

  /* mark the icode as generated */
  ic->generated = 1;
}
```

4.3 Resumo

As seções anteriores mostraram como o modelo proposto em [6] foi estendido, incluindo sua implementação e validação. Foi apresentada também uma explicação clara de como esse novo modelo deve ser utilizado para se alcançar seu melhor desempenho.

Espera-se que a extensão proposta difunda a distribuição de análises probabilísticas de fluxos de execução como uma técnica poderosa na modelagem de sistemas em que o consumo de energia é crítico. O seu objetivo é agir junto a algum *framework* de modelagem/ análise de consumo de energia no nível de *hardware* e deixar o projetista livre para decidir quais as melhores estratégias a seguir.

A abordagem probabilística permitirá também que a ocorrência de eventos externos (como a alteração na intensidade da corrente elétrica no dispositivo ou carga baixa de bateria, por exemplo) possa ser representada como um conjunto de probabilidades, permitindo simulações com cenários de execução mais próximos dos reais.

Por fim, é importante destacar que a ferramenta que serviu de base para simulação e implementação do modelo, a *CPN-Tools*, é uma ferramenta de propósito geral, logo uma ferramenta de propósito específico pode apresentar resultados mais satisfatórios.

Capítulo 5

Estudo de Caso: Ordenação Bolha

Para consolidar o modelo apresentado no capítulo anterior, faremos aqui a análise do código de uma conhecido algoritmo de ordenação: o algoritmo de ordenação bolha (*bubble sort*) [20]. O objetivo deste estudo de caso é demonstrar a potencialidade (e eventuais limitações) do modelo proposto, através de todas as suas etapas.

Este estudo de caso ilustra o processo de análise e anotação do código e também os resultados obtidos para os diferentes possíveis cenários de execução.

5.1 Ordenação Bolha

O *bubble sort* trabalha através de comparação de cada item de um vetor com o elemento que se encontra logo após dele, e trocando-os de posição caso seja necessário. O algoritmo repete este processo até que seja possível percorrer a lista inteira sem efetuar nenhuma troca (em outras palavras, o vetor está na ordem correta). Num algoritmo de ordenação crescente, o processamento do laço descrito acima faz com que os valores maiores sejam direcionados para o fim da lista enquanto os valores menores ocupam as posições iniciais.

O *bubble sort* é um algoritmo de ordenação de Ordem Quadrática, por isso ele não é recomendado para programas que precisam de velocidade e operem com grande quantidade de dados. De fato, no melhor caso são executadas $(n^2)/2$ operações relevantes. No pior caso, são feitas $2n^2$ operações. No caso médio, são realizadas cerca de $(5n^2)/2$ operações. O tempo de execução do *bubble sort*, de acordo com o número de entradas, é apresentado na Figura 19.

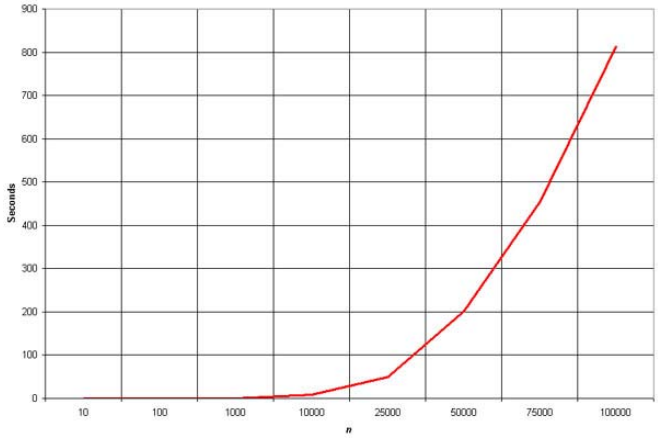

Figura 19. Gráfico de tempo de execução para o *bubble sort*.

A escolha do *bubble sort* para este estudo de caso atende a três necessidades básicas:

- a implementação do algoritmo é simples, não necessitando códigos complexos;
- o algoritmo possui uma quantidade grande de instruções condicionais (cuja notação e tradução são o objeto de estudo desse trabalho);
- o desempenho do algoritmo é muito diferente no melhor e no pior caso de execução; enquanto o algoritmo apresenta o melhor tempo de execução para um vetor ordenado, ele apresenta o pior tempo para um vetor em ordem inversa. Dessa forma, podemos mostrar claramente as análises de consumo de energia para os diversos cenários de execução.

5.2 Implementação

A Listagem 1 apresenta o código referente ao algoritmo de ordenação bolha. Note que o código já possui anotações referentes ao percentual (correspondente à probabilidade de execução) associado a cada instrução de desvio.

Listagem 1: O arquivo *bubble.c*

```
<$1|0.05|2|10000$>

#define ARRAY_SIZE 20
void main()
{
    /* definicao e inicializacao do vetor de elementos */
    int iarray[ARRAY_SIZE] =
      {15,20,4,89,5,0,74,29,36,50,41,22,87,10,15,98,60,20,10,100};
    int x, y, holder;

    for(x = ARRAY_SIZE - 1; x > 0 <@[1.0;1.0;0.1]|true|0.95|20@>; x--)
    {
      for(y = 0; y < x <@[1.0;1.0;0.1]|true|0.5|10@>; y++)
      {
            /* compara elementos vizinhos */
            if(iarray[y] > iarray[y+1] <@[0.0;1.0;0.1]|false|1.0|100@>)
            {
                  holder      = iarray[y+1];
                  iarray[y+1] = iarray[y];
```

```
                iarray[y]   = holder;
            }
        }
    }
}
```

Analisaremos a seguir as anotações associadas a cada instrução de desvio:

- `for(x = ARRAY_SIZE - 1; x > 0 <@[1.0;1.0;0.1]|true|0.95|20@>; x–)`

como o laço pode executar no máximo 20 vezes (`ARRAY_SIZE == 20`), a probabilidade de que a variável x tenha um valor maior que zero é de 95%, o que corresponde à anotação.

- `for(y = 0; y < x <@[1.0;1.0;0.1]|true|0.5|10@>; y++)`

conforme o laço vá executando, os valores das variáveis x e y vão se aproximando, o que nos leva a concluir que a instrução condicional é verdadeira, aproximadamente, em 50% dos casos.

- `if(iarray[y] > iarray[y+1] <@[0.0;1.0;0.1]|false|1.0|100@>)`

para estudar o comportamento do consumo de energia baseado nos cenários de execução determinados por esta instrução, determinou-se o valor de *sweepmodefree* para *false*, o que indica que usaremos o vetor de probabilidades. Nesse caso, teremos gráficos que mostram distribuições de probabilidade desde o pior caso (pi = 0.0) até o melhor caso (pf = 1.0);, totalizando 10 cenários de execução distintos (ps = 0.1).

De acordo com o código, temos ainda as seguintes informações:

- confiabilidade da simulação = 90%;
- erro máximo tolerado = 5%;
- critério de Parada = consumo de energia;
- número máximo de ciclos durante a simulação = 1000;

A Listagem 2 apresenta o código gerado pelo *parser* e que será utilizado para gerar os resultados gráficos que serão apresentados na seção seguinte.

Listagem 1: O arquivo *bubble.asm*

```
;<$1|0.05|2|10000$>

(...)
;bubble.c:11: for(x = ARRAY_SIZE - 1; x > 0 <@[1.0;1.0;0.1]|true|0.95|20@>; x-
-)
;       genAssign
        mov   _main_x_1_1,#0x13
        clr   a
        mov   (_main_x_1_1 + 1),a
00108$:
;       genCmpGt
;       genCmp
```

```
	clr	c
;	Peephole 181	changed mov to clr
	clr	a
	subb	a,_main_x_1_1
;	Peephole 159	avoided xrl during execution
	mov	a,#(0x00 ^ 0x80)
	mov	b,(_main_x_1_1 + 1)
	xrl	b,#0x80
	subb	a,b
;	genIfxJump
	jc	00119$	;<@[1.0;1.0;0.1]|true|0.95|20@>
;	Peephole 251.a	replaced ljmp to ret with ret
	ret
00119$:
;bubble.c:13: for(y = 0; y < x <@[1.0;1.0;0.1]|true|0.5|10@>; y++)
;	genAssign
	mov	r4,#0x00
	mov	r5,#0x00
00104$:
;	genCmpLt
;	genCmp
	clr	c
	mov	a,r4
	subb	a,_main_x_1_1
	mov	a,r5
	xrl	a,#0x80
	mov	b,(_main_x_1_1 + 1)
	xrl	b,#0x80
	subb	a,b
;	genIfxJump
	jc	00120$	;<@[1.0;1.0;0.1]|true|0.5|10@>
	ljmp	00110$
00120$:
;bubble.c:16: if(iarray[y] > iarray[y+1] <@[0.0;1.0;0.1]|false|0.9|100@> ||
;	genLeftShift
;	genLeftShiftLiteral
;	genlshTwo
	mov	ar6,r4
	mov	a,r5
	xch	a,r6
	add	a,acc
	xch	a,r6
	rlc	a
	mov	r7,a
;	genPlus
;	Peephole 236.g	used r6 instead of ar6
	mov	a,r6
	add	a,#_main_iarray_1_1
	mov	r0,a
;	genPointerGet
;	genNearPointerGet
	mov	ar6,@r0
	inc	r0
	mov	ar7,@r0
	dec	r0
;	genCast
	mov	ar2,r4
;	genPlus
;	genPlusIncr
	inc	r2
```

```
;       genLeftShift
;       genLeftShiftLiteral
;       genlshOne
;       Peephole 254      optimized left shift
        mov   a,r2
        add   a,r2
;       genPlus
;       Peephole 177.b    removed redundant mov
        mov   r2,a
        add   a,#_main_iarray_1_1
        mov   r0,a
;       genPointerGet
;       genNearPointerGet
        mov   ar2,@r0
        inc   r0
        mov   ar3,@r0
        dec   r0
;       genCmpGt
;       genCmp
        clr   c
        mov   a,r2
        subb  a,r6
        mov   a,r3
        xrl   a,#0x80
        mov   b,r7
        xrl   b,#0x80
        subb  a,b
;       genIfxJump
        jc    00101$      ;<@[0.0;1.0;0.1]|false|0.1|100@>
;       Peephole 112.b    changed ljmp to sjmp
        sjmp  00106$
00101$:
;bubble.c:19: holder    = iarray[y+1];
;       genCast
        mov   ar2,r4
;       genPlus
;       genPlusIncr
        inc   r2
;       genLeftShift
;       genLeftShiftLiteral
;       genlshOne
;       Peephole 254      optimized left shift
        mov   a,r2
        add   a,r2
;       genPlus
;       Peephole 177.b    removed redundant mov
        mov   r2,a
        add   a,#_main_iarray_1_1
        mov   r0,a
;       genPointerGet
;       genNearPointerGet
        mov   ar2,@r0
        inc   r0
        mov   ar3,@r0
        dec   r0
;bubble.c:20: iarray[y+1] = iarray[y];
;       genLeftShift
;       genLeftShiftLiteral
;       genlshTwo
        mov   ar6,r4
```

```
      mov   a,r5
      xch   a,r6
      add   a,acc
      xch   a,r6
      rlc   a
      mov   r7,a
;     genPlus
;     Peephole 236.g    used r6 instead of ar6
      mov   a,r6
      add   a,#_main_iarray_1_1
      mov   r1,a
;     genPointerGet
;     genNearPointerGet
      mov   ar6,@r1
      inc   r1
      mov   ar7,@r1
      dec   r1
;     genPointerSet
;     genNearPointerSet
      mov   @r0,ar6
      inc   r0
      mov   @r0,ar7
      dec   r0
;bubble.c:21: iarray[y]       = holder;
;     genPointerSet
;     genNearPointerSet
      mov   @r1,ar2
      inc   r1
      mov   @r1,ar3
      dec   r1
00106$:
;bubble.c:13: for(y = 0; y < x <@[1.0;1.0;0.1]|true|0.5|10@>; y++)
;     genPlus
;     genPlusIncr
      inc   r4
      cjne  r4,#0x00,00123$         ;<@[0.0;1.0;0.1]|true|0.5|1000@>

      inc   r5
00123$:
      ljmp  00104$
00110$:
;bubble.c:11: for(x = ARRAY_SIZE - 1; x > 0 <@[1.0;1.0;0.1]|true|0.95|20@>; x-
-)
;     genMinus
;     genMinusDec
      dec   _main_x_1_1
      mov   a,#0xff
      cjne  a,_main_x_1_1,00124$          ;<@[0.0;1.0;0.1]|true|0.5|1000@>

      dec   (_main_x_1_1 + 1)
00124$:
      ljmp  00108$
00112$:
      ret
      .area CSEG    (CODE)
      .area XINIT   (CODE)
```

5.3 Resultados

Nesta seção apresentaremos e comentaremos brevemente alguns dos gráficos gerados pelo *CPN Probabilistic Modeling*, após as simulações.

Como explicado anteriormente, a ferramenta simula cenários de execução através do mapeamento de um conjunto de probabilidades para um conjunto de eventos do sistema. A simulação é pré-definida, permitindo dois modos de operação: modo normal (instruções com probabilidade fixa de execução) e modo *sweep* (instruções cujo valor de probabilidade varia de acordo com a execução do código e das entradas).

Se a probabilidade de o evento ocorrer for fixa, há apenas um cenário a ser definido e a simulação pode ser feita em modo de operação normal, calculando os valores médios de consumo de energia, de acordo com a probabilidade de cada ponto do cenário, como vemos na Figura 20. Podemos ver que há cerca de 24% de probabilidade de o consumo ser entre 33µJ e 34µJ; 45% de ser entre 34µJ e 35µJ e cerca de 23% de ser entre 35µJ e 36µJ.

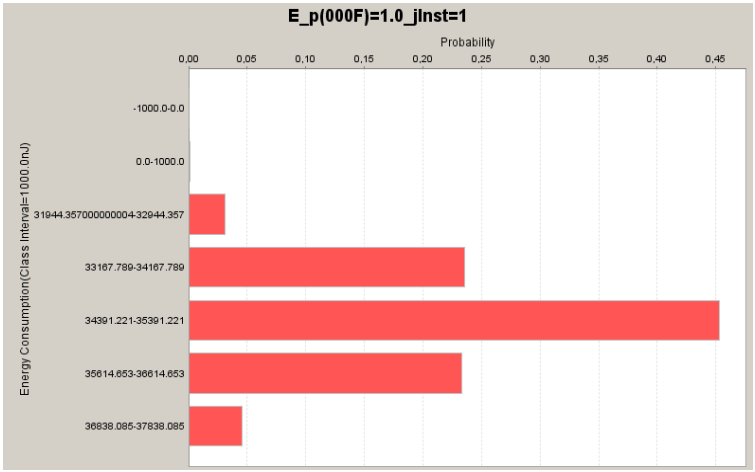

Figura 20. Análise de consumo de energia para instrução com probabilidade fixa.

Entretanto, para um sistema dinâmico, a probabilidade dos eventos podem mudar de acordo com o contexto do ambiente ou com os dados carregados, fazendo-se necessária a análise em mais de um cenário. No nosso caso, desejamos estudar os vários cenários para a instrução `if(iarray[y] > iarray[y+1] <@[0.0;1.0;0.1]|false|1.0|100@>)` de forma a determinar quando o consumo de energia é crítico. As Figuras 21, 22 e 23 apresentam alguns dos cenários de distribuição probabilística gerados para o programa, analisando essa instrução:

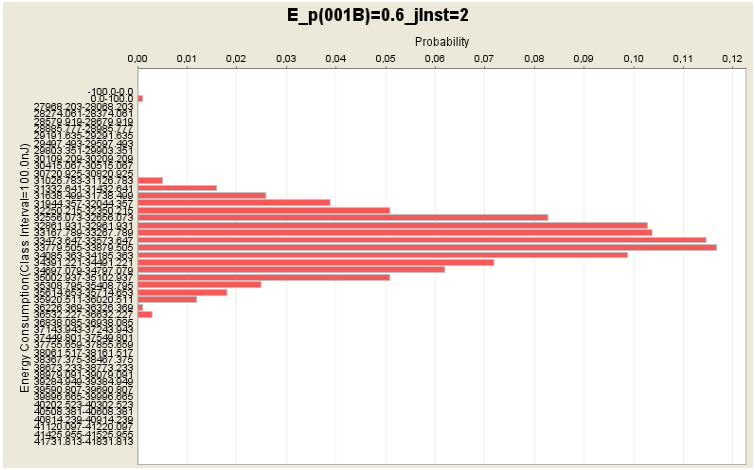

Figura 21. Análise de consumo de energia para instrução `iarray[y] > iarray[y+1]`, considerando probabilidade de 60%.

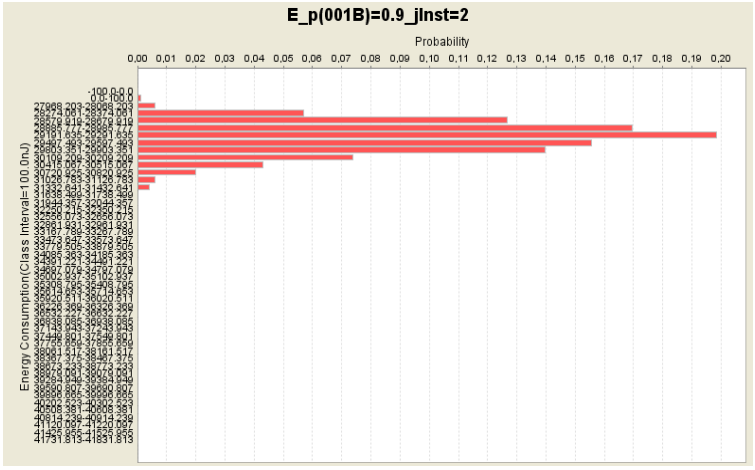

Figura 22. Análise de consumo de energia para instrução `iarray[y] > iarray[y+1]`, considerando probabilidade de 90%.

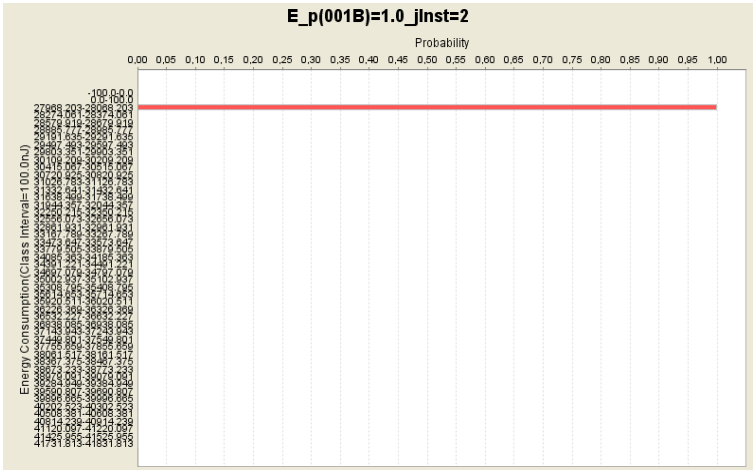

Figura 23. Análise de consumo de energia para instrução `iarray[y] > iarray[y+1]`, considerando probabilidade de 100% (melhor caso).

Com base nas Figuras 21, 22 e 23, vemos que o consumo de energia, neste caso, varia bastante, dependendo do cenário de execução (nesse caso, definido pelo conjunto de entradas) considerado.

Capítulo 6

Conclusões

Apresentamos neste trabalho de conclusão de curso um estudo sobre uma estratégia para consumo de energia em dispositivos móveis baseada em *software*. Além de propor uma nova abordagem, este trabalho visa a familiarizar os leitores não especialistas no assunto com as tendências mais recentes no desenvolvimento de sistemas embarcados e a constante preocupação com a diminuição do consumo de energia.

A nossa contribuição para o modelo probabilístico de análise de consumo de energia foi a extensão dos arquivos de entrada da linhagem assembly para a linguagem C. Com essa extensão, o projeto proposto poderá ser utilizado por um número maior de projetistas, já que inserir anotações em código C é mais fácil do que fazê-lo em código assembly.

O grande objetivo deste *framework* é permitir que os projetistas possam estudar os impactos causados por uma determinada instrução (ou trecho de código) no consumo de energia, quando os cenários de execução são alterados. É interessante lembrar que o uso de probabilidades, no nosso caso, substitui o estudo do comportamento do programa utilizando uma série de vetores de teste diferentes.

A abordagem probabilística permite também que a ocorrência de eventos externos (como a alteração na intensidade da corrente elétrica no dispositivo ou carga baixa de bateria, por exemplo) possa ser representada como um conjunto de probabilidades, permitindo simulações com cenários de execução mais próximos dos reais.

6.1 Dificuldades Encontradas

Durante a fase inicial do projeto, tivemos certa dificuldade de encontrar um bom padrão para descrição das anotações probabilísticas no código-fonte das aplicações.

Apesar de haver comunicação entre os módulos do sistema, não foi possível integrá-los numa única ferramenta. Enquanto nosso trabalho foi desenvolvido na plataforma Linux, o *CPN Probabilistic Modeling* foi desenvolvido para a plataforma Microsoft Windows. Mesmo sendo possível portar o *CPN Probabilistic Modeling* (ele foi desenvolvido em Java) para a plataforma Linux e realizar a integração, não tivemos tempo hábil para realizar tal atividade.

Uma outra dificuldade foi a ausência de documentação referente ao processo de conversão de código no SDCC. Todo o trabalho realizado aqui foi baseado em observações empíricas e discussões com membros ativos da lista de discussões da comunidade do projeto.

6.2 Trabalhos Futuros

A primeira contribuição a ser incorporada a este trabalho é certamente a integração de todo o *framework* numa única ferramenta, deixando assim, transparente para o projetista as diversas operações que são realizadas até que sejam obtidos os resultados das análises. Para que seja realizada a integração, deveremos:

- definição de um sistema operacional padrão (ou desenvolver uma plataforma para os dois mais utilizados: Microsoft Windows e Linux) para o *framework*;
- realização de simulações no *CPN-Tools* de forma automática. Devemos alterar o *CPN Probabilistic Modeling* para que ele inicialize o *CPN-Tools* e rode as simulações sem precisar da interferência do usuário (atualmente as simulações são realizadas manualmente pelo usuário, no *CPN-Tools* – que não é uma ferramenta com interface *user-friendly*);
- deveremos também trabalhar mais profundamente no compilador para que ele consiga gerar as anotações probabilísticas para todas as instruções de desvio e no formato adequado, fazendo desnecessária a utilização de um *parser*.

Outra contribuição que pode ser incorporada ao modelo é o desenvolvimento de um protótipo de integração do nosso modelo com o modelo de análise de tempo de execução no pior caso (WCET – *Worst Case Execution Time*), proposto em [22], de forma que seja possível, no nosso modelo, a determinação desses cenários de execução crítica automaticamente.

Por fim, um estudo das limitações do modelo proposto pode trazer novas idéias na expansão do arcabouço para análise do consumo de energia. Pode-se incorporar ao modelo o tratamento para arquiteturas com *pipelines* e superescalares ou até mesmo a extensão deste para análise de códigos de linguagem de programação de alto nível orientadas a objeto.

Bibliografia

[1] Apple Batteries. Disponível em: http://www.apple.com/batteries. Último acesso em: 11 de Junho de 2006.

[2] Intel Centrino Mobile Technology. Disponível em: http://www.intel.com/products/centrino. Último acesso em: 11 de Junho de 2006.

[3] Tiwari, V., Malik, S., e Wolfe, A. “Power analysis of embedded software: A first step towards software power minimization”. IEEE Transactions on Very Large Scale Integration Systems: 437–445, Dezembro de 1994.

[4] Ayala, J., Veidenbaum, A., e Lopez-Vallejo, M. “Power-aware compilation for register file energy reduction”. International Journal of Parallel Programming: 451-467, 2003.

[5] Maciel, P. R. M. “Introdução às Redes de Petri e Aplicações”. X Escola de Computação. Campinas, Brasil, 1996.

[6] Oliveria Júnior, M. N. “Analyzing Software Performance and Energy Consumption of Embedded Systems by Probabilistic Modeling: An Approach Based on Coloured Petri Nets”. 27th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (Petri Nets 2006). Turku, Finlândia, Junho de 2006.

[7] SDCC – Small Device C Compiler. Disponível em: http://sdcc.sourceforge.net. Último acesso em: 27 de Novembro de 2006.

[8] Schaller, R. R. “Moore’s Law: past, present and future”. Spectrum, IEEE: 52-59. Volume 34. Junho de 1997.

[9] Nair, R. "Effect of increasing chip density on the evolution of computer architectures”. Disponível em: http://www.research.ibm.com/journal/rd/462/nair.html. Último acesso em 18 de Novembro de 2006.

[10] Choi, K. “Dynamic Voltage and Frequency Scaling for energy-efficient system design”. Departamento de Automação e Sistemas. Universidade do Sul da Califórnia. Estados Unidos, 2005.

[11] Hsu, C., Feng, W. “A PowerAware RunTime System for HighPerformance Computing”. ACM/IEEE SC 2005 Conference (SC'05), 2005.

[12] Cpntools, versão 1.4.0. Disponível em http://wiki.daimi.au.dk/cpntools/cpntools.wiki. Último acesso em 13 de Abril de 2006.

[13] Oliveira Júnior, M. N., et al. “A Retargetable Environment for Power-Aware Code Evaluation: An Approach based on Coloured Petri Net”. Integrated Circuit and System Design, 15th International Workshop, PATMOS: 49, Leuven, Bélgica, Setembro de 2005.

[14] Oliveira Júnior, M. N., et al. “Towards a Software Power Cost Analysis Framework Using Colored Petri Net”. Integrated Circuit and System Design: Power and Timing Modeling, Optimization and Simulation, 14th International Workshop, PATMOS: 362-371, Santorini, Grécia, Setembro de 2004.

[15] Lima, R. M. F., et al. “Petri Nets tools integration through Eclipse”. OOPSLA workshop on Eclipse technology Exchange: 90-94, San Diego, Califórnia, Estados Unidos, 2005.

[16] Eclipse.org home. Disponível em http://www.eclipse.org. Último acesso em 11 de Junho de 2006.

[17] G. Yeap. Practical Low Power Digital VLSI Design. Kluwer Academic Publishers, 1998.

[18] The Lex & YACC Page. Disponível em http://dinosaur.compilertools.net. Último acesso em 18 de Abril de 2006.

[19] Bison – GNU Project. Disponível em http://gnu.org/software/bison. Último acesso em 18 de Abril de 2006.

[20] LINS, R. D. “Compiladores Modernos”. Editora Campus. Rio de Janeiro, 2001.

[21] Tenenbaum, A. M. et al. “Estrutura de Dados usando C”. 1. ed. Editora Pearson. São Paulo, 1995.

[22] Bernat, G., Colin, A., e Petters, S. M. “pWCET, a Tool for Probabilistic WCET Analysis of Realtime Systems”. WCET: 21–38, 2003.