PowerSC: Uma Extensão de SystemC para a Captura de Atividade de Transição

Felipe Vieira Klein

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

PowerSC: Uma Extensão de SystemC para a Captura de Atividade de Transição

Felipe Vieira Klein

Abril de 2005

Banca Examinadora:

- Rodolfo Jardim de Azevedo (Orientador)
- Luiz Cláudio Villar dos Santos
  INE-UFSC
- Paulo Cesar Centoducatte
  IC-UNICAMP
- Sandro Rigo (Suplente)
  IC-UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecário: Maria Júlia Milani Rodrigues – CRB8a / 2116

Klein, Felipe Vieira
K672p POWERSC: uma extensão de SystemC para a captura de atividade de transição -- Campinas, [S.P. :s.n.], 2005.

Orientadores : Rodolfo Jardim de Azevedo; Guido Araújo
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Sistemas embutidos de computador. 3. Estimativa de potência. I. Azevedo, Rodolfo Jardim de. II. Araújo, Guido. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Título em inglês: POWERSC: a SystemC extension aiming at the gathering of switching activity

Palavras-chave em inglês (Keywords): 1. Computer architecture. 2. Embedded computer systems. 3. Power estimation.

Área de concentração: Sistemas de computação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Rodolfo Jardim Azevedo (IC-UNICAMP)
Prof. Dr. Luiz Cláudio Villar dos Santos (INE-UFSC)
Prof. Dr. Paulo Cesar Centoducatte (IC-UNICAMP)

Data da defesa: 15/04/2005

PowerSC: Uma Extensão de SystemC para a Captura de Atividade de Transição

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Felipe Vieira Klein e aprovada pela Banca Examinadora.

Campinas, 15 de abril de 2005.

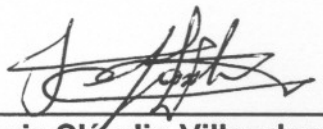
Rodolfo Jardim de Azevedo (Orientador)

Guido C. S. de Araújo (Co-orientador)
IC-UNICAMP

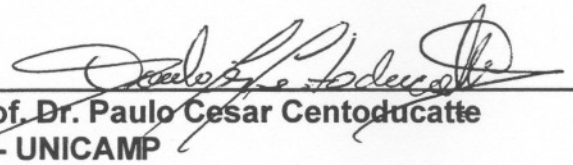
Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 15 de abril de 2005, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Luiz Cláudio Villar dos Santos
INE - UFSC



Prof. Dr. Paulo Cesar Centoducatte
IC - UNICAMP

Rodolfo Jardim de Azevedo

Prof. Dr. Rodolfo Jardim de Azevedo
IC - UNICAMP

"If A is success in life, then A equals x plus y plus z. Work is x; y is play; and z is keeping your mouth shut." *Albert Einstein (1879 - 1955)*

"Success usually comes to those who are too busy to be looking for it." *Henry David Thoreau (1817 - 1862)*

Resumo

Com a constante redução do tamanho dos transistores e o conseqüente aumento do número de transistores em um mesmo *chip*, a potência dissipada pelos circuitos digitais está aumentando exponencialmente. As implicações do aumento de potência vão desde o aumento de custo advindo de soluções elaboradas para o resfriamento do *chip* e da limitação crítica do tempo de bateria até a própria destruição do *chip*. Por estes motivos, o projeto de circuitos digitais visando a redução do consumo de potência têm se tornado um fator cada vez mais importante no fluxo de projeto – o chamado *low power design*. Esta dissertação de mestrado apresenta a PowerSC, uma biblioteca que estende as capacidades de SystemC, dando suporte à captura da atividade de transição de modelos em descrições de alto nível, em código C++. Além disso, propõe-se uma metodologia mais simples e transparente para o usuário, como uma alternativa à metodologia de uma ferramenta comercial. Outra contribuição deste trabalho é o algoritmo SMS, um algoritmo de monitoração eficiente, que consegue reduzir drasticamente o tempo de monitoração, com uma perda mínima de precisão. Os resultados experimentais obtidos mostram a factibilidade do uso de nossa abordagem para a captura efetiva da atividade de transição de modelos SystemC.

Abstract

With the ever-shrinking size of the transistors and the consequent growth in the number of transistors per chip, the power dissipated by digital circuits is raising exponentially. There are several implications of the increasing of power consumption, ranging from the higher cost per chip, resulting from elaborated cooling and packaging solutions, and the critical limitation of the battery's lifetime to the circuit failure. Thus, the design of integrated circuits aiming at the reduction of the power consumption has become an important role in the design flow – the so-called low power design. This master thesis introduces the PowerSC, a library that extends the capabilities of SystemC, enabling the capture of the switching activity of high-level description models, coded in C++. Moreover, a simpler and transparent methodology is proposed, alternatively to a methodology of a commercial tool. Another contribution of this thesis is the SMS algorithm, an efficient monitoring algorithm, which can dramatically reduce the monitoring time, with a minimal loss of accuracy. The experimental results show the feasibility of the using of our approach to the effective capture of switching activity from SystemC models.

Agradecimentos

Eu gostaria de agradecer à minha família e principalmente aos meus pais, Vitor Hugo Klein e Nires T. Vieira Klein, pelo apoio irrestrito e incondicional em toda a minha vida. Meu pai, o grande inspirador, que fez com que eu me apaixonasse por computação desde cedo. E minha mãe, que mesmo sem entender de *bits*, *bytes* e coisas afins, sempre me deu todo o suporte necessário, por saber que era isso o que eu realmente gostava de fazer.

Ao professor Rodolfo Azevedo, meu orientador, que esteve disponível em todos os momentos, expunha idéias, discutia problemas, e cuja contribuição foi crucial para o andamento deste trabalho. Ao professor Guido Araújo, meu co-orientador, que sempre auxiliou com idéias que ajudaram a melhorar a qualidade deste projeto.

Aos meus amigos e colegas de laboratório, em especial: Billo, Patrick, Marcos, Cleyton, Márcio (Juliato), Márcio (Oliveira), Bartho, Renon (magrão), Alexandro, Wesley, Valdiney, Borin. Bom, a lista é grande, mas, resumindo, a todos os meus amigos, presentes ou não.

E também, é claro, aos funcionários do Instituto de Computação e ao pessoal da portaria, principalmente ao "Seu Nélson", sempre bem-humorado.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Com a constante redução no tamanho dos transistores e o conseqüente aumento no número de transistores por *chip*, a potência dissipada pelos circuitos digitais está crescendo exponencialmente. Aliado a isto, a demanda por dispositivos portáteis, tais como PDAs, MP3 *players*, telefones celulares, entre outros, não pára de crescer. A cada nova geração destes produtos, mais funcionalidades são agregadas, aumentando sua complexidade e o consumo de energia.

Este aumento de complexidade foi previsto por Gordon Moore em meados da década de 60, no que se tornou a famosa "Lei de Moore", que estabeleceu que o número de transistores dobraria a aproximadamente cada 18 meses [1]. A figura 1.1 mostra o efeito desta observação na linha de processadores da Intel.

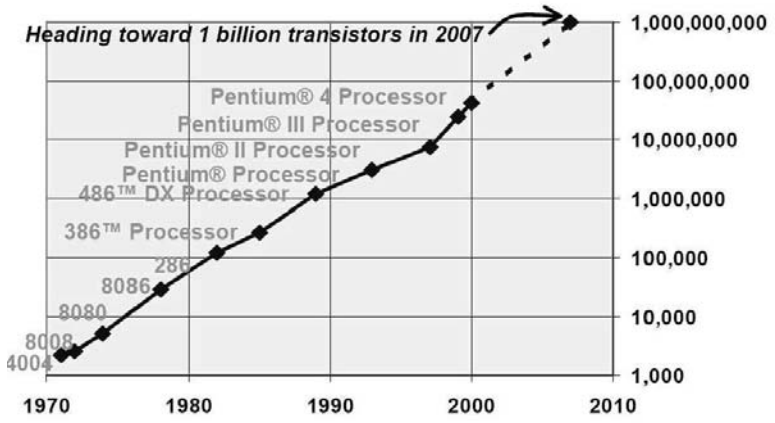


Figura 1.1: Lei de Moore

Entre as várias implicações do aumento do consumo de energia estão:

- **menor tempo de vida das baterias**: este é um fator de mercado importante, pois

pode definir a escolha do usuário por um produto ou outro, que tenha características semelhantes, mas que forneça uma maior autonomia.

- **redução do tempo de vida do circuito**: o aumento exponencial na densidade de potência ($\frac{W}{cm^2}$) simplesmente pode levar os circuitos à falência.
- **aumento do custo para o resfriamento do chip**: é outro fator importante de mercado que, inevitavelmente, ocasiona o aumento do custo do produto final.

Para ilustrar a implicação do aumento da densidade de potência, na figura 1.2 é mostrada uma extrapolação disto feita pela Intel.

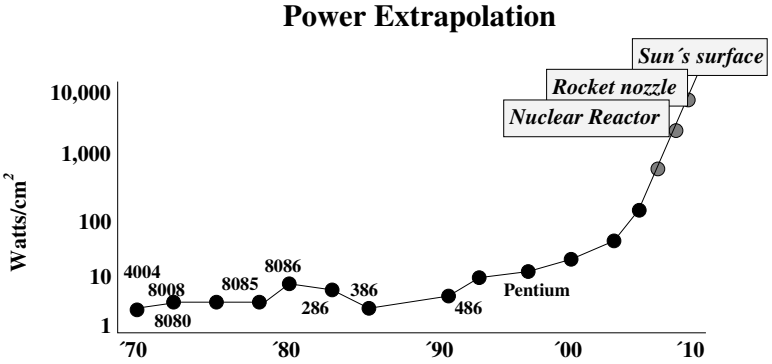


Figura 1.2: Extrapolação de Potência

A figura mostra a evolução da densidade de potência na linha específica de processadores da Intel, mas também demonstra a tendência atual no desenvolvimento de circuitos integrados de uma forma geral. Pat Gelsinger, vice-presidente da Intel, afirmou que se a tendência atual prevalecer e continuarmos na mesma direção, logo teremos um processador com 1.8 bilhões de transistores, mas também um gerador de calor com a intensidade de calor de um reator nuclear [2].

Os dados apresentados na tabela 1.1 fazem uma comparação direta de algumas características de um processador do começo da década de 90 com um atual. A última coluna da tabela mostra o fator de aumento de cada característica do processador mais novo em relação ao antigo. Como se pode ver, o número de transistores aumentou expressivamente, e o pico máximo de potência que era de 5W aumentou 20 vezes, chegando a 100W. Um outro fator importante mostrado na tabela é a freqüência dos processadores, que teve um aumento de 40 vezes.

Em outro gráfico (figura 1.3) da própria Intel, retirado de [3], é ilustrada a relação entre o aumento da dissipação de potência e o custo de resfriamento do chip. Este custo está associado, principalmente, ao encapsulamento do processador e a soluções mais elaboradas para o resfriamento do chip, como por exemplo, um melhor dissipador de calor.

Feature	**Intel486 (0.8μ)**	**Pentium 4 (0.18μ)**	**Factor**
Transistors	1.2M	42M	35x
Frequency	50MHz	2000MHz	40x
Max Peak Power	5W	100W	20x
Power/Transistor	4.2μW	2.4μW	0.6x
Power Density	6.8W/cm^2	46W/cm^2	7x
Max switches/Sec	60x10^{12}	84000x10^{12}	1400x

Tabela 1.1: μArchitecture Trend

Pelos dados apresentados, fica claro que o projeto de circuitos integrados visando a redução do consumo de potência se torna uma diretriz: o chamado *low power design*.

Porém, ao contrário de ferramentas para realizar estimativas de desempenho e área, que têm um grande leque de opções disponíveis tanto na indústria quanto no meio acadêmico, relativamente poucas opções estão disponíveis para se estimar o consumo de potência em níveis de abstração mais altos.

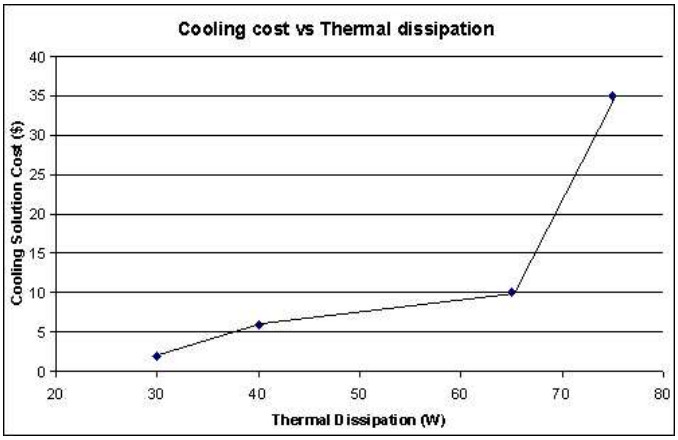


Figura 1.3: Relação entre dissipação de calor e custo para resfriamento do chip

Para os níveis mais baixos, como de transistor (*transistor-level*) e de portas lógicas (*gate-level*), há diversas opções de ferramentas de CAD[1], e.g. SPICE [4]. No entanto, estimar potência de circuitos médios e grandes nestes níveis pode ser uma tarefa excessivamente custosa em termos de tempo. Uma outra dificuldade é que estas ferramentas se localizam nos últimos estágios do ciclo de projeto, retardando a detecção de uma escolha de projeto mal-feita. Isso geraria novas rodadas completas no ciclo de desenvolvimento, até que as restrições de projeto fossem alcançadas, comprometendo o tempo de desenvolvimento.

[1]CAD – Computer-Aided Design

Por isso, conforme a complexidade do circuito aumenta, é necessário prover ao projetista um meio de subir nos níveis de abstração, onde estão as maiores oportunidades de otimização para potência [5]. Sabe-se que a precisão das técnicas de estimativa de potência em alto nível são menos apuradas que as em baixo nível. Porém, mesmo uma estimativa grosseira em alto nível pode economizar muito tempo de projeto, dado que **gargalos de consumo** podem ser detectados mais cedo, acelerando o processo de otimização.

As características mais importantes nas técnicas de alto nível são o fato de se localizarem no início do fluxo do projeto, e da precisão relativa com respeito a outras opções de projeto, bem como a velocidade com a qual se obtém tais estimativas.

1.1 Contribuições deste Trabalho

Este trabalho está inserido na área de projeto de sistemas digitais, mais especificamente em análise de potência de circuitos digitais em alto nível, e foi totalmente desenvolvido dentro do LSC[2] (Laboratório de Sistemas de Computação), do Instituto de Computação da UNICAMP.

As principais contribuições deste trabalho são resumidas abaixo:

- Desenvolvimento de uma biblioteca que estende uma linguagem de descrição de sistemas, dando suporte à captura de atividade de transição (*switching activity*) a partir de modelos numa descrição de alto nível, em código C++.
- Proposição de uma metodologia para análise de potência alternativa à de uma ferramenta comercial, para ser utilizada com a extensão desenvolvida. Esta metodologia, diferentemente da metodologia da ferramenta comercial, é simples e transparente, exigindo um esforço mínimo do projetista para seu uso.
- Superação de uma limitação da ferramenta comercial, incapaz de monitorar a atividade de transição de bancos de registradores.
- Concepção de um algoritmo de monitoração eficiente, capaz de reduzir o tempo necessário de monitoração durante a simulação do modelo de forma expressiva, com uma perda mínima da precisão.

1.2 Organização do Texto

O texto remanescente desta dissertação está organizado da seguinte forma: o capítulo 2 mostra, simplificadamente, como a potência é consumida em circuitos digitais atuais.

[2] http://www.lsc.ic.unicamp.br

O capítulo 3 faz uma revisão bibliográfica de diversos trabalhos relevantes na área de estimativa de consumo de potência.

No capítulo 4 é apresentada a biblioteca PowerSC, principal objeto desta dissertação, e sua metodologia, enquanto no capítulo 5 os resultados experimentais obtidos com a PowerSC são relatados.

O capítulo 6 encerra esta dissertação fazendo as considerações finais e, também, mostra as perspectivas de continuidade em trabalhos futuros.

Capítulo 2

Consumo de Potência

Precisamos inicialmente definir como a potência é dissipada em circuitos digitais e, para isso, será mostrado como isto ocorre em circuito CMOS[1] estático, que é a tecnologia utilizada na maioria dos circuitos digitais contemporâneos [6]. A figura 2.1 mostra os principais componentes de um transistor em tal tecnologia.

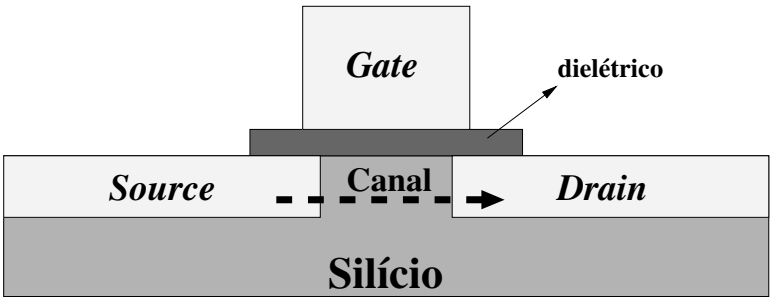


Figura 2.1: Principais componentes de um transistor CMOS estático

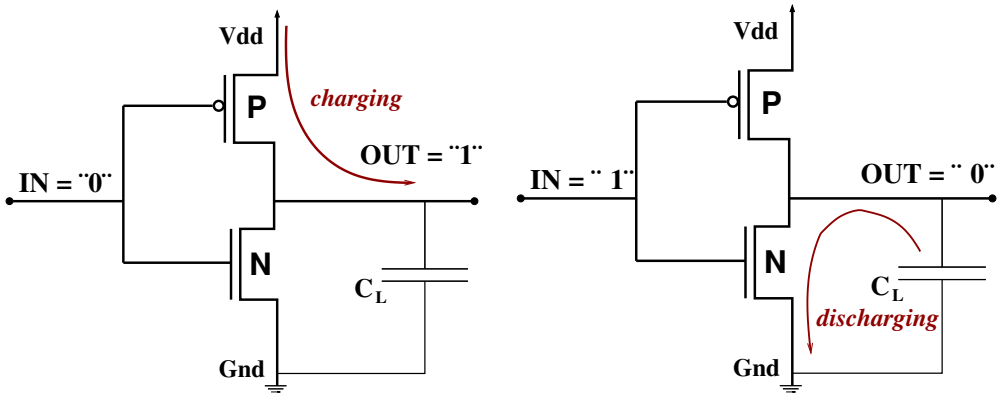
Estes componentes são o *gate*, o fonte (*source*) e o dreno (*drain*). Entre o fonte e o dreno há o **canal**[2], que é por onde a corrente deve passar quando o transistor estiver conduzindo. Por isso é chamado de canal de condução. Ao se aplicar certa tensão sobre o *gate* (dependente de tecnologia), isto faz com que cargas sejam induzidas no canal, permitindo a passagem de corrente.

Podemos dividir o consumo de potência em duas grandes categorias: *estática* e *dinâmica*. Potência dinâmica é a potência dissipada quando o circuito está ativo, é o consumo útil de energia. O consumo de potência em CMOS é ainda dominado pela potência dinâmica e está relacionado com a carga e descarga dos capacitores na saída

[1] CMOS – Complementary Metal Oxide Semiconductor

[2] Quando se fala em tecnologia de processo, .18μ, por exemplo, está se referindo a largura do *canal*

do gate [6]. Também é chamada de *switching power*, pois ocorre no momento em que o circuito está chaveando. Como este processo ocorre pode ser visto na figura 2.2.



(a) IN em 0 carrega C_L (b) IN em 1 descarrega C_L

Figura 2.2: Carga e descarga dos capacitores

Estas figuras mostram a mais simples das portas lógicas CMOS, um inversor. Esta porta lógica é composta por dois transistores: um tipo-P (parte superior da figura 2.2) e um tipo-N (parte inferior da figura 2.2).

Na figura 2.2(a) é mostrado como acontece a carga do capacitor na saída do *gate* (porta lógica): suponha que a entrada *IN* deste *gate* esteja inicialmente com o valor lógico 0. Assim o transistor inferior (tipo N) está desligado; conseqüentemente, o superior (tipo P) está ligado, ou seja, está conduzindo e, desta forma, a capacitância C_L em OUT é carregada, de V_{dd} para OUT.

Agora suponha que *IN* faça a transição 0→1. Isto causa o contrário, fazendo o transistor superior parar de conduzir e o inferior começar a conduzir, como é mostrado na figura 2.2(b). Com isso o capacitor na saída do *gate* é descarregado de OUT para *Gnd*. Existem algumas diferenças entre as transições 0→1 e 1→0 (não discutidas aqui), mas uma boa aproximação para o consumo total de potência dinâmica pode ser feita através da seguinte expressão:

$$P_{dyn} = C_L V_{dd}^2 f T R \tag{2.1}$$

onde C_L é a capacitância de carga do *gate*, V_{dd} é a tensão de alimentação, f é a freqüência de operação do circuito e TR é o *toggle rate*, ou seja, a taxa de transições lógicas por unidade de tempo (ex. $\frac{trans}{seg}$).

Uma característica importante que não pode ser ignorada é que, em projetos reais, a transição 0→1 (1→0) não ocorre instantaneamente. A inclinação que ocorre no sinal de entrada do *gate* gera uma indesejável conexão entre V_{dd} e *Gnd* por um curto período,

quando os transistores P e N estão conduzindo simultaneamente. Esse tipo de dissipação é chamado de potência de curto-circuito, ou *short-circuit power*. Isto é ilustrado na figura 2.3.

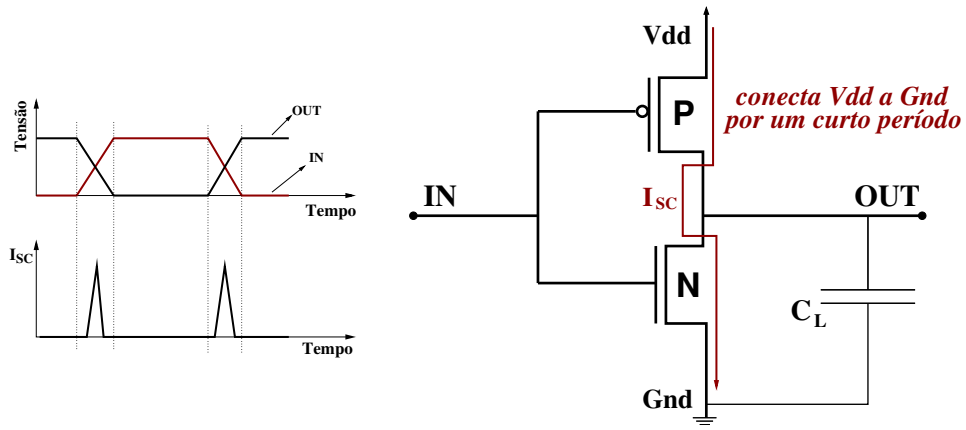


Figura 2.3: Curto-circuito: ambos transistores conduzem ao mesmo tempo

Para circuitos que tenham tempos baixos de transição, a contribuição de *short-circuit power* pode ser baixa. No entanto, para circuitos com tempos altos de transição a contribuição pode ser de até 30% de todo o consumo [7].

Potência estática é a potência dissipada quando o circuito não está chaveando, ou seja, quando o circuito está inativo. Pode-se dizer que esse consumo é inútil. Existem diversas origens para a dissipação estática e a maior parte é resultado do "vazamento" de corrente do fonte para o dreno, que é causado pelos reduzidos limiares de voltagem que impedem o transistor de desligar por completo. Há também o vazamento de corrente através do dielétrico do gate. Por essas razões, potência estática é comumente chamada de *leakage power*. Esta forma de consumo é muito dependente de tecnologia e, nos últimos anos, tem aumentado bastante a cada evolução do processo de fabricação. A redução no tamanho do transistores e o conseqüente aumento do número de transistores numa mesma área de silício também têm contribuido muito para isso (aumentando a relação $\frac{transistores}{área} \times \frac{potencia}{transistores}$).

A contribuição de cada uma das fontes de consumo de potência descritas aqui muda de acordo com a tecnologia, mas o quadro atual é este para CMOS (valores aproximados):

- **Switching Power** *(70%-90%)*
- **Leakage Power** *(5%-25%)*
- **Short-Circuit Power** *(5%)*

A tensão de alimentação (V_{dd}) vem diminuindo ao longo dos anos, mas, de qualquer forma, o consumo total de potência continua a aumentar. Este contínuo aumento de consumo é originado pelo aumento da freqüência de operação dos *chips*, a crescente capacitância e resistência na interconexão, bem como pela contribuição cada vez maior da *leakage power* no consumo total [8]. Contudo, a *switching power* continua sendo responsável pela maior parte do consumo de potência e, segundo o *roadmap* em [8], este panorama deve permanecer válido pelos próximos anos, o que instiga que mais esforços devem ser feitos com o objetivo de análise/otimização de *switching power*.

Capítulo 3

Trabalhos Relacionados

Neste capítulo será feita uma revisão bibliográfica sobre os trabalhos relevantes na área de estimativa de consumo de potência em circuitos digitais, mostrando o atual estado da arte das técnicas existentes. Como esta dissertação de mestrado está focada nos níveis mais altos de abstração, não serão discutidos aqui trabalhos voltados para os níveis mais baixos, como *gate-level* e *transistor-level*. Mais informações a respeito de alguns destes trabalhos podem ser encontrados em [4] e [9].

Primeiramente, é preciso definir o que são e quais são os níveis de abstração existentes. No nível mais baixo temos o *transistor-level* (ou *circuit-level*), que representa o circuito em termos de transistor, numa descrição pronta para ser enviada a *foundry*[1]. Acima deste está o chamado *gate-level*, que é representado através de portas lógicas, tais como AND, OR, NOT e XOR.

Sobre as técnicas que chamamos de **alto nível de abstração** (descritas a seguir) é bem sabido que sua precisão é menos apurada que nos níveis mais baixos, isto porque há muito menos informação a respeito do circuito final em suas descrições. Porém, o mais importante nestas técnicas não é o valor absoluto de consumo de potência fornecido por elas, e sim, valores relativos de consumo, possibilitando ao projetista efetuar *trade-offs* ainda nos estágios iniciais do desenvolvimento do circuito.

Uma boa categorização das técnicas de estimativa de potência em alto nível pode ser encontrada em [10], e usaremos esta mesma categorização para os trabalhos abordados nesta revisão bibliográfica. Os níveis de abstração aqui vistos podem ser divididos em: *arquitetural (RTL*[2]*)*, *comportamental*, *de instrução* e *de sistema*. O nível mais baixo dentre estes é o arquitetural.

[1] Fábrica onde os chips são produzidos

[2] *RTL – Register-Transfer Level*

3.1 Nível Arquitetural

As primitivas em RTL são os blocos funcionais, tais como somadores, registradores, SRAMs, controladores, etc. A grande dificuldade em se estimar potência em RTL vem da ausência de detalhes no nível de circuito (*floorplanning*, interconexão), ou mesmo de portas lógicas. As técnicas utilizadas neste nível podem ainda ser classificadas como métodos *analíticos* e *empíricos*.

3.1.1 Métodos Analíticos

A principal característica dos métodos analíticos é a tentativa de se fazer a relação do consumo de potência de uma descrição RTL com quantidades fundamentais que representem a capacitância física e a atividade do circuito.

Alguns trabalhos consideram somente a complexidade do circuito como uma boa medida inicial, o que ainda subdivide estes métodos em 2 modelos, os modelos baseados em complexidade (*complexity-based*) e os baseados em atividade (*activity-based*).

Modelos Baseados em Complexidade

A principal sustentação destes modelos está no fato de que a complexidade de um circuito pode ser descrita grosseiramente em termos de *gates* equivalentes. *Gates* equivalentes é o termo utilizado para informar o número aproximado de *gates* de referência necessários para a implementação de um bloco funcional qualquer. Por exemplo, poderíamos considerar como *gate* de referência um *NAND* de 2 entradas. Desta forma, o cálculo da estimativa da potência consumida por cada bloco funcional seria feita multiplicando-se a quantidade de potência dissipada por *gate* pelo número total de *gates* de referência. Esta técnica é utilizada pelo CES[3] [11]. A expressão básica utilizada em [11] para o cálculo da potência dissipada pelo circuito é:

$$P = \sum_{i \,\epsilon\, \{blocos\}} GE_i(E_{typ} + C_L^i V_{dd}^2) f A_{int}^i \tag{3.1}$$

onde GE_i é o número gates equivalentes para cada bloco funcional i sendo implementado, E_{typ} é o consumo médio de energia pelo bloco i, C_L^i a capacitância do *gate*, V_{dd} a tensão de alimentação, f é a freqüência de operação e A_{int}^i é a porcentagem média dos *gates* que chaveiam a cada ciclo para o bloco i.

Um problema desta abordagem é que apenas um *gate* é usado como referência para o consumo de todas as partes do circuito. Sendo assim, alguns itens importantes como estilos de circuito, estratégias de *clock* ou técnicas de *layout* não são levados em conta.

[3] CES – Chip Estimation System

Em outro trabalho [12], esta técnica foi melhorada com relação a este problema. A solução foi dividir o circuito em várias entidades (lógica, memória, interconexão e *clock*), e adaptar os cálculos para estimativa de potência para cada uma destas entidades, capturando as particularidades de cada uma. Com isso, foi possível analisar a distribuição de potência entre estas várias entidades, além de se obter melhores resultados.

Existem vantagens e desvantagens nos modelos baseados em complexidade. A necessidade de pouca informação pode ser vista como a sua principal vantagem. Os principais parâmetros necessários são: energia dissipada por *gate*, sua capacitância e o número de *gates* equivalentes do circuito. Em contrapartida, a principal desvantagem é que a modelagem da atividade é bastante imprecisa e geralmente fornecida pelo usuário. Cada bloco funcional terá uma atividade diferente em arquiteturas distintas, pois cada arquitetura tem uma dinâmica diferente, o que acaba gerando imprecisão na atividade do bloco e, consequentemente, na estimativa da potência dissipada pelo circuito.

Modelos Baseados em Atividade

Os modelos baseados em atividade resolvem alguns pontos do problema citado acima. Geralmente, os trabalhos destes modelos se baseiam no conceito de entropia, da teoria da informação, como medida de atividade do circuito [13, 14].

Em [13] é apresentada uma técnica para se estimar a atividade de chaveamento média dentro de um circuito combinacional, fornecidas somente as entradas/saídas de suas funções booleanas. Entropia é um conceito antigo e, anteriormente, a entropia associada à uma função booleana já foi utilizada para se prever a área de silício necessária para se implementar esta função, sem saber sua implementação em *gate-level* [15]. Para entender como é feita a relação entre entropia e consumo de potência, considere um circuito combinacional com N *gates* cujos nós de saída são denotados por $x_i, i = 1, 2, ..., N$. Assim, a potência média consumida por este circuito é representada por

$$P_{avg} = \frac{1}{2} V_{dd}^2 \sum_{i=1}^{N} C_i D(x_i) \tag{3.2}$$

onde C_i é a capacitância no nodo i e $D(x_i)$ é a densidade de transições (número médio de transições lógicas por segundo) em x_i. O autor então observa que a potência é proporcional ao produto da capacitância física e a atividade, e fazendo algumas aproximações a equação anterior é simplificada:

$$P_{avg} \propto A \times D \tag{3.3}$$

onde A é uma estimativa da área do circuito, utilizada para representar a capacitância física e D é a densidade de transições média do circuito. Sua metodologia então consiste em executar uma simulação RTL do circuito de forma a se medir as entropias de entrada e

saída dos blocos funcionais e, após isso, utilizando-se destes dados, estimar D, A, e assim calcular a potência média consumida pelo circuito.

Existem alguns problemas nesta abordagem. Por exemplo, pelas simplificações existentes nas expressões mostradas, há a suposição implícita de que a capacitância é uniformemente distribuída por toda a área do circuito, o que não é verdade. Informações de temporização também não são levadas em conta nos cálculos da entropia.

Mas, reiterando, a vantagem destes modelos é a necessidade de pouca informação. Embora a precisão absoluta se mostre bem limitada, para comparações relativas essa precisão parece ser mais apurada. Esta abordagem é considerada como estando em sua "infância", e muito trabalho deve ser feito para que estas técnicas se provem realmente utéis.

3.1.2 Métodos Empíricos

Nas técnicas do método analítico, mostradas anteriormente, existe a vantagem de que pouca informação é necessária para a realização das estimativas de consumo de potência. Por conseqüência disto, estas técnicas acabam tendo uma relação fraca com o *hardware* real.

Ao contrário disto, no método empírico, essa relação é muito mais forte. Ao invés de se tentar relacionar o consumo dos componentes RTL com parâmetros fundamentais, a idéia aqui é de se medir o consumo de potência de implementações existentes de blocos funcionais e criar modelos a partir destas medidas, gerando o que é chamado de *macro-modelo*. A **medição** à qual nos referimos aqui é feita através de métodos tradicionais de baixo nível, como *transistor-level* e *gate-level*, dependendo da precisão desejada para o macromodelo. Abordagens baseadas em bibliotecas de tecnologia são mais adequadas para estes métodos, mas isto não é necessariamente um pré-requisito.

A maioria das abordagens utilizadas em ferramentas atuais se utilizam de métodos empíricos. Podemos dividir estes métodos em três categorias: os de atividade de sinal fixa (*fixed-activity*), os que levam em conta dados estatísticos das atividades dos sinais de entrada (*activity-sensitive*) e, por último, temos os métodos que são baseados nas transições das entradas (*transition-sensitive*)·

Modelos Empíricos de Atividade Fixa

A primeira técnica apresentada para este modelo foi a PFA[4] [16], na qual os modelos de energia são caracterizados em termos de parâmetros de complexidade e uma constante de proporcionalidade *PFA*. No artigo, os autores mostram a macro-modelagem de multiplicadores, memórias e controladores de E/S e, com isso, sua técnica pode ser ampliada para

[4]PFA – Power Factor Approximation

caracterizar individualmente cada elemento de uma bilbioteca. A estimativa da potência consumida, usando esta técnica, é feita baseada na expressão 3.4 abaixo:

$$P = \sum_{i \, \epsilon \, \{blocos\}} \kappa_i G_i f_i \tag{3.4}$$

onde κ_i é a constante PFA de proporcionalidade (extraída empiricamente de implementações anteriores do bloco), G_i é a medida da complexidade do bloco i e f_i é a freqüência com a qual o bloco é ativado. Para um multiplicador, por exemplo, a complexidade do bloco poderia ser relacionada com o quadrado do tamanho de palavra de sua entrada (n^2), e a freqüência de multiplicações no bloco definiria f_i.

Analisando a expressão mostrada, podemos ver que há a suposição implícita de que os valores de entrada não afetam a atividade de chaveamento interno ao bloco. Com isso, dependendo dos vetores de entrada utilizados, erros substancialmente grandes podem ocorrer no cálculo da estimativa de potência consumida pelo circuito.

Modelos Empíricos Sensíveis à Atividade

Para superar a fraqueza dos modelos de atividade fixa mostrados anteriormente, foram criados os modelos empíricos sensíveis à atividade (*activity-sensitive*). A grande maioria dos modelos utilizados atualmente encaixam-se nesta categoria. Nestes modelos, o que se faz é levar em consideração a influência de dados estatísticos da atividade dos dados nas estimativas de consumo de potência. Uma forma comum usada para armazenar as propriedades de potência destes modelos é através de tabelas, utilizando-se LUTs[5], por exemplo. Diversos trabalhos são facilmente encontrados na literatura [17, 18, 19, 20, 21].

Em [17] foi desenvolvida uma ferramenta chamada SPA, na qual várias entidadas são modeladas separadamente (*datapath*, memória, unidade de controle e interconexão). Cada uma destas entidades é analisada com relação à sua complexidade. Após isso, é executada uma simulação RTL com entradas típicas fornecidas pelo usuário e, durante esta simulação, a atividade dos sinais e entidades do projeto é monitorada. Usando-se destes dados é feita uma análise da atividade para a geração de estatísticas (conceito que eles chamam de *activity profiling*), que então são usadas para alimentar os modelos de cada entidade, que levam em conta tanto a atividade quanto a complexidade. Feito isso, e realizados os cálculos para cada entidade, estes dados são unidos e a potência consumida pelo circuito é finalmente estimada.

A técnica apresentada em [19] é análoga a anterior, porém, como uma contribuição adicional, a atividade de *glitching* é levada em conta. Em seu trabalho, os autores mostram que, em alguns casos, a desconsideração dos *glitches* pode levar a erros substanciais nas

[5]LUT - Look-Up Table

estimativas. A metodologia deles é dividida em 3 fases: primeiro executa-se uma simulação RTL com entradas típicas, para gerar as estatísticas dos vários sinais do circuito. Na segunda fase, eles usam estas estatísticas para estimar a atividade de *glitching* no circuito RTL. Então, na terceira fase de sua metodologia, a estimativa de *glitching* juntamente com as estatísticas da primeira fase são usadas para calcular a potência consumida em cada bloco do circuito, de uma maneira semelhante a feita em [17].

Modelos Empíricos Sensíveis à Transição

A terceira abordagem dos modelos empíricos, diferentemente da anterior, não se baseia em estatísticas das entradas, e sim nas transições das entradas. Isto foi proposto inicialmente em [22], onde supõe-se que exista um modelo de energia para cada unidade funcional, isto é, uma tabela contendo a potência consumida para cada transição na entrada. Detectando-se transições na entrada muito parecidas e padrões de energia, consegue-se diminuir o tamanho da tabela através de *clustering*. Um simulador de energia totalmente baseado nesta técnica é o SimplePower [23], que será discutido na seção 3.3.

Em [24], são levadas em conta apenas o chaveamento das entradas, e as LUTs são caracterizadas de forma a armazenarem a potência consumida pelas transições das entradas. A estimativa de potência é feita usando uma combinação de duas alternativas com relação ao chaveamento das entradas: *word-level* (relaciona a energia com o número de *bits* chaveando simultaneamente) e *bit-level* (relaciona a energia ao chaveamento dos *bits* individualmente). Isso, combinado com alguns fatores de ajuste para cada alternativa, mostrou um bom resultado nas estimativas.

Nos métodos empíricos mostrados, há a vantagem de existir uma relação forte com o *hardware* real e, sendo assim, a precisão absoluta obtida é superior à dos métodos analíticos. O que pode ser apontado como uma desvantagem nestes métodos é o tempo que os projetistas gastam para caracterizar um modelo, apesar de isso ser feito apenas uma vez, na maioria dos casos.

3.2 Nível Comportametal

Como visto, as técnicas do nível arquitetural apresentam várias simplificações e suposições na modelagem da potência consumida, devido a limitação de informação disponível, o que limita a precisão dos resultados obtidos. E quanto maior é o nível de abstração usado na descrição do modelo, maior é a dificuldade de se obter uma boa estimativa de potência pelas razões já mencionadas.

Portanto, o que se faz no nível comportamental para superar estas dificuldades e produzir estimativas é tentar mapear as descrições comportamentais para as técnicas RTL

empíricas (seção 3.1.2), supondo certos estilos arquiteturais. Entre os diversos pontos que são desconhecidos neste nível estão: configuração de memória, arquitetura do barramento, comprimento médio dos fios, número de transações no barramento, complexidade da unidade de controle, entre muitos outros.

As técnicas comportamentais podem ser divididas em duas vertentes: predição estática (*static-activity prediction*) e predição dinâmica (*dynamic-activity prediction*).

3.2.1 Predição Estática

A tarefa principal na predição estática é estimar a freqüência de acesso aos diferentes recursos de hardware, analisando a descrição comportamental da função sendo implementada, que pode ser em C, Verilog, VHDL[6], ou até mesmo o próprio CDFG[7]. Uma abordagem que usa predição estática é o HYPER-LP [25]. A estimativa de potência deste trabalho é aproximada usando a expressão 3.5:

$$P = \sum_{r \,\epsilon\, \{todos\ recursos\}} f_r C_r V_{dd}^2 \tag{3.5}$$

onde f_r é a freqüência de acesso ao recurso r, determinada pela ánalise estática da função sendo implementada. C_r é a capacitância do recurso r, o qual foi obtida através de modelos arquiteturais empíricos de atividade fixa (mostrados anteriormente). Por exemplo, foram utilizados 46 projetos diferentes para o construir modelo da unidade de controle neste trabalho. Embora, para valores absolutos, estas técnicas tenham se mostrado limitadas, resultados aceitáveis foram obtidos ao se considerar valores relativos. A principal vantagem da predição estática é a velocidade da análise.

3.2.2 Predição Dinâmica

Na predição dinâmica, as freqüências de ativação são obtidas através da execução de uma simulação da descrição comportamental, sendo as entradas fornecidas pelo usuário. Após a obtenção destas informações, os dados são ligados a um modelo análogo ao de predição estática.

A vantagem aqui é que as dependências de dados que não conseguem ser capturadas na predição estática são facilmente obtidas, pois a execução do comportamento é que as define (e portanto as freqüências obtidas são mais confiáveis). A desvantagem é que, por causa desta simulação e da necessidade do usuário fornecer as entradas, esta técnica é um pouco mais lenta e trabalhosa. Como exemplo da predição dinâmica temos o *Power-Profiler* [26].

[6]VHDL – VHSIC Hardware Description Language
[7]CDFG – Control-Dataflow Graph

3.3 Nível de Instrução

No nível de intrução, o objetivo é capturar o comportamento (com relação ao consumo de potência) do hardware, durante a execução de suas instruções. Por isso, as técnicas deste nível utilizam implementações de hardware dedicadas, tais como os bem conhecidos processadores de uso geral e os DSPs[8]. Uma outra característica é que elas são baseadas nas técnicas de macro-modelagem empíricas mostradas na seção 3.1.2.

Em [27], a estratégia é a seguinte: cada instrução do processador alvo é colocada num laço e executada neste processador. Durante essa execução, os componentes do processador são monitorados, e a potência consumida por cada instrução é registrada e armazenada numa tabela de *custos-base*. Após isso, eles também consideram o que eles chamam de efeitos inter-instrução, vindos do fato que a potência consumida pelas instruções não é totalmente independente. Como exemplo, considere uma instrução I_j qualquer de um programa. As instruções I_{j-1} e I_{j+1} têm efeito sobre a dissipação de I_j, pois diferentes partes do circuito são ativadas com sua execução, por isso a necessidade desta análise.

Um trabalho que se baseia nas idéias apresentadas em [27] é a chamada plataforma SEA [28]. Em sua proposta, eles refinam a técnica anterior, levando em conta a contribuição das variações de dados nas instruções na potência, além de diferenciar, no *pipeline*, entre ciclos de execução e ciclos em *stall*, o que se mostrou benéfico para as estimativas. O processador utilizado por eles, para validar seu modelo, foi o MicroSparcIIep, que é um *IP*[9] *core* em RTL sintetizável, disponível em dominio público. A entrada para a SEA é um código-objeto de um programa escrito em C, por exemplo. Então, usa-se este binário em um simulador de conjunto de instruções (ISS[10]) para alimentar o ambiente com *traces* de instruções com informações de temporização. Um banco de dados contendo os modelos de potência foi gerado uma vez, através da simulação do processador em questão em conjunto com uma ferramenta comercial de estimativa de potência. Usando este banco de dados, a ferramenta acessa os modelos de potência capturando seqüências de instruções, dependências de dados, e efeitos de pipeline, gerando suas estimativas e estatísticas de consumo de potência.

Outro trabalho no nível de instrução é o SimplePower [23], no qual é usado um *core* RTL de um processador com uma arquitetura de pipeline de 5 estágios, com um subconjunto do conjunto de instruções do MIPS. A técnica usada aqui para estimar o consumo de potência é diferente de [27] e [28], pois é baseada em um modelo empírico sensível à transição, mostrado na seção 3.1.2. O fluxo desta ferramenta começa usando um compilador próprio, que gera executáveis da arquitetura SimplePower. Então, este executável

[8] DSP – Digital Signal Processor
[9] IP –Intellectual Property
[10] ISS – Instruction Set Simulator

é simulado, provendo, ciclo a ciclo, estimativas de energia e estatísticas de capacitância para o *datapath*, memória e barramentos *on-chip*. O SimplePower não captura ainda informações de potência consumida para algumas entidades do processador, como a unidade de controle.

3.4 Nível de Sistema

Este é o mais alto dentre os níveis de abstração mostrados aqui, e as técnicas deste nível são voltadas para os chamados SoCs[11]. Como se pode presumir, há menos informação disponível que nas abordagens mostradas anteriormente. Os componentes do sistema incluem desde partes análogicas, digitais e *mixed-signal* até partes eletromecânicas. A precisão absoluta obtida com as técnicas deste nível é baixa, mas o que se faz atualmente é usá-las para se obter uma noção inicial (mesmo que grosseira) do consumo do SoC como um todo. A principal utilidade é identificar os **gargalos** de consumo de potência, ajudando o projetista a particionar o sistema de uma forma mais adequada.

As técnicas para estimativa de potência utilizadas aqui são derivadas das técnicas empíricas. Uma ferramenta web chamada PowerPlay [29] foi desenvolvida para análise de potência em nível de sistema. A abordagem deles é chamada de *spreadsheet-like*, composta por um conjunto de fórmulas de potência simplificadas. Essas fórmulas são usadas com o intuito de se obter consumo médio de potência através de parâmetros estimados fornecidos, como tamanho e tipo de bloco lógico, capacitâncias, atividade de chaveamento, número de acessos ao recurso de *hardware*, freqüência, etc. Esta ferramenta foi utilizada com sucesso para a modelagem de potência de um terminal multimídia, comprovando a sua usabilidade. Em [30], é apresentada uma técnica que estende a técnica anterior, aplicando também o uso de métodos formais para o cálculo do consumo de potência de todo o SoC.

3.5 Ferramentas de CAD Comerciais

Embora não haja um vasto conjunto de ferramentas comerciais voltadas para estimativa de potência em alto nível, podemos citar alguns exemplos de ferramentas que permitem a análise de circuitos no nível arquitetural, como XPower [31], Power Compiler [7] e PowerTheater [32].

[11] SoC – System-On-Chip

3.5.1 XPower

A ferramenta XPower é propriedade da empresa Xilinx, faz parte do seu pacote de ferramentas de CAD, e é utilizada para fazer estimativas de consumo potência de suas FPGAs[12]

O cálculo da estimativa de potência é feito baseado na observação de que a potência dinâmica dissipada em circuitos CMOS se dá principalmente (mas não somente) pela atividade de chaveamento. A ferramenta tem um modelo de capacitância para vários elementos da FPGA, como LUT, *flip-flop*, BRAM e segmento de roteamento. A combinação de vários dados fornecidos pelo usuário (taxa de atividade, freqüência de operação) com dados específicos de tecnologia (capacitância, potência estática e outros) é usada para se obter uma estimativa do consumo de potência.

De acordo com a documentação da ferramenta, a potência consumida (em mW) por cada elemento de chaveamento é dado pela seguinte expressão:

$$P = CV^2Ef1000 \tag{3.6}$$

onde C é a capacitância para o projeto do usuário, e específica de acordo com a FPGA utilizada, V é a tensão determinada para a FPGA, F é a freqüência de operação em Hertz, e finalmente, E é a atividade de chaveamento (número médio de transições por ciclo do relógio).

Existem três maneiras para entrar com a informação de atividade de chaveamento nesta ferramenta, que pode ser através de uma simulação feita pelo usuário, na qual um arquivo VCD[13] é gerado com os valores dos sinais do projeto que foram monitorados. Outra forma é o usuário entrar com a taxa de atividade de chaveamento manualmente, e a última é deixar a ferramenta assumir um valor padrão global. É importante ressaltar que a qualidade na obtenção da taxa de atividade determina a qualidade da estimativa gerada.

3.5.2 Power Compiler

O Power Compiler é uma ferramenta da Synopsys, que permite a análise e otimização dos projetos do usuário com relação ao consumo de potência e, trabalhando em conjunto com o Design Compiler (também da Synopsys), provê otimizações de temporização e área.

O Power Compiler consegue fazer análise e otimização de potência tanto para FPGAs quanto para ASICs[14], bastando apenas que a biblioteca de tecnologia caracterizada para

[12]FPGA – Field-Programmable Gate Array
[13]VCD – Variable-Change Dump
[14]ASIC – Application-Specific Integrated Circuit

potência (e suportada pelas ferramentas da Synopsys) seja fornecida. Os níveis de abstração suportados são o RTL e o *gate-level*. Uma das otimizações aplicadas por esta ferramenta é o chamado *clock gating*, que consiste basicamente em inibir o *clock* nas regiões do circuito onde seu uso é desnecessário.

No Power Compiler, o consumo de potência é modelado por três componentes: potência estática (mencionada no capítulo 2), potência de chaveamento (transições lógicas ocorridas nas saídas das células[15]) e potência interna. A potência de chaveamento é obtida através de uma simulação (RTL ou *gate-level*, que é armazenada num arquivo e usado como entrada da ferramenta. A potência interna se refere à potência dissipada dentro dos limites das células da biblioteca (p. ex., um multiplexador 8x1) e devidamente caracterizados dentro da biblioteca de tecnologia.

Diversos tipos de relatórios e estatísticas de consumo de potência são mostrados pela ferramenta, além das otimizações efetuadas.

3.5.3 PowerTheater

De propriedade da Sequence Design, o PowerTheater [32] (antigamente chamado WattWatcher) é um conjunto de ferramentas que permite a análise e otimização de qualquer circuito tanto em RTL quanto em *gate-level*, além do suporte ao que eles chamam de depuração de atividade para potência.

Como o Power Compiler, o Power Theater obtém informações de atividade de chaveamento através de simuladores como ModelSim [33], entre outros. Como é uma ferramenta comercial, muitos dos detalhes internos são desconhecidos, mas sabe-se que sua modelagem de potência é baseada nos métodos empíricos sensíveis à atividade (veja a seção 3.1.2) e que, em seus modelos, a contribuição de *glitching* é levada em conta. Sua metodologia de análise de potência inclui modelos para memória (RAM, ROM), E/S, *clocks*, *datapath* e unidade de controle. A plataforma SEA [28] mencionada anteriormente utiliza esta ferramenta para a validação de seus resultados.

Neste capítulo foi feita uma revisão bibliográfica das técnicas de estimativa de potência de diversos níveis de abstração, dentre os quais estão os níveis de sistema, comportamental, de instrução e arquitetural. A PowerSC, foco deste trabalho, está caracterizada dentro do nível arquitetural, e ela é apresentada a seguir, no capítulo 4.

[15] Blocos definidos na biblioteca de tecnologia

Capítulo 4

PowerSC

Neste capítulo será apresentada a biblioteca *PowerSC*, o principal objeto desta dissertação. A PowerSC é uma extensão da linguagem SystemC [34, 35], a qual será brevemente apresentada na seção 4.1. Em seguida, na seção 4.2, será mostrada como a atividade de transição é capturada, além de apresentar a metodologia de uma ferramenta comercial, e nossa proposta alternativa. Na seção 4.3 será apresentada a biblioteca PowerSC e detalhes de sua implementação. O algoritmo proposto é apresentado na seção 4.4, onde são mostrados também os possíveis tipos de simulação com a PowerSC. A metodologia proposta é detalhada na seção 4.5.

4.1 SystemC

SystemC é um conjunto de classes em C++ [36] e uma metodologia para uso destas classes, ligados a um núcleo de simulação. Todos estes componentes juntos permitem a modelagem efetiva de sistemas. Pode-se dizer que SystemC estende a linguagem C++, e dentre os recursos fornecidos destacam-se:

- comportamento concorrente e reativo;
- noção de operações sequenciadas no tempo (*hardware timing*);
- tipos de dados para descrever *hardware*;
- noção de *delta cycles*;
- possibilidade de desenvolvimento de modelos com precisão de ciclos e
- suporte à simulação dirigida por eventos.

Com todos estes recursos, o projetista consegue obter uma especificação executável de seu sistema. Uma especificação executável nada mais é do que uma descrição de um sistema em SystemC, o qual é código 100% compatível com C++ e, portanto, pode ser compilado com qualquer compilador C++, exibindo o mesmo comportamento que o sistema quando executado. Na figura 4.1 pode-se ver como está definida a arquitetura de SystemC.

Figura 4.1: Arquitetura da Linguagem SystemC

Os blocos destacados com a cor cinza formam o que se chama de núcleo padrão da linguagem SystemC e qualquer implementação de SystemC deve seguir estes padrões por questões de compatibilidade. A partir deste ponto é pressuposto que o leitor tenha algum conhecimento de como usar o SystemC. Para maiores informações sobre como utilizar SystemC para modelar sistemas ou mesmo sobre a sua especificação, favor se referir a [34] e [35].

4.2 Capturando a Atividade de Transição

Existem dois formatos de arquivo principais que são utilizados para a captura de atividade de transição[1]: VCD e SAIF[2].

[1]Do inglês, switching activity
[2]SAIF – Switching Activity Interchange Format

O arquivo VCD é gerado durante a simulação e provê taxas de transição (*toggle rates*) dos sinais do projeto. O formato é composto da escala de tempo utilizada, definições de escopo e tipo de variável sendo registrada, e também das mudanças nos valores das variáveis a cada incremento da simulação. SystemC suporta este formato de arquivo. No entanto, a seleção dos elementos a serem monitorados tem que ser feita manualmente. Outro problema do VCD é que, para grandes tempos de simulação, o arquivo gerado se torna muito grande e, facilmente, chega aos *gigabytes*.

O formato SAIF é complementar ao VCD e foi criado pela Synopsys com o objetivo de padronizar um formato para potência. O SAIF é muito mais compacto e não cresce em tamanho durante a simulação. O formato é basicamente composto do número de transições (*toggle count*) e da probabilidade de estado, que é o tempo no qual um sinal fica num certo estado lógico. Em contraste ao VCD, a informação contida no SAIF já está processada e pronta para ser anotada no projeto.

4.2.1 Fluxo de Ferramentas de CAD para Potência

Como exemplo de um fluxo de ferramentas, utilizaremos o Power Compiler, da Synopsys, que foi descrito na seção 3.5. À esquerda da linha tracejada na figura 4.2 pode ser visto o fluxo utilizado pelo Power Compiler para análise de consumo de potência de modelos SystemC RTL. O processo mostrado na figura é o seguinte: primeiro, o modelo SystemC

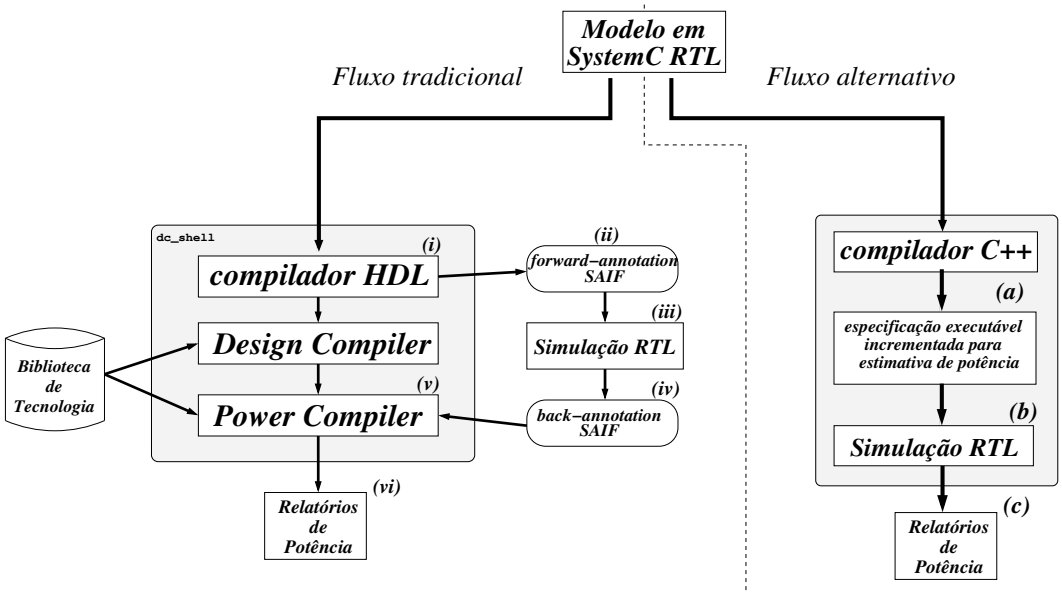


Figura 4.2: Dois fluxos de ferramentas para estimativa de potência

RTL é convertido para uma descrição em Verilog ou VHDL *(i)* e então compilado para

uma representação interna independente de tecnologia. Depois disso, um arquivo SAIF para *forward-annotation* é gerado a partir do modelo *(ii)*, com os elementos a serem monitorados durante a simulação. Feito isso, uma simulação RTL é executada usando este arquivo *(iii)* como entrada e, como sua saída, outro arquivo é gerado, o SAIF para *back-annotation (iv)*, que é usado pelo Power Compiler, junto com a informação extraída da biblioteca de tecnologia *(v)*, para computar a potência média consumida e gerar os relatórios com as estatísticas de consumo *(vi)*. A Synopsys disponibiliza uma *interface* que permite ao simulador ler e escrever diretamente os arquivos SAIF (tanto *forward-* quanto *back-annotation*) e diversos simuladores são suportados como, por exemplo, o bem conhecido ModelSim [33], que pode ser conectado a ela. Mas é importante dizer que esta *interface* suporta somente Verilog e VHDL e, por este motivo, o primeiro passo da figura é necessário. Maiores detalhes sobre o funcionamento do Power Compiler podem ser obtidos em [7].

Este fluxo tradicional utilizado pelo Power Compiler requer diversas ferramentas de diferentes fabricantes e muitos passos, o que pode transformar o processo de estimar potência numa tarefa lenta e complicada. Alternativamente a isto, o fluxo de ferramentas que propomos juntamente com a PowerSC é visto à direita da linha tracejada na figura 4.2. Este fluxo é mais simples que o anterior, dado que necessita de menos passos, e de ferramentas é necessário somente um compilador padrão de C++, responsável por gerar a especificação executável do modelo em SystemC, que é incrementada para capturar a atividade de transição *(a)*. Após isso, a especificação incrementada é executada, rodando uma simulação RTL *(b)*, onde a atividade de transição é capturada. Terminada a simulação, relatórios de estatísticas são produzidos *(c)*, encerrando o fluxo. Este processo vai ser explicado nas próximas seções. Ambos os fluxos mostrados monitoram a atividade de transição dos elementos *synthesis-invariant* do modelo, por exemplo, entradas primárias, elementos seqüenciais, *black boxes* e portas hierárquicas.

4.3 A Biblioteca PowerSC

A biblioteca PowerSC é composta por um conjunto de classes que estende as capacidades da linguagem SystemC. O objetivo principal da PowerSC é fornecer ao usuário uma maneira simples e transparente de obter informações sobre a atividade de transição, a partir de descrições SystemC RTL, de modo a reduzir os passos necessários para tal fim. Esta informação capturada pode então ser usada para estimar a potência consumida pelo modelo através de técnicas similares àquelas apresentadas no capítulo 3.

Será mostrado que, usando PowerSC, é possível obter a atividade de transição rapidamente, transparentemente, usando como ferramentas somente o SystemC e um compilador padrão de C++. A figura 4.3 mostra a abrangência da extensão feita pela PowerSC.

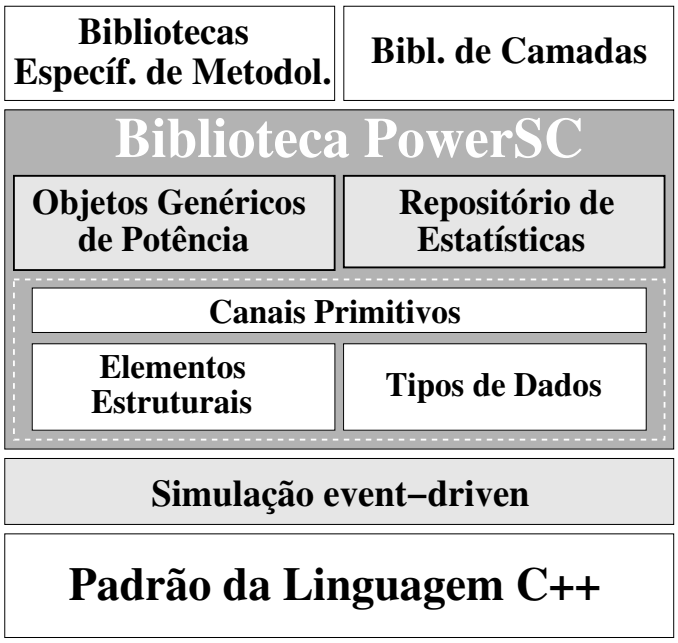


Figura 4.3: PowerSC: extensão do SystemC

Comparando a figura 4.3 com a 4.1, pode-se ver que foram estendidos três componentes da arquitetura do SystemC:

- elementos estruturais, como portas e *interfaces*;
- tipos de dados, desde tipos de lógica de 4 valores ('0', '1', 'X' e 'Z'), representações de valores inteiros até tipos de dados que representam *bits* e vetores de *bits* e
- canais primitivos, basicamente os sinais.

É importante ressaltar que as capacidades do SystemC foram aumentadas, e não restringidas a um certo domínio. Desta forma, os modelos em SystemC compilados com a PowerSC têm todas suas funcionalidades mantidas.

4.3.1 Implementação

A PowerSC habilita o projetista a capturar a atividade de transição de modelos em SystemC RTL. Isto foi feito especializando-se cada uma das classes relevantes (tipos de dados, sinais, portas, etc), de forma a monitorar seus valores durante a simulação e, para permitir a impressão de relatórios detalhados com toda a informação capturada.

Para ilustrar o funcionamento da PowerSC e como ela pode ser usada para incrementar um modelo SystemC, usaremos um exemplo de uma máquina de estados simples (fig. 4.4).

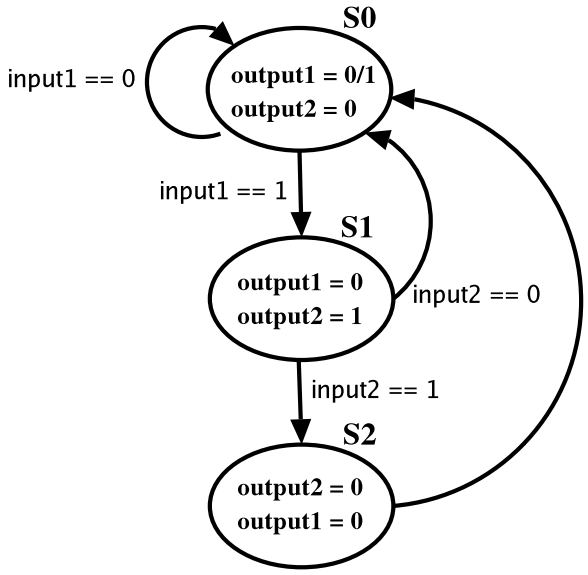


Figura 4.4: Uma máquina de estados simples

O modelo em SystemC equivalente à maquina de estados da figura está descrito nas figuras 4.5 e 4.6, mostrando os arquivos de cabeçalho e implementação, respectivamente.

A biblioteca é composta basicamente por três conjuntos de classes distintos, que nomeamos da seguinte forma: ***power object***, ***SystemC augmented*** e ***power object database***.

Power Object Classes

O primeiro conjunto, representado na figura 4.3 pelo bloco **Objetos Genéricos de Potência**, consiste das classes que contém propriedades e comportamento gerais relacionados com a captura da atividade de transição.

Os objetos das classes deste bloco armazenam informações sobre a atividade de transição capturada, probabilidade de estado, e estatísticas de consumo associados com o objeto em questão. Além disso, eles têm a capacidade de atualizar automaticamente estas informações, sempre que uma mudança ocorra durante a simulação. Na próxima subseção ficará mais claro como estas classes funcionam, mostrando a sua inter-relação.

SystemC Augmented Classes

O segundo conjunto, representado na figura 4.3 pelos blocos entre as linhas tracejadas, é composto pelas classes relevantes ao modelo do SystemC propriamente dito. Utilizando-se do recurso de múltipla herança de C++, as classes do SystemC deste conjunto foram

```
1 // header file: fsm.h
2 #include <systemc.h>
3
4 // An example of a finite-state machine
5 SC_MODULE(fsm) {
6   // input ports
7   sc_in_clk clk;
8   sc_in<sc_bit> reset;
9   sc_in<sc_bit> input1, input2;
10
11  // output ports
12  sc_out<sc_bit> output1, output2;
13
14  // signals
15  sc_signal<sc_uint<2> > state, next_state;
16
17  // processes
18  void ns_logic();
19  void output_logic();
20  void update_state();
21
22  // enumerate states
23  enum state_t { S0, S1, S2 };
24
25  SC_CTOR(fsm) {
26    SC_METHOD(update_state);
27    sensitive_pos << clk;
28
29    SC_METHOD(ns_logic);
30    sensitive << state << input1 << input2;
31
32    SC_METHOD(output_logic);
33    sensitive << state << input1 << input2;
34  }
35 };
```

Figura 4.5: Modelo exemplo em SystemC

```
1 // implementation file: fsm.cpp
2 #include "fsm.h"
3
4 void fsm::update_state()
5 {
6   if ( reset.read() == true )
7     state = S0;
8   else
9     state = next_state;
10 }
11
12 void fsm::ns_logic()
13 {
14   switch ( state ) {
15     case S0:
16       if ( input1.read() == 1 )
17         next_state = S1;
18       else
19         next_state = S0;
20       break;
21
22     case S1:
23       if ( input2.read() == 1 )
24         next_state = S2;
25       else
26         next_state = S0;
27       break;
28
29     case S2:
30       next_state = S0;
31       break;
32
33     default:
34       next_state = S0;
35       break;
36   }
37 }
38
39 void fsm::output_logic()
40 {
41   output1.write( state == S0 && (input1.read() || input2.read()) );
42   output2.write( state == S1 );
43 }
```

Figura 4.6: Implementação do modelo-exemplo

especializadas de forma a serem subclasses tanto das classes de SystemC quanto das classes do primeiro conjunto, herdando assim as propriedades e o comportamento de ambos.

Isto é mostrado em um trecho de código da PowerSC na figura 4.7. Este trecho

```
1   ...
2 template <int W, class T>
3 class psc_objinfo : public psc_objinfo_base, public psc_objinfo_if
4 {
5   ...
6 };
7   ...
8 template <int W>
9 class psc_bv : public sc_bv<W>, public psc_objinfo<W, sc_bv<W> >
10 {
11   ...
12 }
13   ...
```

Figura 4.7: Exemplo da herança

de código mostra o inter-relacionamento entre os conjuntos de classes. Mais especificamente, na figura é mostrado como a classe `sc_bv<W>` (vetor de bits de tamanho arbitrário) do SystemC está sendo especializada. O tipo equivalente na PowerSC é o `psc_bv` (note o acréscimo do prefixo **p**) que, como pode ser visto, é subclasse de `sc_bv<W>` e `psc_objinfo<W, sc_bv<W> >`, onde o parâmetro '`int W`' representa a largura em bits do tipo de dado, e '`class T`' denota a classe específica que `psc_objinfo` vai representar. Através do exemplo é possível ver como cada um dos tipos de dados, canais, etc, são especializados pela PowerSC.

Para possibilitar a monitoração das operações realizadas por estes objetos e, conseqüentemente de seus valores, os operadores das classes-base, como soma, atribuição, etc, foram sobrecarregados. Para exemplificar, considere o processo `update_state` de nosso modelo-exemplo (fig. 4.6). Com a sobrecarga do operador de atribuição torna-se possível monitorar o sinal `state` durante a execução deste processo e, com isso, capturar a atividade de transição resultante da mudança de seu valor. Isto é ilustrado na figura 4.8. Na simulação deste modelo, quando a operação de atribuição a `state` é alcançada, uma chamada ao operador de atribuição de nossa classe `psc_signal` é feita *(i)* e, dentro do código do operador, através de uma chamada ao método `write` (herdado de `sc_signal`), o valor sendo atribuído a `state` (`S0` ou `next_state`) é modificado. O que ocorre dentro deste método é outra atribuição, mais especificamente ao tipo de dado sendo representado pela classe `psc_signal` que, de acordo com a sua declaração no modelo, é `psc_uint<2>` *(ii)*. Agora, o que ocorre nesta atribuição é a atualização das informações do objeto com relação a atividade de transição. Ressaltando, a classe `psc_uint` além de `sc_uint` é também um *power object*, ou seja, ele sabe como executar esta tarefa, que é feita através do método `update_toggle_count` *(iii)*.

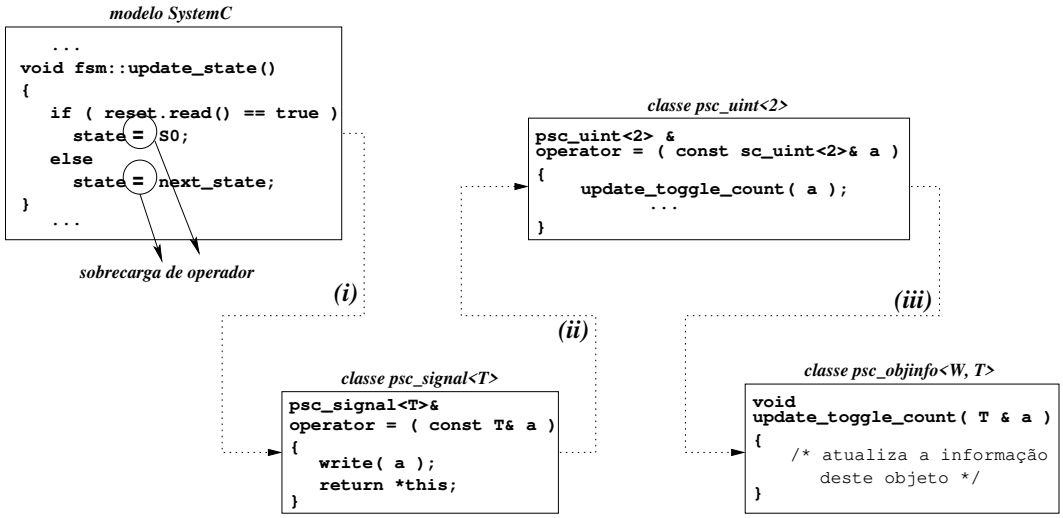


Figura 4.8: Sobrecarga do operador de atribuição

É importante dizer aqui que a declaração dos objetos no modelo não precisa ser modificada para os tipos da PowerSC e, inclusive, muito pouco precisa ser alterado para se adaptar à nossa abordagem. Para evitar a reescrita do modelo do sistema, foram utilizados alguns recursos da linguagem C, o que será explicado mais adiante.

Uma característica importante de ser mencionada também é que PowerSC leva em conta os *delta cycles* durante a simulação. Considere o trecho de código da figura 4.9, extraído de um processo qualquer de um modelo SystemC.

```
1  ...
2 data = 42; // default value
3 if ( /* qualquer condição */ )
4   data = in.read();
5  ...
```

Figura 4.9: Possível dupla atribuição

Suponha que, em algum ponto da simulação, esse código seja executado e o resultado da avaliação da condição seja `true`, e considere também como x o valor de `data` imediatamente anterior a execução deste código. Portanto, haverá duas atribuições para `data`: `42` e posteriormente o valor de `in.read()`, sendo que estas duas atribuições ocorrerão dentro de um mesmo *delta cycle*. Isto occore porque, a cada vez que um processo de SystemC é disparado, todo o código deste processo é executado pelo núcleo de simulação num *delta cycle* específico. Se este conceito for ignorado, ocorrerão duas transições para `data`: $x \rightarrow 42$ e $42 \rightarrow in.read()$, o que levará a uma super-estimação da atividade de transição real do modelo. No entanto, a implementação da PowerSC irá considerar somente a última atribuição, de forma que a única transição considerada será $x \rightarrow in.read()$.

Power Object Database Classes

A principal tarefa das classes deste conjunto é funcionar como um repositório das informações capturadas durante a simulação, bem como para o gerenciamento destes dados, como por exemplo, para geração de estatísticas. Este conjunto de classes está representado na figura 4.3 (pág. 25) pelo bloco nomeado **Repositório de Estatísticas**.

Não cabe ao repositório a monitoração dos elementos do modelo na simulação. Cada um dos objetos envolvidos na simulação é responsável pelo seu registro no repositório e por monitorar seus dados, bem como pela atualização das informações capturadas no repositório.

O diagrama de classes simplificado da biblioteca PowerSC é mostrado na figura 4.10. Nem todas as classes estão presentes nesta figura por questões de espaço. Pode-se ver na figura, separadas em caixas distintas, as classes originais de SystemC (*SystemC Classes*), como também as descritas até o momento (*SystemC Augmented*, *Power Object* e *Power Object Database classes*).

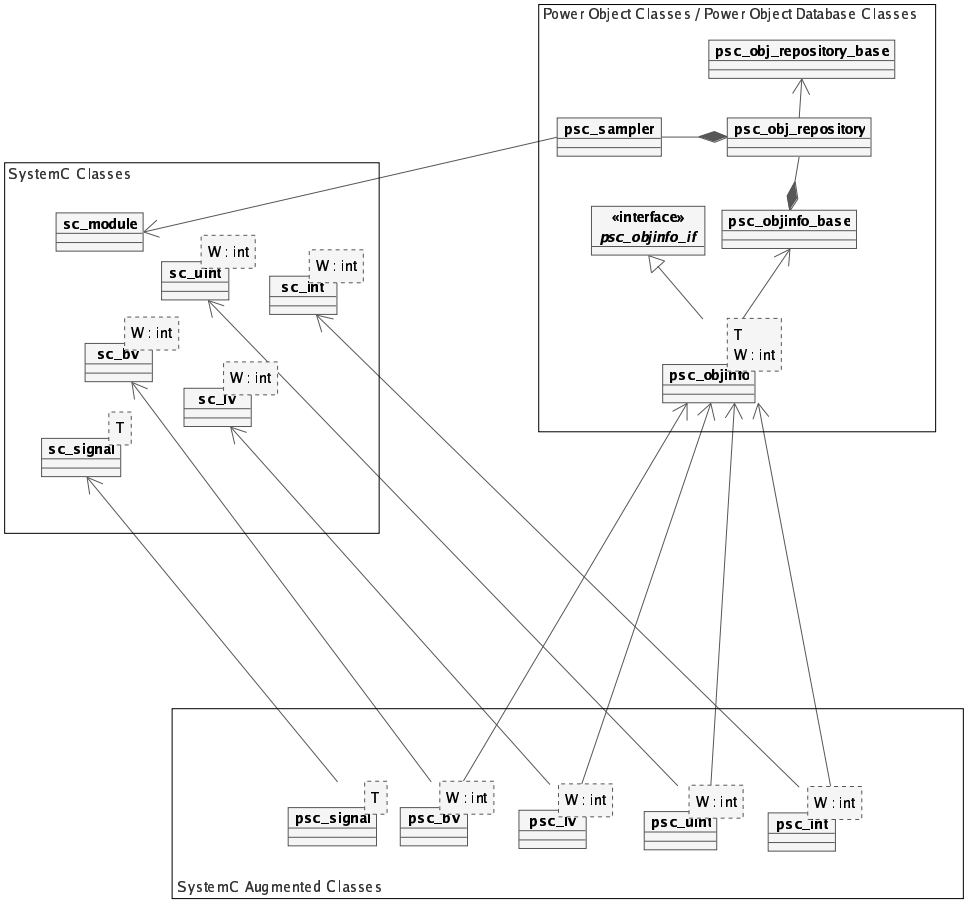


Figura 4.10: Diagrama de classes simplificado da PowerSC

Note a múltipla herança nas classes derivadas de SystemC, herdando propriedades e comportamento tanto dos tipos de SystemC, como também de `psc_objinfo<W, T>`. A única exceção é para `psc_signal<T>`, onde isto não é necessário, pois o que acontece é se ter a seguinte declaração em SystemC:

```
sc_signal< sc_uint<2> > signal;
```

que em PowerSC tranforma-se em:

```
psc_signal< psc_uint<2> > signal;
```

ou seja, o tipo do sinal é que precisa das características de `psc_objinfo<W, T>`, que é o que ocorre.

Para facilitar o entendimento desta hierarquia de classes, na próxima seção será mostrada a relação entre todos os conjuntos vistos anteriormente, ou seja, o funcionamento da especificação executável incrementada como um todo.

4.4 Simulação com a PowerSC

Podemos classificar a simulação em PowerSC em dois tipos, de acordo com o tipo de monitoração realizada: *simulação com monitoração total* e *simulação com monitoração por amostragem*.

4.4.1 FMS – Simulação com Monitoração Total

FMS[3] é o tipo de monitoração padrão utilizado pela PowerSC e é caracterizado pelo fato de os elementos do modelo serem monitorados durante toda a execução da simulação. Para entender como esta monitoração é realizada, considere novamente os sinais `state` e `next_state` do modelo-exemplo (fig. 4.5, pág. 27).

As etapas desde a execução do modelo compilado até o seu final são mostradas na figura 4.11, e são como se segue: o modelo compilado usando a PowerSC (*a.out*) é inicialmente executado, o que leva à etapa de elaboração, imediatamente anterior à simulação. Durante a elaboração, os elementos estruturais (módulos, canais, portas, etc) do modelo são instanciados e conectados através de sua hierarquia. No que se refere especificamente aos objetos da biblioteca PowerSC, há a criação de um identificador único para estes objetos (utilizado pelo repositório), e a inicialização de seus atributos. Na etapa de simulação, estes objetos encarregam-se de realizar a sua própria monitoração e toda operação realizada com estes objetos, que implique em alguma modificação de seus valores internos, é interceptada e a informação capturada é anotada e mantida internamente.

[3]Do inglês, Full Monitored Simulation

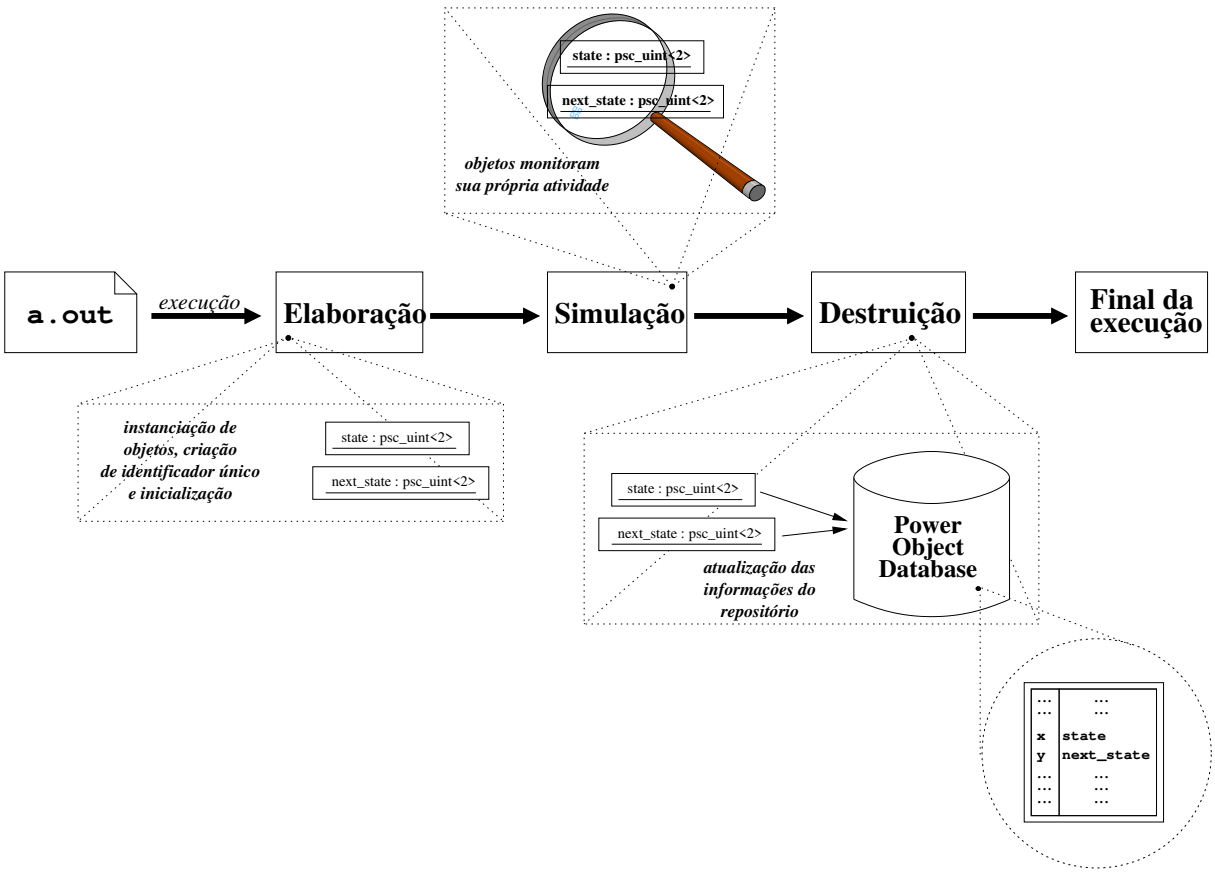


Figura 4.11: Simulação Monitorada Total

Ao final desta etapa, quando o modelo simulado está para ser liberado da memória, os objetos atualizam as informações capturadas para o repositório (*power object database*). Isto é facilitado pelo comportamento de destruição de objetos de C++.

A atualização das informações capturadas para o repositório é feita somente no momento em que o objeto é destruído, de modo a evitar que, a cada *delta cycle*, o repositório verifique o *status* de cada um dos objetos da simulação, procurando por mudanças ocorridas. Usando esta abordagem, consegue-se minimizar o *overhead* da monitoração sobre o desempenho do simulador.

4.4.2 SMS – O Algoritmo Proposto

Nesta seção é apresentada uma técnica semelhante a técnica anterior, porém elaborada para reduzir o tempo necessário de monitoração sempre que possível, com a mesma precisão da FMS. A idéia geral da *simulação com monitoração por amostragem*, ou SMS[4],

[4]Do inglês, Sampled Monitored Simulation

é realizar amostragens de monitoração durante a simulação, de forma que o circuito seja monitorado apenas dentro destas janelas de amostragem. Uma média global prevista de atividade de transição é fornecida durante a simulação, e é calculada usando a informação tanto do tempo monitorado quanto do tempo em que a monitoração está suspensa.

As etapas aqui são semelhantes às que foram mostradas na figura 4.11. A principal diferença está na etapa de simulação, dado que esta não é monitorada durante todo o tempo.

No tipo anterior, FMS, o repositório não tem muito controle do que ocorre durante a simulação, ele somente recebe as informações capturadas pelos objetos no final da simulação, para então poder gerar estatísticas a partir destes dados. Na SMS, diferentemente, o repositório tem um papel mais importante durante a simulação, como controlador, ressaltando porém que ele não é o responsável pela monitoração dos objetos. Durante certos intervalos da simulação, o repositório anota as informações capturadas pelos objetos e gera algumas estatísticas baseadas nestes dados.

Isto é ilustrado na figura 4.12. Durante a etapa de simulação, os objetos envolvidos requisitam ao repositório autorização para realizar sua auto-monitoração *(i)*, e este concede, ou não, baseado nas estatísticas *(ii)* feitas com os dados anotados pelo repositório.

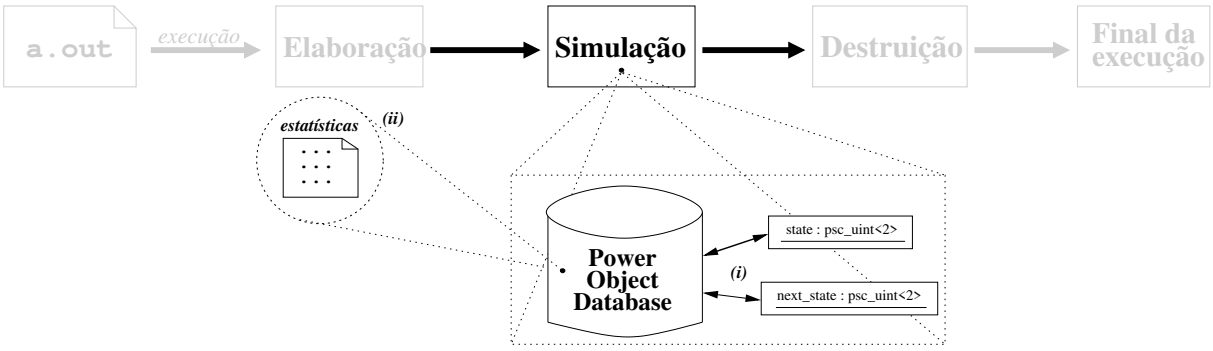


Figura 4.12: Simulação Monitorada Amostrada

A figura 4.13 ilustra a idéia do algoritmo SMS (mostrado a seguir), que implementa o processo de amostragem. Como pode ser visto pela figura, o circuito é inicialmente monitorado, com o objetivo de se obter a primeira média global de atividade de transição. Então janelas de monitoração (t^i_{mon}) e não monitoração (t^j_{pred}) são intercaladas, e note que as janelas têm seus tamanhos modificados durante a simulação.

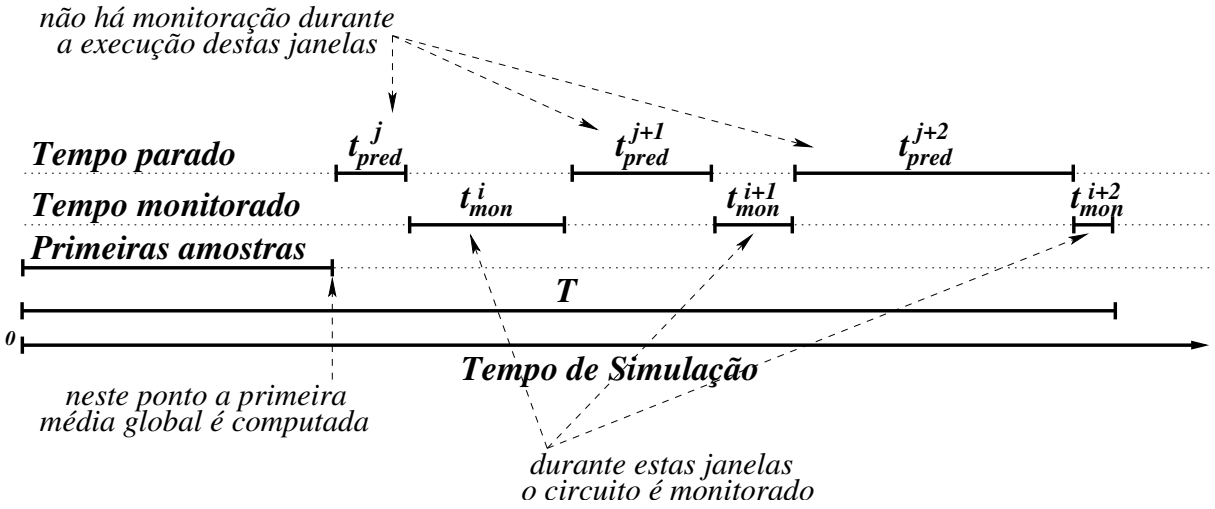


Figura 4.13: Visão geral do processo de amostragem

O algoritmo em pseudo-código da figura 4.14 mostra simplificadamente como o processo de amostragem funciona. O conceito de **amostra** aqui é definido como sendo uma unidade do intervalo de tempo mínimo que a simulação deve estar monitorando, ou não, dependo do estado da execução do algoritmo. O intervalo de amostragem é de *500 ciclos*. Os identificadores do algoritmo são os seguintes:

- *monitor*: variável de controle, indica aos objetos da simulação se estes devem realizar sua auto-monitoração;
- *min_mon_samples, max_mon_samples*: número mínimo e máximo de amostras com monitoração por iteração;
- *min_pred_samples, max_pred_samples*: mesmo que o anterior, mas para amostras não monitoradas (*predicted*);
- s_{mon}: número atual de amostras monitoradas;
- s_{pred}: número atual de amostras não monitoradas;
- *num_first_samples*: número de amostras monitoradas no início da simulação, e utilizadas para o cálculo da primeira média;
- t_s: tempo de uma única amostra (500 ciclos);
- t_{mon}: tempo da janela de amostragem atual;
- t_{pred}: tempo da janela de simulação não monitorada atual;

- *take_first_samples(x)*: função responsável pela primeira amostragem. Monitora a simulação pelo tempo x;
- μ_G: média global de atividade de transição atual;
- *compute_global_average()*: calcula a média global de atividade de transição, considerando o tempo de amostragem e o tempo não monitorado da simulação;
- *stop_monitoring(x)*: suspende a monitoração dentro da simulação pelo tempo x;
- *resume_monitoring(x)*: retoma a monitoração na simulação pelo tempo x;
- μ_L: média local de atividade de transição da última janela de amostragem;
- *compute_local_average()*: calcula a média de atividade de transição, considerando somente as informações capturadas dentro da última janela de simulação monitorada;
- *threshold*: percentual de diferença entre a média local e a média global permitido;
- z: desvio da média global permitido para a média local;
- μ_{mon}: média de atividade de transição, considerando somente o tempo de simulação monitorado.

O primeiro passo do algoritmo é obter a primeira média de atividade de transição global. Isto é feito rodando-se a simulação e monitorando a atividade de transição por *num_first_samples*, de forma a computar a primeira média global, baseada na atividade de transição capturada até o final deste passo. Após este passo, e pelo resto da simulação, o algoritmo tenta aumentar ao máximo o número de amostras não monitoradas, reduzindo o tempo monitorado. A simulação tem alternadas janelas de monitoração e não monitoração, e estas janelas têm também seus tamanhos (número de amostras) modificados, dependendo da estabilidade do circuito. Por exemplo, se o circuito está relativamente estável, a tendência é de crescer a janela de não monitoração, e contrair a janela de monitoração. No caso contrário, isto é, de ser detectada instabilidade no circuito, o inverso é feito.

Depois da execução de uma janela de monitoração, uma média de atividade de transição local μ_L (em relação à esta janela) é computada. Está média é então comparada com a média global μ_G e, baseada nesta comparação, as janelas são modificadas: se a média local respeita o *threshold*, é suposto pelo algoritmo que a atividade do circuito é estável, contraindo a janela de monitoração e crescendo a janela de não monitoração.

Ensure: $min\_mon\_samples \leq s_{mon} \leq max\_mon\_samples$
Ensure: $min\_pred\_samples \leq s_{pred} \leq max\_pred\_samples$
1: $monitor \leftarrow true$
2: $take\_first\_samples(num\_first\_samples)$
3: $\mu_G \leftarrow compute\_global\_average()$
4: $s_{mon} \leftarrow num\_first\_samples/2$
5: $s_{pred} \leftarrow min\_pred\_samples$
6: $t_{mon} \leftarrow s_{mon} \times t_s$
7: $t_{pred} \leftarrow s_{pred} \times t_s$
8: **while** simulation is running **do**
9: $monitor \leftarrow not\ monitor$
10: **if** $monitor = true$ **then**
11: $resume\_monitoring(t_{mon})$
12: $\mu_L \leftarrow compute\_local\_average()$
13: $z \leftarrow \mu_L \times threshold$
14: **if** $-z \leq \mu_G - \mu_L \leq z$ **then**
15: $s_{mon} \leftarrow s_{mon}/2$
16: $s_{pred} \leftarrow s_{pred} \times 2$
17: **else**
18: $s_{mon} \leftarrow s_{mon} \times 2$
19: $s_{pred} \leftarrow s_{pred}/2$
20: **end if**
21: $\mu_{mon} \leftarrow update\ the\ monitored\ average$
22: **else**
23: $stop\_monitoring(t_{pred})$
24: $\mu_{pred} \leftarrow update\ the\ predicted\ average$
25: **end if**
26: $t_{mon} \leftarrow s_{mon} \times t_s$
27: $t_{pred} \leftarrow s_{pred} \times t_s$
28: $\mu_G \leftarrow compute\_global\_average()$
29: **end while**

Figura 4.14: Algoritmo em pseudo-código do processo de amostragem

A média da atividade de transição do tempo de monitoração (μ_{mon}) é também atualizada após este passo, de acordo com a seguinte expressão:

$$\mu_{mon} = \sum_{i=0}^{N_{mon}-1} \frac{TC^i_{mon}}{t^i_{mon}} \tag{4.1}$$

onde TC^i_{mon} é a contagem de transições[5] capturada, t^i_{mon} a duração da janela, e i e N_{mon} são a janela e o número total de janelas de monitoração, respectivamente.

Na próxima iteração do algoritmo, a janela de não monitoração é executada, isto é, a simulação continua, porém a monitoração é suspensa pelo tempo t_{pred}, calculado na iteração anterior durante a checagem do *threshold*. A média da atividade de transição prevista é atualizada de forma análoga à anterior, usando a expressão:

$$\mu_{pred} = \sum_{i=0}^{N_{pred}-1} \frac{TC^i_{pred}}{t^i_{pred}} \tag{4.2}$$

onde i é a janela de não monitoração e N_{pred} o número total de janelas de não monitoração; t^i_{pred} é a duração da *i-ésima* janela, calculada no fim da execução da janela de monitoração. TC^i_{pred} é o número de transições previsto para a *i-ésima* janela. Para cada janela de não monitoração, a média de atividade de transição assumida é igual à última média local, logo, sua contagem de transições é calculada como:

$$TC^i_{pred} = \mu_L \times t^i_{pred} \tag{4.3}$$

Ao final de cada iteração do algoritmo, a média global (μ_G) é calculada levando-se em conta tanto a média prevista (μ_{pred}), como a média monitorada (μ_{mon}), pesando seus valores de acordo com a contribuição de cada uma para o tempo total de simulação. Esta média global de atividade de transição ponderada é o valor fornecido ao usuário durante a simulação, e é computada como segue:

$$\mu_G = \omega_{mon}\mu_{mon} + \omega_{pred}\mu_{pred} \tag{4.4}$$

onde ω_{mon} e ω_{pred} são os pesos da monitoração e da não monitoração, respectivamente. Os valores ω_{mon} e ω_{pred} são médias ponderadas simples sobre e tempo total de simulação, como mostrado abaixo:

$$T = \sum_{i=0}^{N_{mon}-1} t^i_{mon} + \sum_{j=0}^{N_{pred}-1} t^j_{pred} + (num\_first\_samples \times t_s) \tag{4.5}$$

$$\omega_{mon} = \frac{\sum_{i=0}^{N_{mon}-1} t^i_{mon}}{T} \tag{4.6}$$

$$\omega_{pred} = \frac{\sum_{j=0}^{N_{pred}-1} t^j_{pred}}{T} \tag{4.7}$$

[5]Do inglês, toggle count

A intenção principal deste algoritmo é de detectar, com eficácia, períodos de estabilidade do circuito na simulação, e tirar proveito deste períodos estáveis, parando a monitoração, reduzindo assim o tempo total necessário de monitoração dentro da simulação, com uma perda mínima de precisão. De forma análoga, a instabilidade também é detectada e, então, a monitoração é reiniciada, de forma a ajustar a média de atividade de transição global fornecida ao usuário.

Para circuitos altamente instáveis, o tempo necessário de monitoração é maior que em outros circuitos mais bem comportados. Este comportamento instável é detectado pelo algoritmo e o tempo de monitoração é ajustado de acordo, fornecendo sempre uma média global confiável, que é a característica mais importante das estimativas. A idéia contida neste algoritmo foi implementada na PowerSC, porém, pode ser facilmente adaptada para outras abordagens.

O algoritmo tem alguns parâmetros que podem ser configurados pelo usuário, dependendo da confiança necessária nos resultados das estimativas e/ou do conhecimento prévio do comportamento do circuito. Estes parâmetros são: *threshold*, o número de amostras iniciais (*num_first_samples*), e os limites mínimos e máximos das amostras monitoradas e não monitoradas (*min_mon_samples*, *max_mon_samples*, *min_pred_samples*, *max_pred_samples*). A influência destes parâmetros nos resultados experimentais é mostrada no capítulo 5.

4.5 A Metodologia Proposta

As intenções desta metodologia são, principalmente, a facilidade de uso e a velocidade de aplicação, reduzindo o número de ferramentas necessárias, e tornando o processo mais simples e transparente para o projetista. O fluxo da PowerSC pode ser visto na figura 4.15, entre as linhas tracejadas.

Inicialmente, o modelo SystemC, juntamente com alguns arquivos de configuração, é fornecido como entrada para um compilador padrão de C++ *(i)*. O compilador então gera como sua saída a especificação executável do modelo, incrementada para a captura de atividade de transição, devidamente ligada com as bibliotecas PowerSC e SystemC *(ii)*. O próximo passo é então executar esta especificação que, ao iniciar, irá disparar uma simulação RTL do modelo *(iii)*, que pode ser de dois tipos, conforme mostrado na seção 4.4: *simulação com monitoração total (***FMS***)* ou *simulação com monitoração por amostragem (***SMS***)*.

Ao final da simulação, os resultados obtidos são mostrados em relatórios e estatísticas *(iv)*. No caso de SMS, durante a execução da simulação, estatísticas de atividade de transição média são fornecidas. Com base nos resultados obtidos, o projetista é capaz de detectar possíveis gargalos de consumo de potência, fazer refinamentos em seu modelo, e

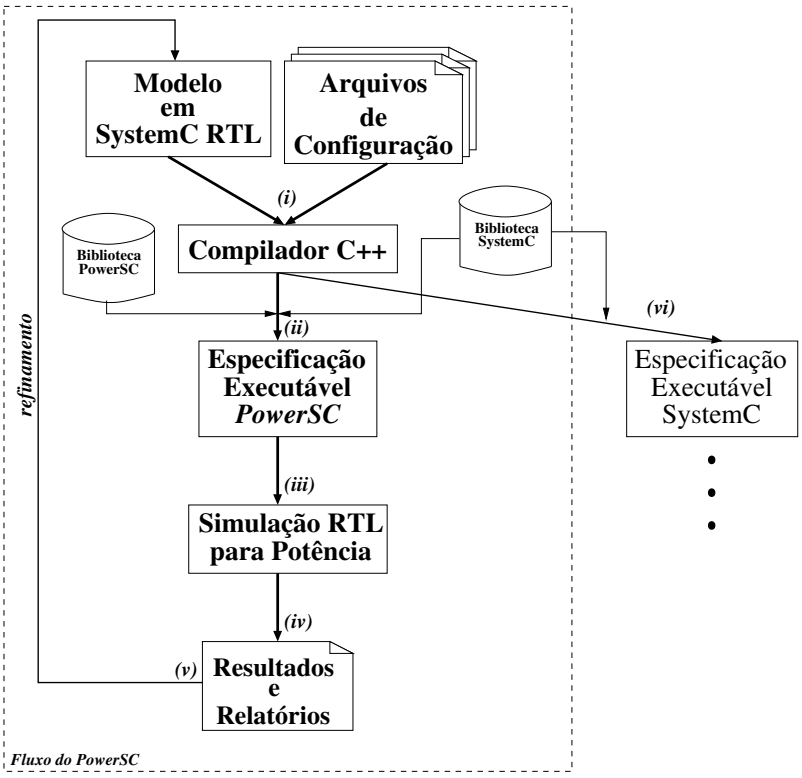


Figura 4.15: O fluxo da PowerSC

rodar o fluxo novamente *(v)* para uma outra análise de seu modelo.

Alternativamente ao fluxo da PowerSC, o fluxo tradicional do SystemC pode ser rodado facilmente, necessitando-se apenas instruir o compilador C++ (através dos arquivos de configuração) a gerar a especificação executável usual *(vi)*.

Note que esta metodologia se localiza num estágio bem inicial do fluxo de projeto, justamente onde estão as maiores oportunidades de se localizar os gargalos de consumo do sistema. Além do que a recorrência ao início do fluxo aqui é pequena e, portanto, pode-se descobrir possíveis problemas de projeto antecipadamente, economizando tempo nas etapas mais avançadas do projeto.

São necessárias poucas modificações nas descrições para que elas se adaptem à esta metodologia, e dentre estas, uma obrigatória é a inclusão do arquivo de cabeçalhos principal do PowerSC (`powersc.h`) nos arquivos do modelo, e a chamada à macro `PSC_REPORT_SWITCHING_ACTIVITY` no final da função `sc_main` do simulador. Isto é ilustrado nas figuras 4.16 e 4.17.

A modificação no código de nosso exemplo da máquina de estados da figura 4.5 (pág. 27), para a figura 4.16 é somente a inclusão do arquivo de cabeçalhos principal da PowerSC, localizada na linha 4. O arquivo de implementação do nosso exemplo (fig.

```
1  /* initial example adapted to PowerSC */
2  // header file: fsm.h
3  #include <systemc.h> // <-- it can be optionally removed --
4  #include <powersc.h> // <-- mandatory modification --
5
6  // An example of a finite-state machine
7  SC_MODULE(fsm) {
8    // input ports
9    sc_in_clk clk;
10   sc_in<sc_bit> reset;
11   sc_in<sc_bit> input1, input2;
12
13   // output ports
14   sc_out<sc_bit> output1, output2;
15
16   // signals
17   sc_signal<sc_uint<2> > state, next_state;
18
19   // processes
20   void ns_logic();
21   void output_logic();
22   void update_state();
23
24   // enumerate states
25   enum state_t { S0, S1, S2 };
26
27   SC_CTOR(fsm) {
28     SC_METHOD( update_state );
29     sensitive_pos << clk;
30
31     SC_METHOD( ns_logic );
32     sensitive << state << input1 << input2;
33
34     SC_METHOD(output_logic);
35     sensitive << state << input1 << input2;
36   }
37 };
```

Figura 4.16: Exemplo inicial da máquina de estados adaptado à PowerSC

4.6, pág 28) não requer nenhuma modificação para se adequar à metodologia, e por isso não é mostrado aqui.

Na figura 4.17 é mostrado um trecho de código do arquivo principal do modelo-exemplo, que contém a chamada à função principal do simulador (`sc_main`). Neste arquivo, as modificações que foram necessárias estão localizadas nas linhas 3 e 22: a inclusão do arquivo de cabeçalhos `powersc.h` e a chamada à macro `PSC_REPORT_SWITCHING_ACTIVITY`, que gera os relatórios com as estatísticas ao final da simulação. O único pré-requisito para esta macro é que uma chamada à ela deve estar localizada após a última chamada à `sc_start`, a função de SystemC que executa a simulação. As principais macros da PowerSC disponíveis para o usuário podem ser vistas na tabela 4.1.

Macro	**Descrição**
PSC_REPORT_SWITCHING_ACTIVITY	imprime na saída padrão um relatório detalhado com as informações capturadas na simulação
PSC_FULL_SAMPLING	seleciona o método de monitoração *FMS* (padrão). Deve ser colocado em qualquer posição antes do início da simulação
PSC_ENABLE_SMS	seleciona o método de monitoração *SMS*. Deve ser colocado em qualquer posição antes do início da simulação
PSC_SAMPLES_PRED_RANGE(*min*, *max*)	configura o número mínimo (*min*) e máximo (*max*) de amostras não monitoradas. Válido somente para **SMS**
PSC_SAMPLES_MON_RANGE(*min*, *max*)	configura o número mínimo (*min*) e máximo (*max*) de amostras monitoradas. Válido somente para **SMS**
PSC_NUM_FIRST_SAMPLES(n)	configura o número inicial (n) de amostras monitoradas. Válido somente para **SMS**
PSC_SAMPLING_THRESHOLD(t)	configura o *threshold* (t) utilizado no algoritmo **SMS**
PSC_STATUS_QUO(i, s)	imprime na *stream* **s** as médias de atividade de transição intermediárias em intervalos de tempo i
PSC_WRITE_CSV_FILE(*name*)	grava em arquivo (*name*) as informações capturadas na simulação, no formato CSV (*comma-separated values*)
PSC_IGNORE(*obj*)	configura o objeto (*obj*) para que seja ignorado pelo processo de monitoração
PSC_OBJ_ALIAS(*obj*, *alias*)	configura uma *string* (*alias*) como um apelido para o objeto *obj*. O apelido só é utilizado na impressão dos relatórios

Tabela 4.1: Principais macros disponíveis para o usuário

```
1  /* file: main.cpp */
2  #include <systemc.h>
3  #include <powersc.h> // <-- mandatory modification --
4  #include "fsm.h"
5  #include "testbench.h"
6
7  void create_model()
8  {
9     testbench tb1( "tb1" );
10    fsm fsm1( "fsm1" );
11
12    sc_clock clk( "clk", sc_time(20, SC_NS) );
13    sc_signal<sc_bit> sig_reset( "sig_reset" );
14    sc_signal<sc_bit> sig_input1( "sig_input1" );
15    ... /* and so on */
16 }
17
18 int sc_main( int argc, char **argv )
19 {
20    create_model(); // <-- instantiate and bind modules --
21    sc_start( simulation time );
22    PSC_REPORT_SWITCHING_ACTIVITY; // <-- print the report --
23    return( 0 );
24 }
```

Figura 4.17: Arquivo main.cpp do exemplo

Como pode ser visto nos exemplos mostrados, o esforço necessário para habilitar a captura de atividade de transição em PowerSC é mínima. Deve ter sido notado que, de todas as modificações descritas, nenhuma delas faz referência a alterações no modelo dos objetos do SystemC para os objetos da PowerSC.

Isto é feito automaticamente pela PowerSC, necessitando apenas de um arquivo de configuração adequado para isto. No arquivo `powersc.h` estão todas as definições necessárias para o uso da biblioteca da maneira como foi descrita. Um fragmento deste arquivo é mostrado na figura 4.18.

Neste arquivo, além da inclusão dos tipos próprios, *namespaces* e macros da PowerSC, os tipos do SystemC são redefinidos usando-se diretivas de compilação. Ou seja, se a diretiva `POWER_SIM` está estabelecida durante a compilação do modelo, os objetos SystemC declarados serão redefinidos automaticamente, e as macros da PowerSC serão definidas. E, considerando que a diretiva não esteja estabelecida, as macros estarão definidas como vazio, desta forma evitando que o modelo tenha que ser reescrito. Como pode ser visto, o mesmo modelo pode ser utilizado tanto no fluxo usual do SystemC, como no fluxo da PowerSC, necessitando-se apenas o ajuste de algumas diretivas de compilação. Portanto, tudo que teria de ser feito é prover duas opções de *Makefiles* (arquivos de configuração), uma para cada fluxo, e uma simples recompilação do simulador usando o *Makefile* adequado ativaria (ou não) a captura de atividade de transição. A figura 4.19 ilustra isto para o nosso modelo-exemplo.

```
1    ...
2  #include "psc_uint.h"
3  #include "psc_signal.h"
4  #include "psc_clock.h"
5    ...
6  using psc_dt::psc_uint;
7  using psc_dt::psc_int;
8  using psc_dt::psc_bv;
9    ...
10 using namespace psc_util;
11   ...
12 #ifdef POWER_SIM
13 #define sc_uint psc_uint
14 #define sc_int psc_int
15 #define sc_signal psc_signal
16 #define sc_bv psc_bv
17 #define sc_clock psc_clock
18   ...
19 #else
20   ...
21 #endif
22   ...
```

Figura 4.18: Fragmento do arquivo de cabeçalho principal

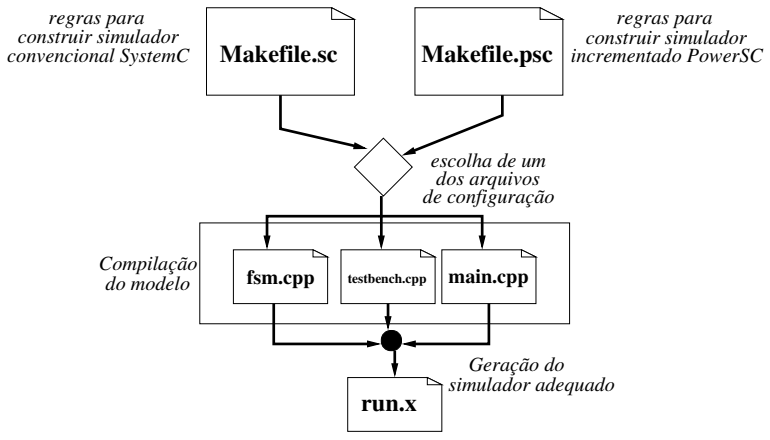


Figura 4.19: Processo de compilação para o modelo-exemplo

Considerando novamente o nosso modelo-exemplo, poderíamos ter dois arquivos de configuração: *Makefile.sc* e *Makefile.psc*, no qual o primeiro arquivo possuiria as regras normais para compilação usando o SystemC, e o segundo teria as regras para a PowerSC. Um trecho de um possível arquivo é dado na figura 4.20.

```
1  # file: Makefile.psc -- Builds the PowerSC simulator
2  CC=g++
3  EXE=run.x
4  SC_DIR=/usr/local/systemc
5  PSC_DIR=/usr/local/powersc
6  LIB_DIR=-L$(SC_DIR)/lib-linux -L$(PSC_DIR)/lib
7  LIBS=-lsystemc -lpowersc
8  INC_DIR=-I$(PSC_DIR)/include -I$(SC_DIR)/include -I. -I..
9  CFLAGS=-Wall -Wno-deprecated -Wno-char-subscripts -O3
10 SRCS=fsm.cpp testbench.cpp main.cpp ...
11 OBJS=$(SRCS:.cpp=.o)
12 DEFS=-DPOWER_SIM -DSIMULATION
13
14 all: $(EXE)
15
16 $(EXE): $(OBJS)
17       $(CC) $(LIB_DIR) -o $@ $(OBJS) $(LIBS)
18
19 fsm.o: fsm.cpp
20       $(CC) $(CFLAGS) $(DEFS) $(INC_DIR) -c fsm.cpp
21 ...
```

Figura 4.20: Um trecho de um arquivo Makefile.psc

Este arquivo contém as informações necessárias para a geração da especificação executável do modelo, como qual compilador utilizar, caminhos de arquivos de cabeçalho, opções de compilação, etc. Porém, à parte deste detalhes, note nas linhas 7 e 12, que são selecionadas as bibliotecas para a ligação[6] do simulador (`-lsystemc -lpowersc`) e estabelecida a diretiva `POWER_SIM`, mostrada anteriormente.

Como pôde ser visto neste capítulo, a metodologia e a forma de uso da PowerSC são caracterizadas pela simplicidade e facilidade de uso, por requererem apenas o mesmo compilador já utilizado nas simulações convencionais. No próximo capítulo serão apresentados os resultados obtidos pelo uso da PowerSC, e será feita uma análise do comportamento da biblioteca.

[6] do inglês, *linking*

Capítulo 5

Resultados Experimentais

Neste capítulo, serão apresentados os resultados experimentais obtidos com o uso da biblioteca PowerSC, apresentada no capítulo 4. Para a avaliação dos resultados obtidos, foram executados dois diferentes *design flows*: o da PowerSC, visto na figura 4.15 (pág. 40) e o fluxo do Power Compiler [7], mostrado na figura 4.2 (pág 23).

É importante ressaltar aqui que a comparação feita entre as duas metologias e ferramentas é feita com relação à captura da atividade de transição (*switching activity*) de modelos SystemC RTL.

O simulador utilizado em conjunto com o Power Compiler foi o ModelSim SE 5.8b [33]. Para a PowerSC, o GCC 3.3.2 [37] foi utilizado como o compilador C++. A máquina *host*, onde todos os experimentos foram executados, foi um Pentium IV 2.8GHz 1GB RAM, rodando o sistema operacional Linux.

Os experimentos foram divididos em duas categorias:

I. **Experimentos de validação** (seção 5.1)

II. **Experimentos com o algoritmo SMS** (seção 5.2)

5.1 Validação da Biblioteca PowerSC

Os experimentos da primeira categoria têm o objetivo principal de validar a corretude dos resultados obtidos pela PowerSC com relação a uma ferramenta comercial. Estes experimentos utilizam o método de simulação com monitoração total (FMS). O conjunto de modelos em SystemC utilizado para os testes com esta categoria foi o seguinte:

- **shift_reg + counter**: um registrador de deslocamento com um contador;
- **count_zeros_seq**: um circuito que recebe como entrada uma palavra de 8 *bits* e fornece como saída o número de zeros da palavra de entrada;

- **drink_machine**: simula uma máquina de venda de bebidas, que recebe moedas para cada tipo de bebida e devolve o troco, se necessário;
- **MP3-DCT**: módulo DCT (*Discrete Cosine Transform*) de uma implementação real de um decodificador de MP3;
- **FIR**: projeto de um filtro FIR (*Finite Impulse Response*);
- **ripple-carry adder**: um somador *ripple-carry* de 16 *bits* e
- **CLA adder**: um somador *carry-lookahead* de 16 *bits*.

Os resultados obtidos para esta categoria podem ser vistos nas tabelas 5.1 e 5.2.

Experim.	**Sem.**	**PowerSC**		**Power Compiler**		**Comparação**	
		Tempo	**TC**	**Tempo**	**TC**	**Tempo**	**Erro**
shift_reg + counter	#1	0m18s	31800K	0m24s	31800K	1,30	$\sim 0\%$
	#2	0m18s	31800K	0m22s	31800K	1,2	$\sim 0\%$
	#3	0m18s	31800K	0m24s	31800K	1,29	$\sim 0\%$
	#4	0m18s	31800K	0m24s	31800K	1,29	$\sim 0\%$
	#5	0m18s	31800K	0m24s	31800K	1,30	$\sim 0\%$
count_zeros_seq	#1	0m54s	58975K	0m45s	58975K	0,83	$\sim 0\%$
	#2	0m54s	58973K	0m45s	58973K	0,83	$\sim 0\%$
	#3	0m54s	58976K	0m45s	58976K	0,83	$\sim 0\%$
	#4	0m54s	58984K	0m44s	58984K	0,82	$\sim 0\%$
	#5	0m54s	58971K	0m45s	58971K	0,83	$\sim 0\%$
drink_machine	#1	0m46s	29062K	0m20s	28437K	0,42	2%
	#2	0m46s	29062K	0m20s	28437K	0,42	2%
	#3	0m46s	29062K	0m20s	28437K	0,42	2%
	#4	0m46s	29062K	0m20s	28437K	0,42	2%
	#5	0m46s	29062K	0m20s	28437K	0,42	2%
MP3-DCT	#1	4m39s	960390K	7m11s	828665K	1,54	15%*
	#2	4m39s	960396K	7m15s	828645K	1,55	15%*
	#3	4m39s	960430K	7m20s	828574K	1,57	15%*
	#4	4m39s	960346K	7m18s	828616K	1,56	15%*
	#5	4m39s	960371K	7m31s	828640K	1,61	15%*

*Diferença devida a uma limitação no Power Compiler

Tabela 5.1: Resultados Básicos - Parte I

Estas tabelas estão divididas em cinco colunas: a primeira coluna mostra o nome do experimento realizado e a segunda coluna indica para qual semente (para os números pseudo-aleatórios) o resultado se refere (foram utilizadas cinco sementes diferentes para

Experim.	Sem.	PowerSC		Power Compiler		Comparação	
		Tempo	TC	Tempo	TC	Tempo	Erro
FIR	#1	0m31s	63503K	0m35s	47476K	1,10	33%*
	#2	0m31s	63503K	0m35s	47476K	1,11	33%*
	#3	0m31s	63503K	0m35s	47476K	1,09	33%*
	#4	0m31s	63503K	0m35s	47476K	1,12	33%*
	#5	0m31s	63503K	0m35s	47476K	1,12	33%*
ripple-carry adder	#1	6m28s	169156K	2m10s	169348K	0,33	$\sim 0\%$
	#2	6m29s	169168K	2m11s	169364K	0,33	$\sim 0\%$
	#3	6m26s	169158K	2m13s	169359K	0,34	$\sim 0\%$
	#4	6m28s	169179K	2m14s	169376K	0,34	$\sim 0\%$
	#5	6m29s	169158K	2m11s	169355K	0,33	$\sim 0\%$
CLA adder	#1	10m47s	249160K	2m45s	249159K	0,25	$\sim 0\%$
	#2	10m46s	249153K	2m45s	249155K	0,25	$\sim 0\%$
	#3	10m47s	249141K	2m45s	249142K	0,25	$\sim 0\%$
	#4	10m46s	249176K	2m49s	249174K	0,26	$\sim 0\%$
	#5	10m47s	249161K	2m45s	249164K	0,25	$\sim 0\%$

*Diferença devida a uma limitação no Power Compiler

Tabela 5.2: Resultados Básicos - Parte II

cada experimento). A terceira e quarta colunas são subdividas em duas colunas, e mostram os resultados de tempo de simulação e contagem de transições (*toggle count*) para a PowerSC e Power Compiler/ModelSim, respectivamente. A quinta e última coluna das tabelas apresenta um comparativo entre as duas abordagens e também é subdividida em duas colunas. A razão de tempo ($\frac{Tempo_{PwC}}{Tempo_{PSC}}$) entre as duas é apresentada na primeira subcoluna, e o erro na captura da atividade de transição do PowerSC em relação ao Power Compiler é apresentado na segunda. O tempo simulado para cada um dos experimentos foi de 5.000.000 (5M) de ciclos.

Como pode ser visto nas tabelas, os resultados obtidos para os experimentos *shift_reg+counter*, *count_zeros_seq*, *ripple-carry adder* e *CLA adder* com relação a captura da atividade de transição são praticamente iguais, com uma diferença irrelevante.

No experimento *drink_machine*, a PowerSC super-estima o Power Compiler por algo em torno de 2%. Esta super-estimação ocorre devido à inexistência de representação dos quatro estados lógicos ('0', '1', 'X', 'Z') para alguns tipos de dados. O SystemC tem este suporte para alguns tipos, como por exemplo, `sc_logic` e `sc_lv<W>`, que representam um valor lógico e um vetor de valores lógicos, respectivamente. Porém, tipos como `sc_int<W>` e `sc_uint<W>` não têm este suporte.

Vamos ilustrar esse problema através de um exemplo. Considere o código em SystemC da figura 5.1. O código Verilog equivalente é mostrado na figura 5.2. Durante a simulação

```
1       ...
2  sc_signal<sc_uint<32> > mul_b;
3  sc_signal<sc_uint<32> > temp;
4  sc_signal<sc_uint<32> > y_out;
5       ...
6       ...
7  void some_proc() {
8       ...
9     mul_b = temp - y_out;
10      ...
11 }
```

Figura 5.1: Trecho de código em SystemC

```
1       ...
2  reg [31:0] mul_b;
3  reg [31:0] temp;
4  reg [31:0] y_out;
5       ...
6  always @ ( ... )
7     begin : some_proc
8                ...
9        mul_b = temp - y_out;
10               ...
11    end
```

Figura 5.2: Trecho de código em Verilog

deste código com o ModelSim, se alguma das entradas (`temp` ou `y_out`) têm, por alguma razão, o valor lógico 'X' (ou 'Z'), este valor é então propagado para a saída, de forma que `mul_b` também terá o valor lógico 'X' (ou 'Z'), não contando transições do tipo 0→1 ou 1→0. No entanto, na simulação SystemC, este comportamento não é ainda detectado pela PowerSC para os tipos de dados mencionados, porque seus valores representam apenas '0' ou '1'. Portanto, uma atribuição, como essa mostrada na figura 5.1, irá contribuir na contagem total de transições.

Para os experimentos *MP3-DCT* e *FIR*, vemos uma diferença de 15% e 33%, respectivamente. Esta expressiva diferença se dá, principalmente, por causa de uma limitação na *interface* do Power Compiler com o ModelSim: ele é incapaz de capturar informação de atividade de transição em bancos de registradores. Este é um problema documentado pela Synopsys em seu serviço de suporte online, SolvNet [38], como documento número 903543.

Tanto o modelo *MP3-DCT*, quanto *FIR* possuem bancos de registradores em suas descrições e o erro mostrado na tabela se refere a esta limitação do Power Compiler. Abaixo é mostrada a construção que não é suportada pelo Power Compiler:

```
reg [31:0] buff [31:0];
```

A construção equivalente em SystemC é:

```
sc_uint<32> buff[32];
```

que é suportada sem problemas pela PowerSC.

Esta limitação do Power Compiler pode resultar numa grande sub-estimação para alguns circuitos, como mostrado, o que pode levar a uma baixa precisão na hora de calcular a potência consumida. Quando retiramos este suporte da PowerSC, os resultados finais tornam-se similares. Note que é a PowerSC que está correta.

Os resultados apresentados nas tabelas 5.1 e 5.2 mostram que, usando somente a PowerSC, é possível obter-se resultados muito semelhantes aos das ferramentas comerciais Power Compiler e ModelSim, inclusive superando uma limitação do Power Compiler.

5.2 Avaliação do Algoritmo SMS

Nesta categoria de experimentos, são apresentados os resultados obtidos utilizando-se o método de simulação com monitoração por amostragem (SMS), introduzido no capítulo 4. Foram extraídos dois modelos significativos do primeiro conjunto de experimentos da primeira categoria para a validação do algoritmo SMS. O primeiro é o módulo MP3-DCT, escolhido por representar um modelo real, e validado em *hardware*. O segundo é o somador completo (*ripple-carry*) de 16 bits, escolhido por ser possível controlar seu comportamento facilmente, de forma a testar algumas características do nosso algoritmo, como será mostrado.

5.2.1 Configurações dos Parâmetros

Para os módulos escolhidos para os experimentos com o SMS, diversos testes foram realizados com relação à variação dos parâmetros do algoritmo. Todas as configurações de parâmetros utilizadas podem ser vistas na tabela 5.3.

Variação de Param.	**max_mon**	**max_pred**	**threshold**	**num_first_samples**
Default	128	256	0,005	128
First Samples	128	256	0,005	16
				256
				512
Samples Range	512	512	0,005	128
	256	128		
	64	64		
	2048	4096		
	4096	2048		
Threshold	128	256	0,0005	128
			0,01	
			0,02	
			0,05	

Tabela 5.3: Variação dos parâmetros do SMS para os experimentos

A primeira coluna desta tabela mostra o nome do teste, que identifica qual conjunto de parâmetros está sendo variado nos experimentos. As colunas remanescentes mostram os valores para cada parâmetro considerado nos testes.

Os parâmetros *min_mon_samples* e *min_pred_samples* não são mostrados na tabela, pois seus valores não são modificados e o valor padrão para estes parâmetros é *1*. Os testes mostrados na tabela 5.3 são os seguintes:

- **Default**: utilizados os padrões de configuração dos parâmetros;

- **First Samples**: apenas o número de amostras iniciais é modificado em relação ao *Default*;
- **Samples Range**: varia-se o número de máximo de amostras monitoradas e não monitoradas permitidas e
- **Threshold**: somente o parâmetro *threshold* é alterado.

Além da variação nos valores dos parâmetros do algoritmo SMS, alguns testes específicos foram aplicados em cada módulo, como será visto.

5.2.2 Experimentos com o Módulo MP3-DCT

Para o MP3-DCT, foram escolhidos três estilos musicais diferentes: música clássica, *heavy metal* e *blues*. O objetivo desta escolha é o de prover um conjunto representativo de entradas reais para o MP3, que capture as peculiaridades de cada estilo musical.

Ademais, duas músicas para cada estilo foram selecionadas, com um tempo aproximado de 2 minutos por música. Como o módulo MP3-DCT não é o IP completo do decodificador de MP3, os arquivos em MP3 das músicas selecionadas foram decodificadas por *software* até o ponto exatamente anterior às entradas do módulo DCT.

Os resultados dos testes *First Samples*, *Samples Range* e *Threshold* com o módulo MP3-DCT podem ser vistos nas tabelas 5.4, 5.5 e 5.6, respectivamente. A primeira coluna destas tabelas mostra as músicas utilizadas como entrada para o módulo, nomeadas de *M1-a* até *M6-a*. A segunda coluna (α) indica qual o valor específico do parâmetro principal do teste está sendo usado. Para *First Samples*, α é o parâmetro *num_first_samples*, para *Samples Range* α é igual a *(max_pred, max_mon)* e, finalmente, para *Threshold* α indica o valor do parâmetro *threshold*.

A terceira coluna mostra os resultados obtidos para cada teste, e é subdivida em duas colunas: *T.A. (Tempo Amostrado)*, que indica o tempo total que a simulação foi efetivamente monitorada e *Erro*, que mostra o erro da simulação com monitoração por amostragem em relação a simulação com monitoração total. A quarta coluna também é subdivida em duas colunas e mostra os resultados obtidos usando-se os padrões de configuração dos parâmetros.

Como se pode ver pelos dados nas tabelas, para todos os testes com o módulo MP3-DCT, o erro não passou de *0,135%* e, para a maior parte dos casos, o tempo monitorado foi reduzido significativamente. Esta redução do tempo monitorado se dá pelo fato que a média global da atividade de transição converge durante a decodificação das músicas. O algoritmo detecta esta convergência e reduz o número de amostras monitoradas até o mínimo, quando possível. Utilizando os padrões de configuração (coluna *Default*) para os parâmetros, os tempos de amostragem variaram de 43% (M6-a) a 87% (M1-a).

Experimento	α	First Samples		Default	
		T.A.	Erro	T.A.	Erro
Clássica M1-a	16	88%	0,005%	87%	0,003%
	256	87%	0,012%		
	512	85%	0,009%		
Clássica M2-a	16	84%	0,003%	83%	0,003%
	256	85%	0,003%		
	512	81%	0,008%		
Heavy Metal M3-a	16	37%	0,030%	40%	0,048%
	256	39%	0,048%		
	512	39%	0,041%		
Heavy Metal M4-a	16	52%	0,076%	51%	0,078%
	256	54%	0,064%		
	512	53%	0,061%		
Blues M5-a	16	80%	0,028%	82%	0,031%
	256	81%	0,031%		
	512	76%	0,028%		
Blues M6-a	16	46%	0,070%	43%	0,089%
	256	46%	0,056%		
	512	43%	0,089%		

Tabela 5.4: Resultados para o MP3-DCT usando SMS (*First Samples*)

Para os dois primeiros estilos musicais (clássica e *heavy metal*), os resultados utilizando estas configurações foi parecido, indicando uma certa similaridade no comportamento do circuito quando decodificando músicas destes estilos. Para o *blues*, pode-se ver que o tempo de amostragem teve uma diferença grande no tempo de monitoração.

Nos resultados específicos da tabela 5.4, a influência obtida ao se aumentar o número de amostras iniciais foi a de reduzir um pouco o tempo de amostragem para maior parte dos casos. Isto ocorre pelo fato de que, com mais amostras iniciais obtidas, a primeira média global utilizada pelo algoritmo se torna mais precisa e, como a média global converge durante a decodificação, isto acaba por reduzir o tempo de monitoração nestes casos.

Nos testes com a variação nos limites de amostras monitoradas e não monitoradas (tabela 5.5) há uma maior influência no tempo de monitoração em configurações distintas. É possível notar que quando é utilizada a configuração *(max_pred, max_mon) = (64, 64)* o tempo de monitoração é reduzido para M1-a, M2-a e M5-a. Estas músicas têm o maior tempo monitorado dentre as músicas da tabela e, como o número máximo de amostras monitoradas foi reduzido, mais janelas não monitoradas ocorrem durante a simulação, mesmo que pequenas e, por isso, o tempo de amostragem total foi reduzido.

Experimento	α	**Samples Range**		**Default**	
		T.A.	**Erro**	**T.A.**	**Erro**
Clássica M1-a	512/512	89%	0,020%	87%	0,003%
	128/256	92%	0,007%		
	64/64	86%	0,015%		
	4096/2048	95%	0,001%		
	2048/4096	96%	0,007%		
Clássica M2-a	512/512	79%	0,027%	83%	0,003%
	128/256	86%	0,004%		
	64/64	86%	0,003%		
	4096/2048	91%	0,001%		
	2048/4096	98%	0,015%		
Heavy Metal M3-a	512/512	39%	0,055%	40%	0,048%
	128/256	44%	0,044%		
	64/64	46%	0,045%		
	4096/2048	38%	0,025%		
	2048/4096	40%	0,004%		
Heavy Metal M4-a	512/512	43%	0,091%	51%	0,078%
	128/256	59%	0,031%		
	64/64	59%	0,048%		
	4096/2048	51%	0,091%		
	2048/4096	51%	0,101%		
Blues M5-a	512/512	88%	0,011%	82%	0,031%
	128/256	87%	0,018%		
	64/64	77%	0,025%		
	4096/2048	89%	0,022%		
	2048/4096	88%	0,021%		
Blues M6-a	512/512	42%	0,110%	43%	0,089%
	128/256	57%	0,044%		
	64/64	56%	0,036%		
	4096/2048	29%	0,048%		
	2048/4096	46%	0,075%		

Tabela 5.5: Resultados para o MP3-DCT usando SMS (*Samples Range*)

O contrário ocorre para as músicas M3-a, M4-a e M6-a, isto é, mais janelas monitoradas ocorrem durante a simulação, aumentando o tempo de amostragem.

Com um número maior de amostras permitidas, *(max_pred, max_mon) = (4096, 2048)*, por exemplo, pode-se notar que ocorre um aumento no tempo de monitoração para as músicas M1-a, M2-a e M5-a, enquanto ocorre uma redução para M3-a, M4-a e M6-a. O aumento ocorre para o primeiro, pois estas músicas são as que têm a maior variação nas médias locais e, por isso, um limite maior para o número de amostras monitoradas leva o algoritmo a crescer mais as janelas de monitoração, aumentando o tempo de amostragem. No segundo caso, onde há redução no tempo de monitoração, ocorre o inverso, isto é, as janelas não monitoradas crescem mais.

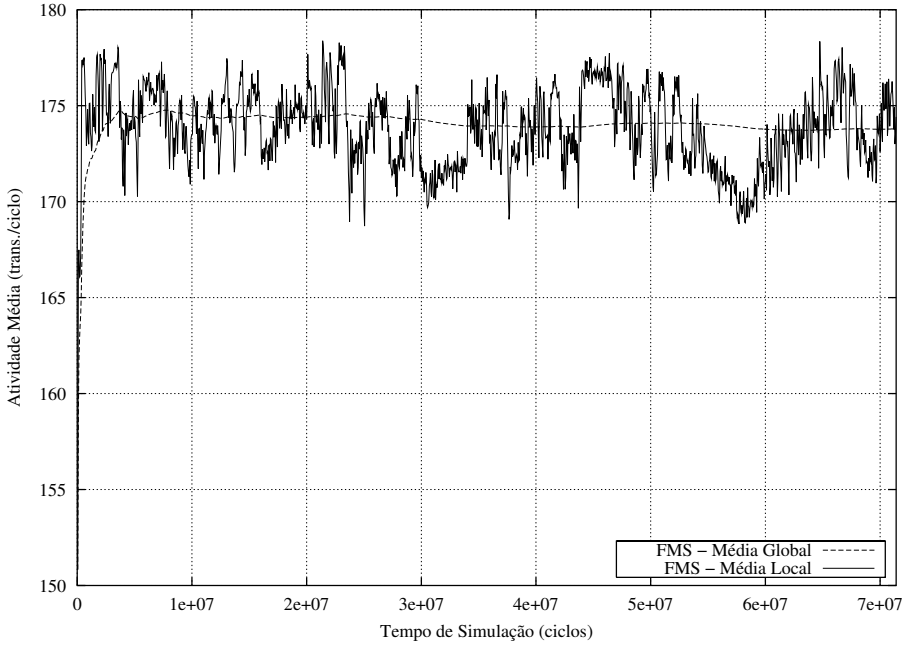
Experimento	α	**Threshold**		**Default**	
		T.A.	**Erro**	**T.A.**	**Erro**
Clássica M1-a	0,0005	99%	0,001%	87%	0,003%
	0,01	34%	0,017%		
	0,02	4%	0,087%		
	0,05	1%	0,135%		
Clássica M2-a	0,0005	99%	0,000%	83%	0,003%
	0,01	47%	0,009%		
	0,02	16%	0,126%		
	0,05	1%	0,072%		
Heavy Metal M3-a	0,0005	99%	0,000%	40%	0,048%
	0,01	13%	0,076%		
	0,02	2%	0,037%		
	0,05	1%	0,078%		
Heavy Metal M4-a	0,0005	99%	0,000%	51%	0,078%
	0,01	20%	0,119%		
	0,02	10%	0,050%		
	0,05	2%	0,128%		
Blues M5-a	0,0005	99%	0,000%	82%	0,031%
	0,01	32%	0,115%		
	0,02	8%	0,088%		
	0,05	2%	0,036%		
Blues M6-a	0,0005	99%	0,000%	43%	0,089%
	0,01	9%	0,116%		
	0,02	4%	0,035%		
	0,05	1%	0,039%		

Tabela 5.6: Resultados para o MP3-DCT usando SMS (*Threshold*)

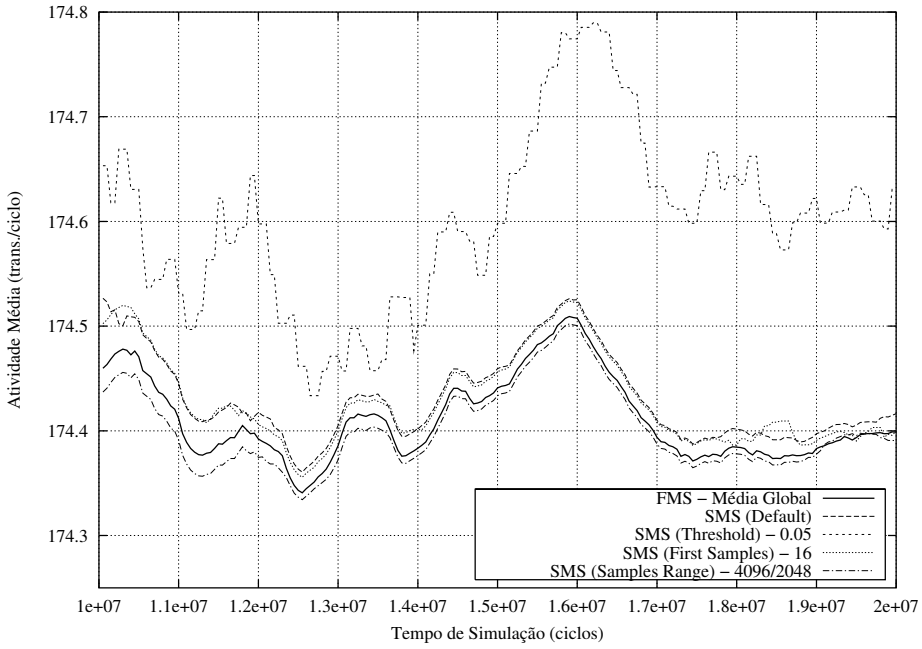
Na tabela 5.6 os resultados para os testes com a variação do parâmetro *threshold* são

mostrados, e este parâmetro é o que tem a maior influência no tempo de amostragem. Em todos os casos, quando o *threshold* é relaxado para 5% (0,05), o tempo total monitorado é de não mais que 2%, com um erro máximo irrelevante de apenas 0,135%. Considere agora quando o *threshold* é apertado para 0,05% (0,0005). Nestes casos, o tempo total de amostragem foi de aproximadamente 99%, praticamente durante toda a simulação. Isto acontece porque, quando é utilizado um *threshold* tão apertado, a média local pode variar apenas ±0,05% da média global, evitando assim que o número de amostras não monitoradas aumente.

As figuras 5.3–5.8 mostram os gráficos com a atividade de transição para as músicas M1-a–M6-a. As subfiguras 5.3(a)–5.8(a) apresentam as médias global e local (em intervalos de 50K ciclos) durante a decodificação das músicas utilizando o método FMS, enquanto os gráficos das figuras 5.3(b)–5.8(b) mostram algumas estimativas com o algoritmo SMS usando diferentes parâmetros durante um intervalo da simulação (de 10M a 20M ciclos). Note a escala destes gráficos, que foi aumentada, mostrando que as estimativas usando SMS são bem próximas dos resultados utilizando a simulação com monitoração total.

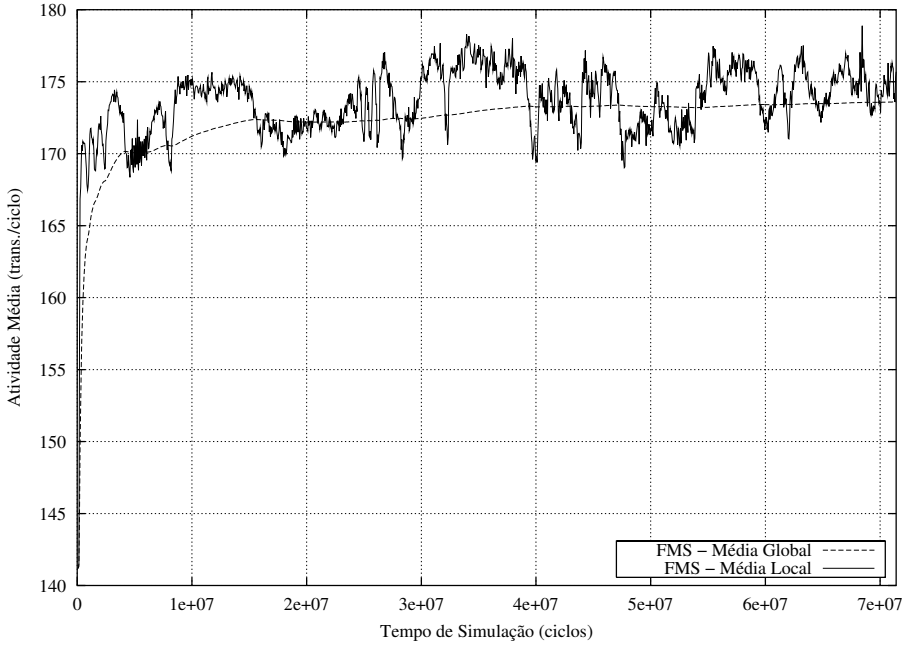


(a) Usando FMS, mostrando as médias global e local

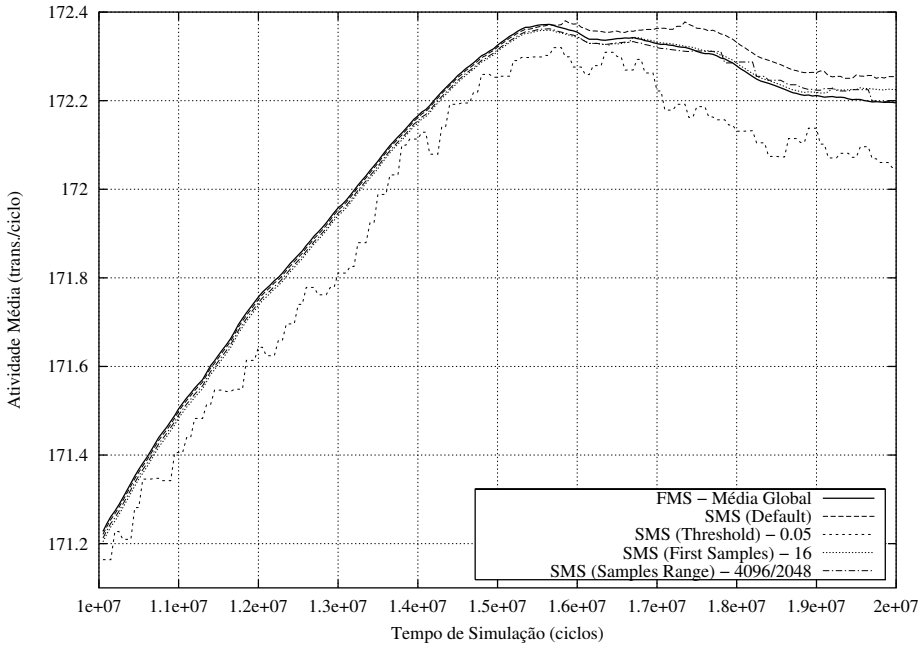


(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.3: Atividade de transição para música M1-a durante a simulação

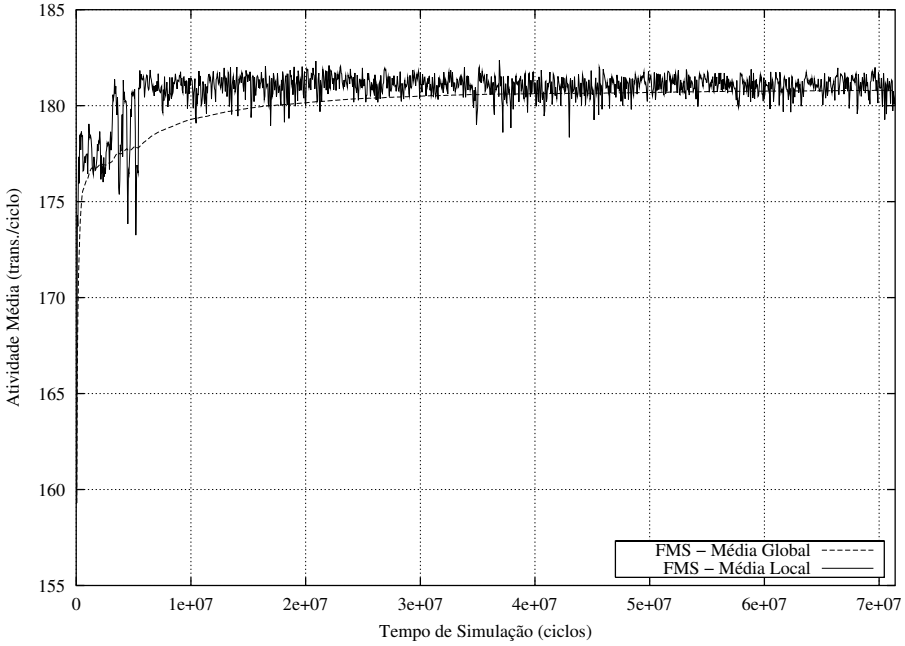


(a) Usando FMS, mostrando as médias global e local

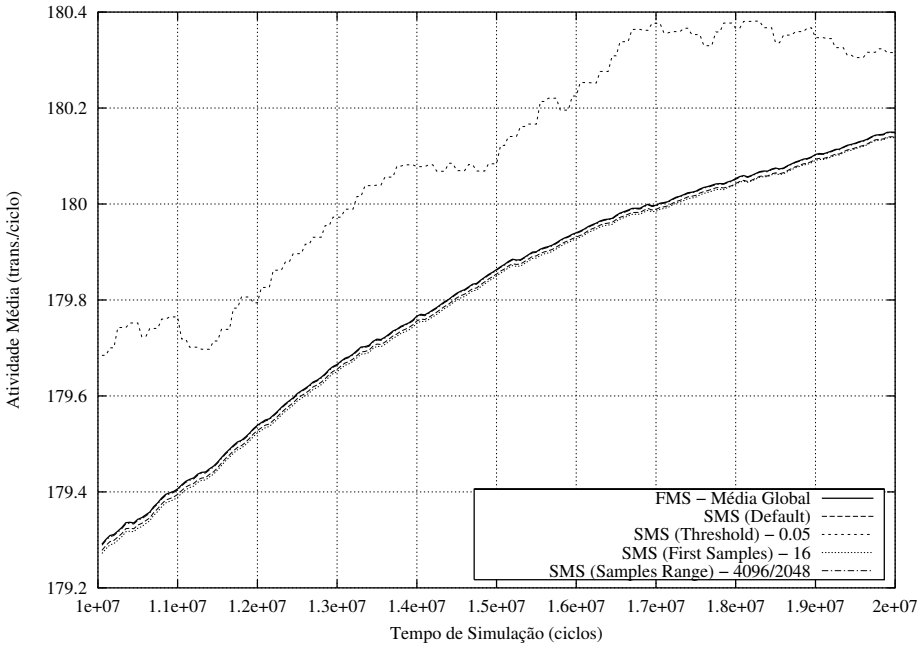


(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.4: Atividade de transição para música M2-a durante a simulação

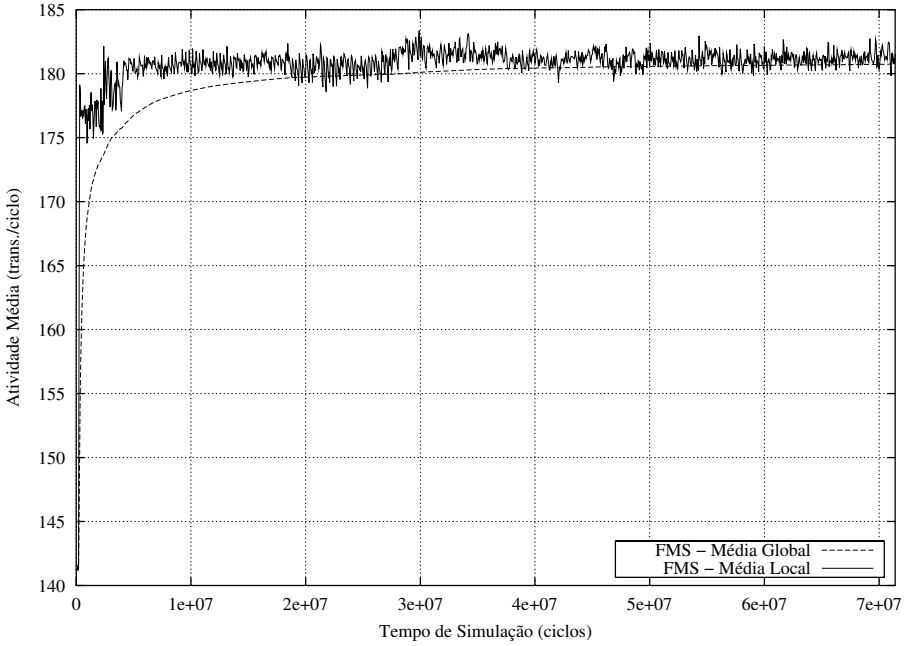


(a) Usando FMS, mostrando as médias global e local

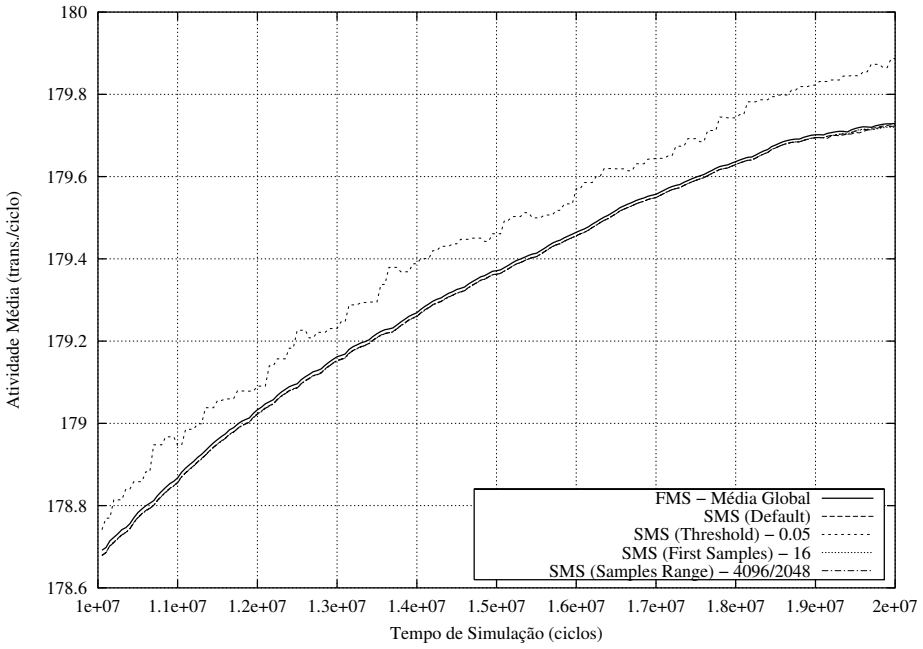


(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.5: Atividade de transição para música M3-a durante a simulação

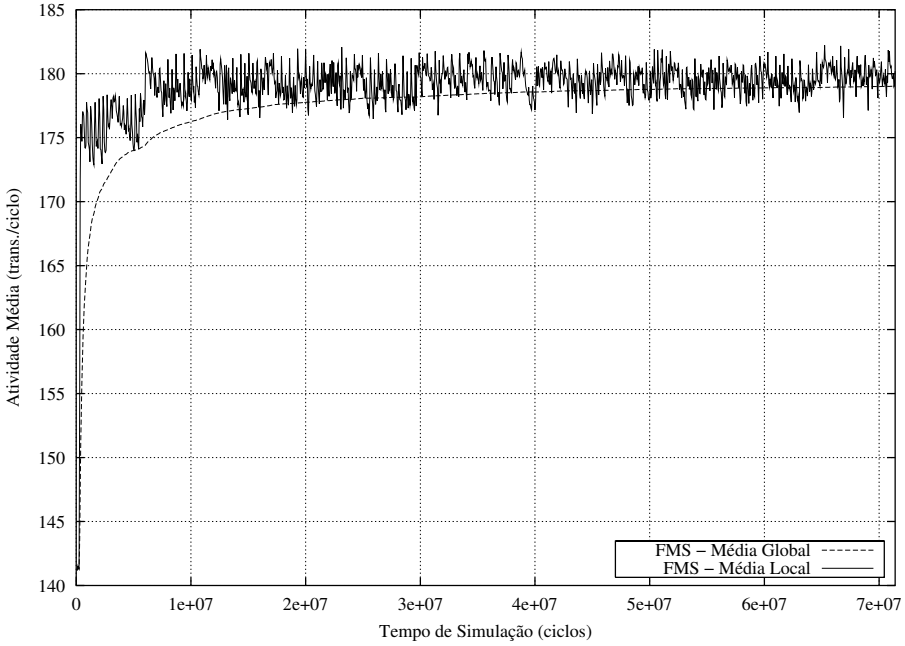


(a) Usando FMS, mostrando as médias global e local

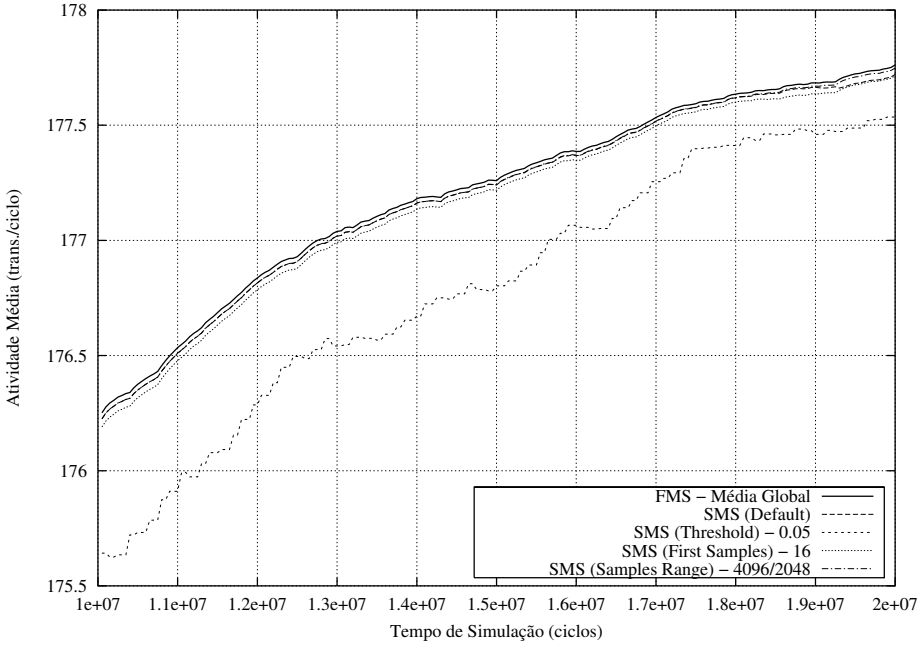


(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.6: Atividade de transição para música M4-a durante a simulação

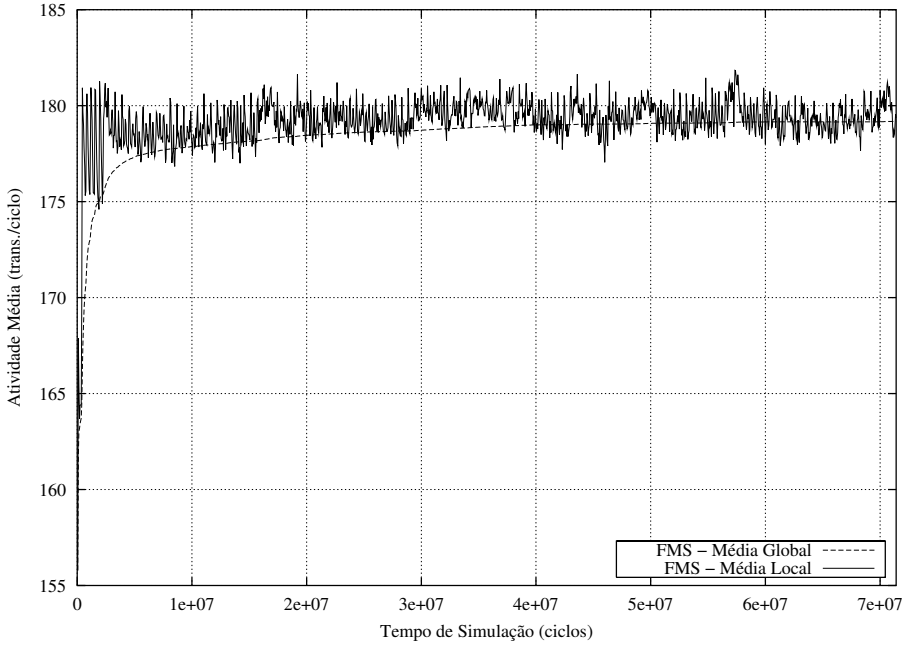


(a) Usando FMS, mostrando as médias global e local

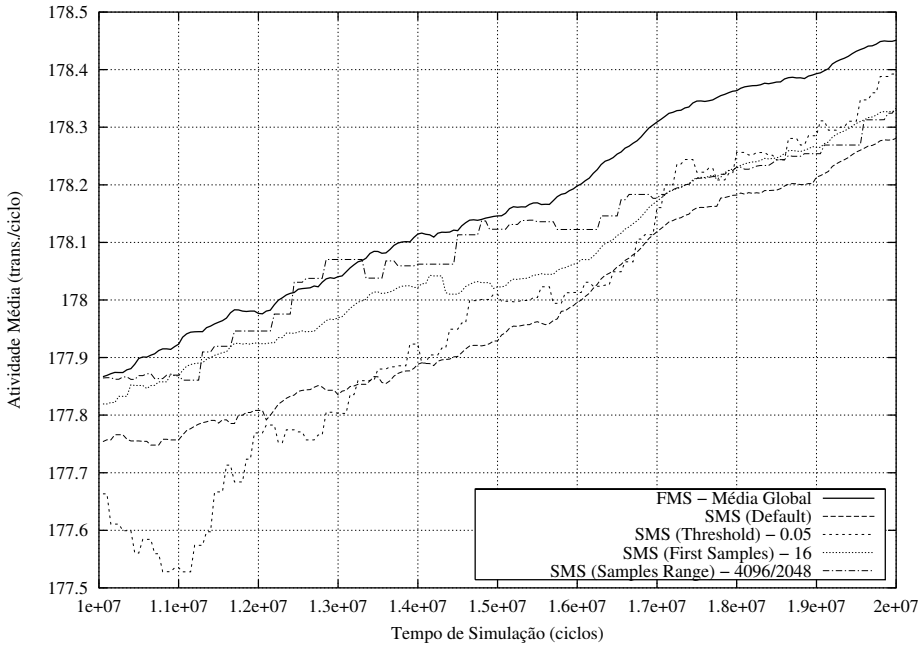


(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.7: Atividade de transição para música M5-a durante a simulação



(a) Usando FMS, mostrando as médias global e local



(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.8: Atividade de transição para música M6-a durante a simulação

Simulação de Pausa na Decodificação

Outro teste realizado com o módulo MP3-DCT foi fazer com que, durante um grande intervalo de tempo da decodificação da música, fossem congeladas as entradas do módulo com o valor zero. Isto seria uma forma artificial de simular uma pausa no circuito e, desta forma, poder verificar a reação do algoritmo a este comportamento. Os resultados para este teste podem ser vistos nas tabelas 5.7, 5.8 e 5.9.

Experimento	α	**First Samples**		**Default**	
		T.A.	**Erro**	**T.A.**	**Erro**
Clássica M1-b	16	93%	0,003%	93%	0,001%
	256	92%	0,003%		
	512	91%	0,001%		
Clássica M2-b	16	92%	0,003%	91%	0,004%
	256	93%	0,008%		
	512	91%	0,003%		
Heavy Metal M3-b	16	86%	0,039%	89%	0,009%
	256	88%	0,009%		
	512	88%	0,032%		
Heavy Metal M4-b	16	95%	0,009%	94%	0,012%
	256	94%	0,009%		
	512	94%	0,005%		
Blues M5-b	16	94%	0,017%	96%	0,014%
	256	96%	0,010%		
	512	97%	0,010%		
Blues M6-b	16	83%	0,036%	82%	0,033%
	256	83%	0,039%		
	512	83%	0,033%		

Tabela 5.7: Simulando uma pausa na decodificação em MP3-DCT (*First Samples*)

As colunas estão organizadas de forma semelhante às tabelas do teste anterior, e as músicas são as mesmas utilizadas para o primeiro teste com o módulo MP3-DCT. Porém, como aqui há uma parada no meio da decodificação das mesmas, então a nomeação é feita de forma diferente (M1-b, ..., M6-b). Os gráficos das figuras 5.9(a)–5.14(a) mostram a diferença induzida na decodificação das músicas. Note que, aproximadamente no meio da simulação, ocorre uma queda brusca na atividade de transição do circuito e a média local capturada fica baixa por um intervalo de tempo, forçando a média global a cair. Após este intervalo, a decodificação continua normalmente e a média global volta a subir.

Experimento	α	**Samples Range**		**Default**	
		T.A.	**Erro**	**T.A.**	**Erro**
Clássica M1-b	512/512	95%	0,008%	93%	0,001%
	128/256	94%	0,007%		
	64/64	93%	0,004%		
	4096/2048	96%	0,002%		
	2048/4096	97%	0,007%		
Clássica M2-b	512/512	94%	0,001%	91%	0,004%
	128/256	94%	0,007%		
	64/64	93%	0,001%		
	4096/2048	93%	0,000%		
	2048/4096	98%	0,014%		
Heavy Metal M3-b	512/512	90%	0,003%	89%	0,009%
	128/256	89%	0,001%		
	64/64	88%	0,016%		
	4096/2048	90%	0,034%		
	2048/4096	91%	0,106%		
Heavy Metal M4-b	512/512	89%	0,018%	94%	0,012%
	128/256	97%	0,001%		
	64/64	93%	0,009%		
	4096/2048	94%	0,007%		
	2048/4096	92%	0,037%		
Blues M5-b	512/512	99%	0,007%	96%	0,014%
	128/256	98%	0,005%		
	64/64	94%	0,013%		
	4096/2048	98%	0,008%		
	2048/4096	100%	0,003%		
Blues M6-b	512/512	83%	0,036%	82%	0,033%
	128/256	90%	0,025%		
	64/64	86%	0,021%		
	4096/2048	75%	0,044%		
	2048/4096	89%	0,030%		

Tabela 5.8: Simulando uma pausa na decodificação em MP3-DCT (*Samples Range*)

Experimento	α	**Threshold**		**Default**	
		T.A.	**Erro**	**T.A.**	**Erro**
Clássica M1-b	0,0005	99%	0,000%	93%	0,001%
	0,01	72%	0,030%		
	0,02	58%	0,052%		
	0,05	28%	0,005%		
Clássica M2-b	0,0005	99%	0,001%	91%	0,004%
	0,01	81%	0,008%		
	0,02	69%	0,098%		
	0,05	35%	0,098%		
Heavy Metal M3-b	0,0005	99%	0,001%	89%	0,009%
	0,01	69%	0,040%		
	0,02	60%	0,017%		
	0,05	41%	0,096%		
Heavy Metal M4-b	0,0005	99%	0,002%	94%	0,012%
	0,01	75%	0,049%		
	0,02	66%	0,039%		
	0,05	43%	0,095%		
Blues M5-b	0,0005	99%	0,001%	96%	0,014%
	0,01	80%	0,001%		
	0,02	65%	0,007%		
	0,05	38%	0,061%		
Blues M6-b	0,0005	99%	0,002%	82%	0,033%
	0,01	66%	0,036%		
	0,02	61%	0,011%		
	0,05	37%	0,045%		

Tabela 5.9: Simulando uma pausa na decodificação em MP3-DCT (*Threshold*)

Note que os tempos de amostragem aumentaram bastante nestes testes e, com os padrões de configuração, os tempos ficaram entre 82% e 96%. Isto ocorre pois a queda brusca na atividade de transição do circuito é detectada pelo algoritmo, indicando um período de instabilidade e, desta forma, as janelas monitoradas são aumentadas, reduzindo-se as janelas de não monitoração. Note que os erros também ficaram menores, mesmo com estes períodos de instabilidade do circuito, isto porque o tempo de amostragem aumentou, evitando desta forma que a precisão das estimativas decaísse.

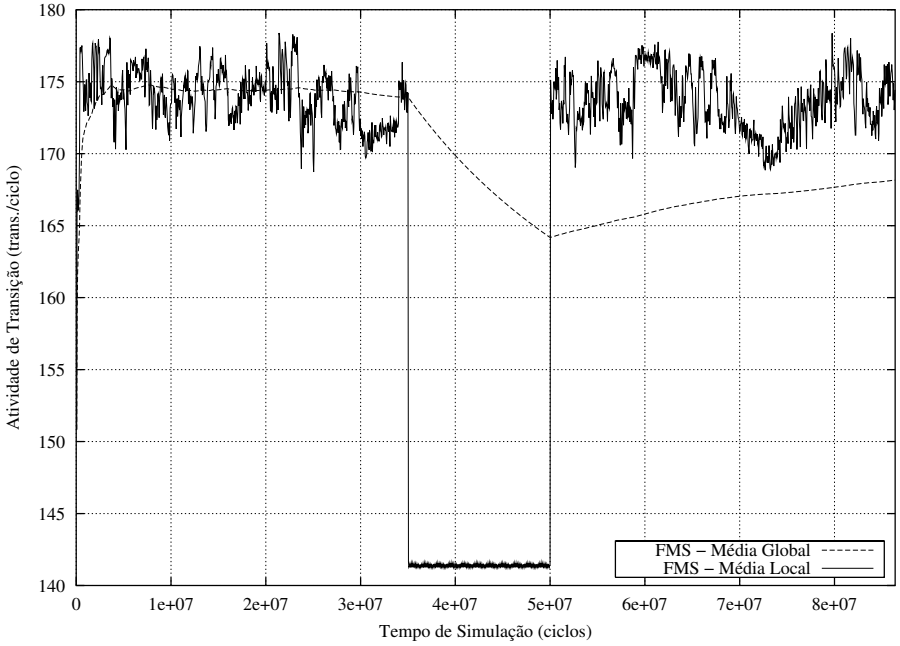
Em alguns resultados das tabelas, o tempo amostrado aparece como sendo 100%, porém isto é um arredondamento do tempo amostrado e não deve ser tomado como valor absoluto.

Na tabela 5.9, o comportamento é muito similar ao que ocorre para a decodificação sem

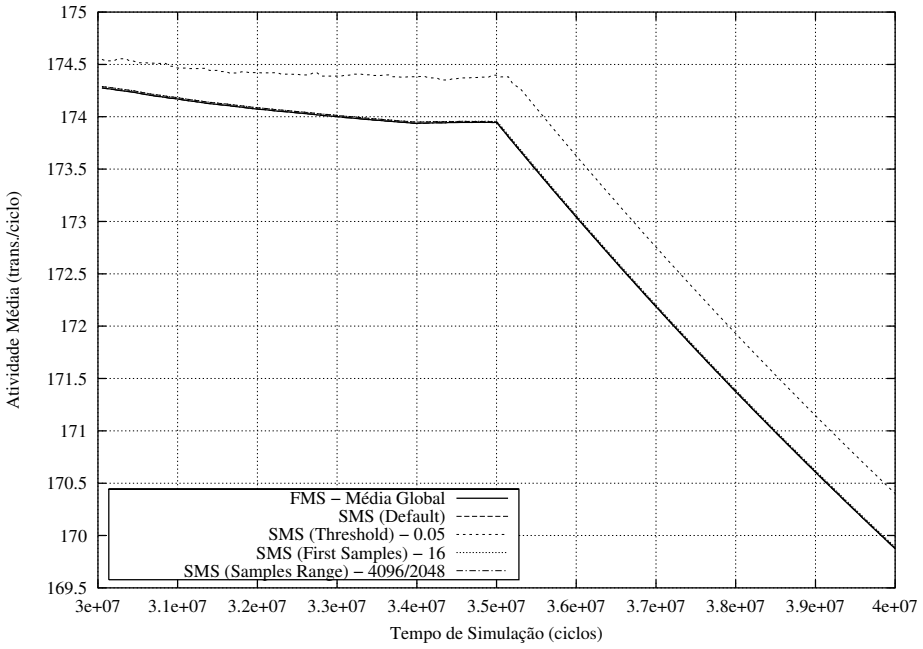
a pausa, a diferença é que, relaxando o *threshold* para 5% (0.05), como já era esperado, o tempo amostrado não reduz na mesma proporção que antes, mas mesmo assim consegue-se diminuir para até 28% este valor.

As figuras 5.9(b)–5.14(b) mostram gráficos com algumas estimativas utilizando o algoritmo SMS para as músicas das figuras 5.9(a)–5.14(a). Nestas figuras é mostrado o intervalo de 30M a 40M ciclos, de forma a enfatizar o intervalo no qual ocorre a forte queda na atividade de transição.

Como pôde ser visto nestes dois testes com o módulo MP3-DCT, o algoritmo SMS consegue reduzir bastante o tempo de amostragem necessário durante a simulação, detectando a estabilidade do circuito com uma perda mínima na precisão com relação ao algoritmo FMS.

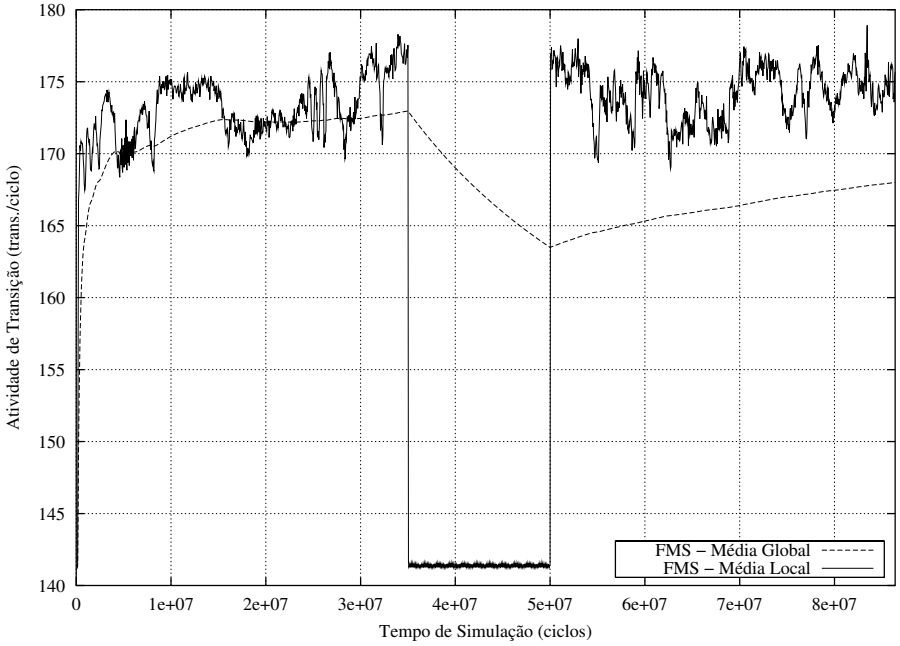


(a) Usando FMS, mostrando as médias global e local

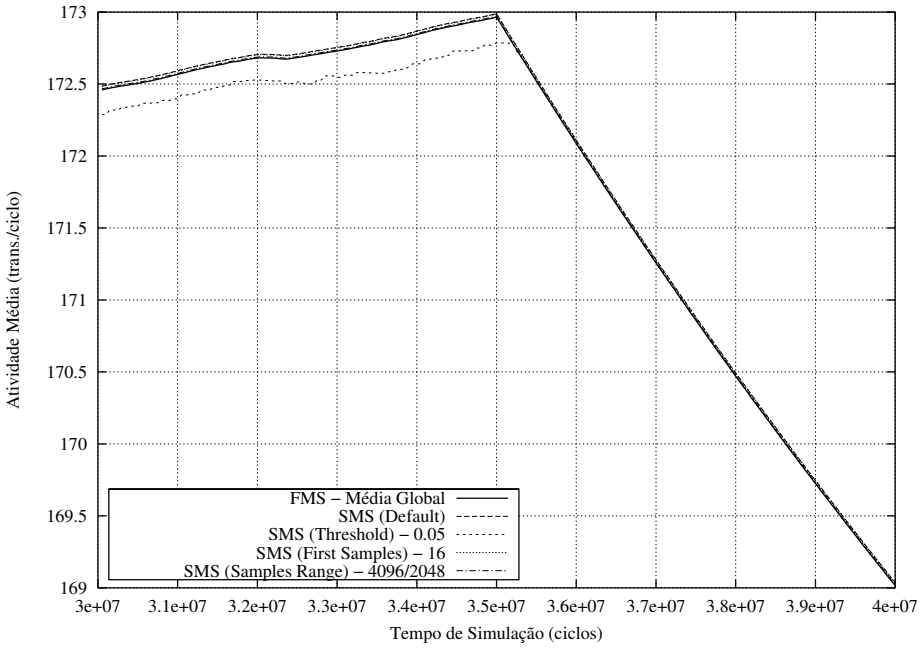


(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.9: Atividade de transição para música M1-b durante a simulação

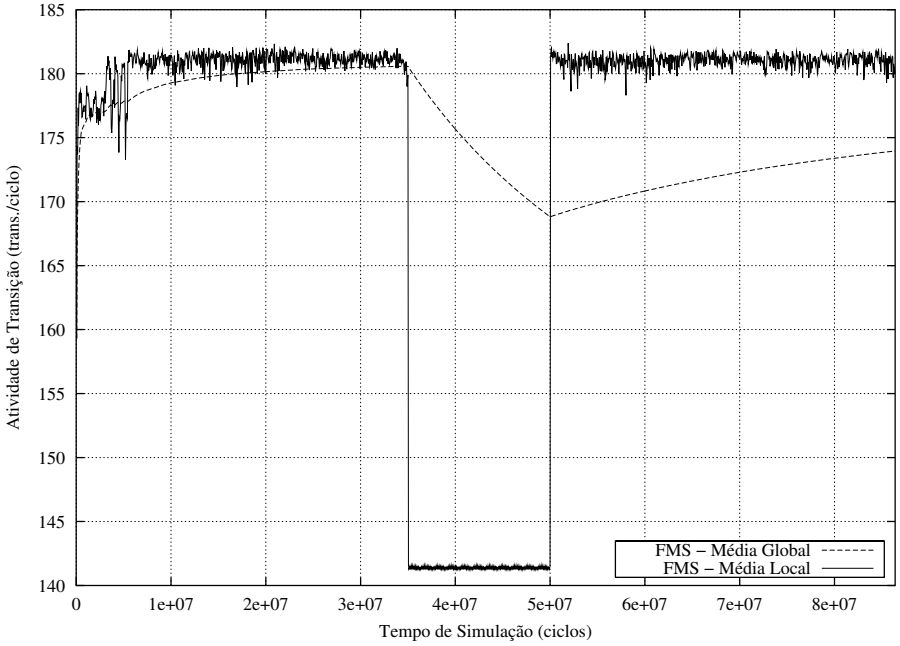


(a) Usando FMS, mostrando as médias global e local

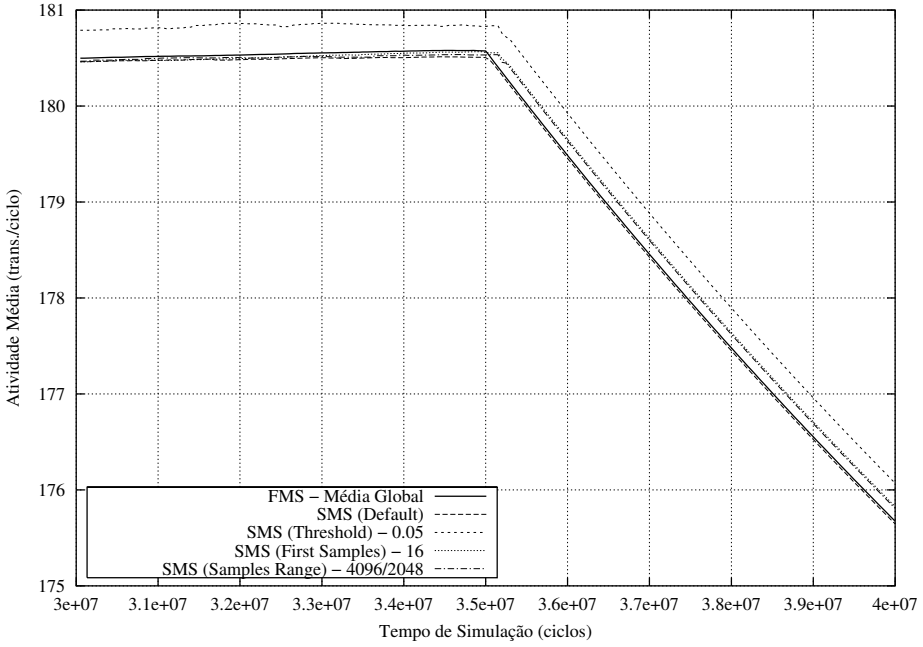


(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.10: Atividade de transição para música M2-b durante a simulação

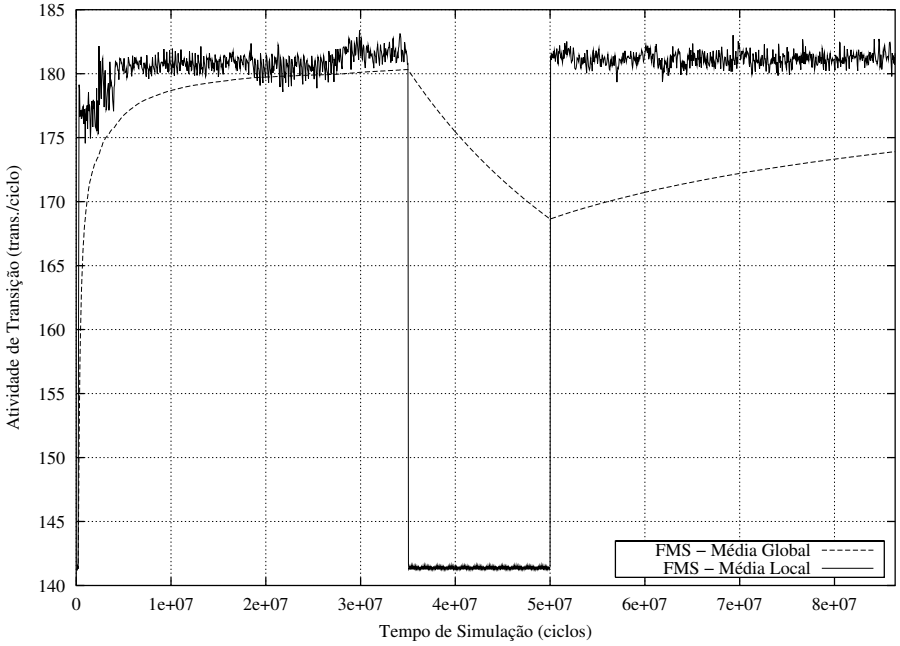


(a) Usando FMS, mostrando as médias global e local

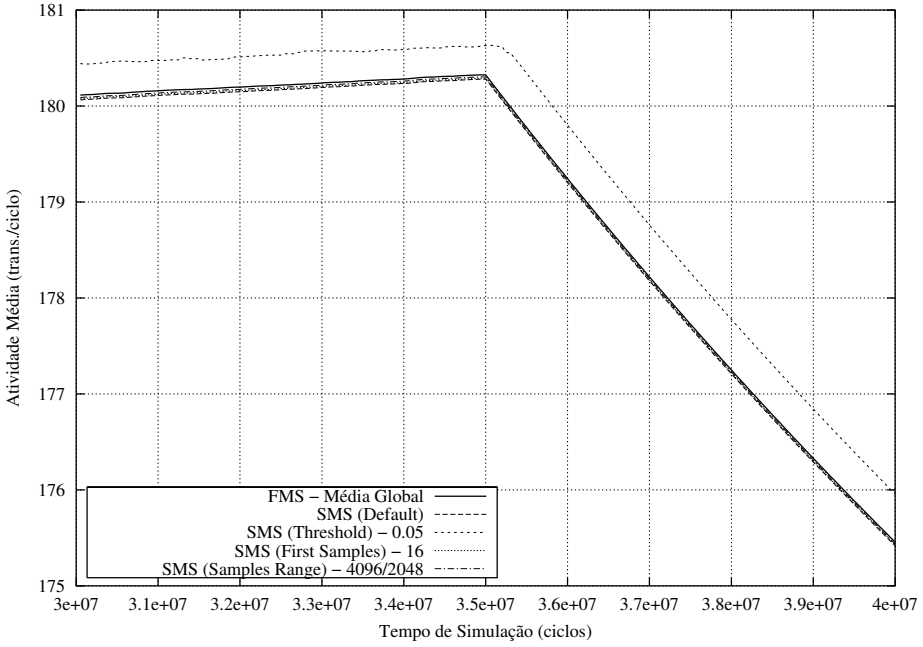


(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.11: Atividade de transição para música M3-b durante a simulação

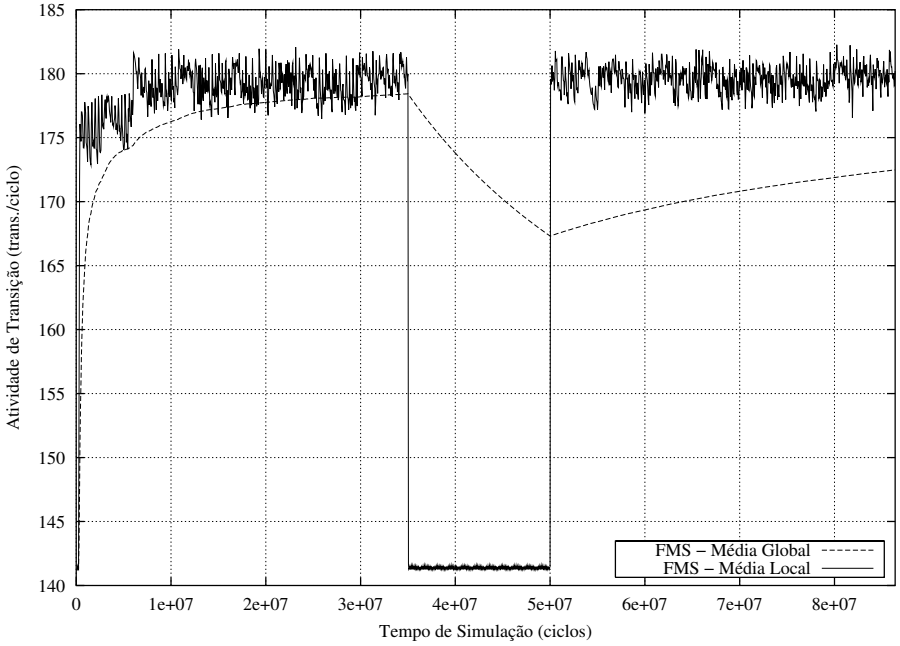


(a) M4-b usando FMS mostrando as médias global e local

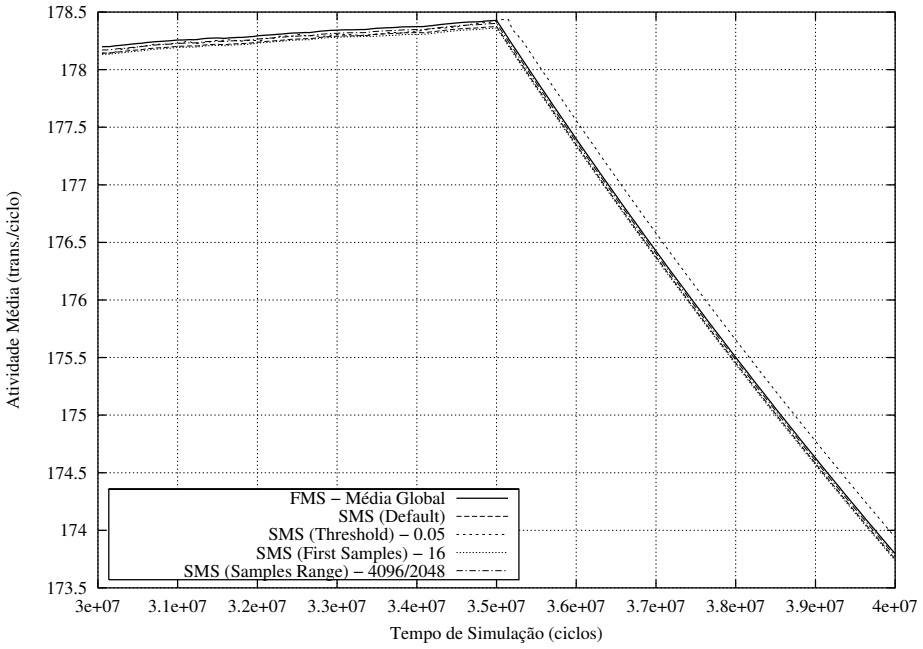


(b) M4-b usando SMS mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.12: Atividade de transição para música M4-b durante a simulação

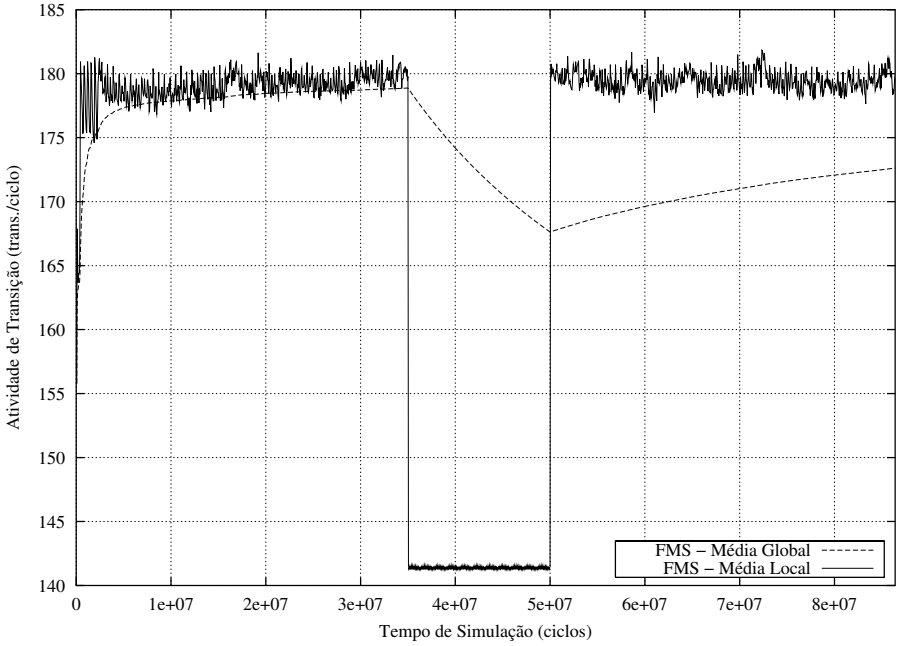


(a) Usando FMS, mostrando as médias global e local

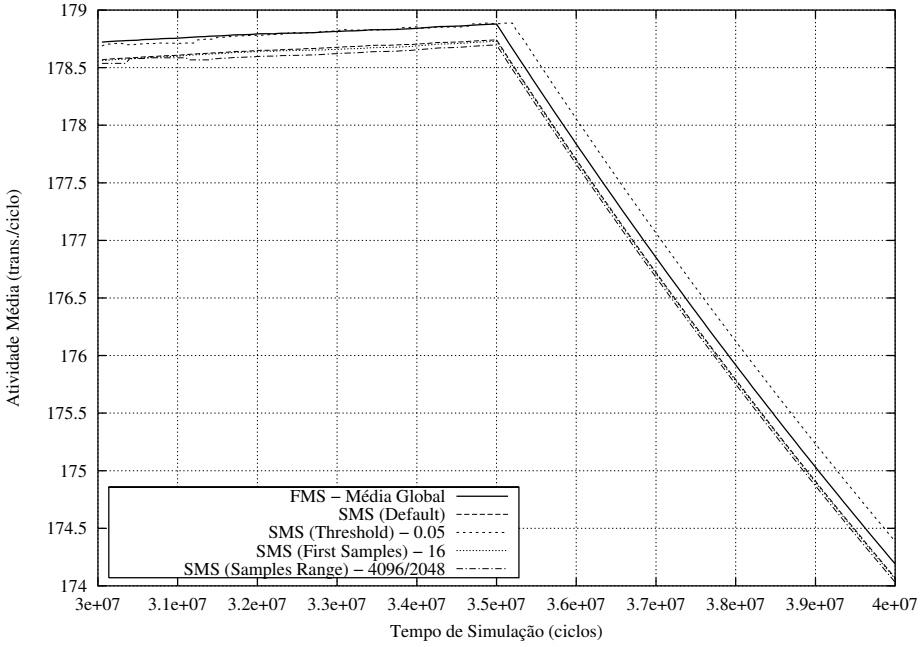


(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.13: Atividade de transição para música M5-b durante a simulação



(a) Usando FMS, mostrando as médias global e local



(b) Usando SMS, mostrando algumas configurações de parâmetros durante intervalo da simulação

Figura 5.14: Atividade de transição para música M6-b durante a simulação

5.2.3 Experimentos com o Módulo Somador *Ripple-Carry*

Para o módulo do somador *ripple-carry*, números pseudo-aleatórios são gerados para suas entradas (são realizados testes com cinco sementes diferentes) a cada ciclo durante a simulação. A diferença nos testes para o somador está em como estes números são gerados. O objetivo disto é induzir algum comportamento específico ao circuito. A geração destes números é feita como descrito abaixo:

- **T1 – random bit-width**: a largura de bits dos números gerados é modificada regularmente e de forma aleatória (ex., 2, 6, 9, 15, 3, 8,...);
- **T2 – increasing bit-width**: a largura de bits dos números é aumentada regularmente e, quando alcançar a largura máxima, o processo é reiniciado com a largura mínima (2, 3, 4, 5,...,16, 2, 3, 4,...);
- **T3 – decreasing bit-width**: a largura de bits dos números é diminuída regularmente e, quando a largura mínima é alcançada, o processo é reiniciado com a largura máxima (16, 15,...,3, 2, 16,...);
- **T4 – freeze bit-width**: a largura de bits dos números sorteados é alternada periodicamente entre 2 e 16 (2, 16, 2, 16,...) e
- **T5 – freeze inputs**: as entradas são temporariamentes congeladas durante a simulação, forçando uma queda brusca na atividade de transição global, simulando um efeito de *standby*.

O comportamento do circuito para cada um destes testes é mostrado nas figuras 5.15(a)–5.19(a), nesta ordem.

Os resultados obtidos para T1–T5 podem ser vistos nas tabelas 5.10–5.12 (as médias dos resultados com as cinco sementes são apresentadas). Para a maioria destes testes (T1 a T4), o erro máximo obtido não ultrapassou 0.780% e também o tempo amostrado foi bastante alto, no mínimo 88%. Este alto tempo de monitoração já era esperado, pois, como pode ser visto pelas figuras 5.15(a)–5.18(a), o comportamento destes circuitos é instável durante todo o tempo de simulação. Estes resultados mostram que esse comportamento instável do circuito é eficientemente detectado pelo algoritmo, e o número de amostras monitoradas é mantido alto durante a maior parte da simulação.

O teste T5 é ligeiramente diferente dos testes T1–T4. A atividade de transição é mantida estável por algum tempo durante a simulação, e então os valores das entradas são "congelados", forçando a média global a cair rapidamente. Isto é mostrado na figura 5.19(a). O erro obtido para este teste foi maior do que para os outros, chegando ao máximo de 9.628%, e isto é explicado mais à frente.

Experimento	α	First Samples		Default	
		T.A.	Erro	T.A.	Erro
T1	16	98%	0,225%	99%	0,048%
	256	99%	0,008%		
	512	99%	0,002%		
T2	16	99%	0,027%	99%	0,013%
	256	99%	0,015%		
	512	99%	0,000%		
T3	16	98%	0,029%	99%	0,009%
	256	99%	0,006%		
	512	99%	0,005%		
T4	16	98%	0,591%	99%	0,124%
	256	99%	0,006%		
	512	99%	0,001%		
T5	16	71%	3,211%	73%	2,276%
	256	73%	2,857%		
	512	75%	2,325%		

Tabela 5.10: Resultados para testes T1 a T5 usando SMS (*First Samples*)

Como pode ser visto na tabela 5.10, a variação no número de amostras iniciais tem certo impacto no erro das estimativas: aumentando-se este valor consegue-se diminuir o erro. Pelo fato do circuito ser instável nestes testes, quando se tem um número maior de amostras iniciais, é obtida uma média global inicial mais precisa, melhorando a resposta do algoritmo para estes testes.

Nos testes com a variação no limite máximo das amostras monitoradas e não monitoradas, cujos resultados estão na tabela 5.11, pode-se ver que para T1–T4 as diferentes configurações tiveram pouco impacto nos resultados, o tempo de amostragem se manteve alto e o erro foi pequeno e variou de forma irrelevante. Novamente, devido ao comportamento instável do circuito, o algoritmo não consegue aumentar o número de amostras não monitoradas durante a simulação e, por isso, o impacto foi tão pequeno nestes testes.

Para T5, diferentemente, a variação destes parâmetros tem maior impacto nos resultados. Para a configuração *(max_pred, max_mon) = (4096, 2048)*, por exemplo, o erro obtido foi alto (9.628%). Isto ocorre porque o circuito fica estável durante um grande intervalo de tempo na simulação e, também, como o número máximo de amostras não monitoradas nesta configuração é alto (4096), há uma forte tendência do algoritmo crescer a janela não monitorada até este valor. Então, quando o comportamento do circuito é subitamente alterado, a simulação continua não monitorada por um tempo excessivo, retardando a correção da média global de transição de atividade do circuito.

Experimento	α	Samples Range		Default	
		T.A.	Erro	T.A.	Erro
T1	512/512	100%	0,057%	99%	0,048%
	128/256	99%	0,054%		
	64/64	98%	0,049%		
	4096/2048	100%	0,055%		
	2048/4096	100%	0,055%		
T2	512/512	100%	0,007%	99%	0,013%
	128/256	99%	0,007%		
	64/64	98%	0,043%		
	4096/2048	100%	0,012%		
	2048/4096	100%	0,006%		
T3	512/512	100%	0,013%	99%	0,009%
	128/256	99%	0,003%		
	64/64	98%	0,016%		
	4096/2048	100%	0,008%		
	2048/4096	100%	0,002%		
T4	512/512	100%	0,089%	99%	0,124%
	128/256	99%	0,113%		
	64/64	98%	0,127%		
	4096/2048	100%	0,114%		
	2048/4096	100%	0,114%		
T5	512/512	71%	3,886%	73%	2,276%
	128/256	74%	1,171%		
	64/64	74%	0,771%		
	4096/2048	66%	9,628%		
	2048/4096	66%	9,625%		

Tabela 5.11: Resultados para testes T1 a T5 usando SMS (*Samples Range*)

Na configuração *(max_pred, max_mon) = (64, 64)* o erro obtido consegue ser reduzido de 9.628%, da configuração anterior, para 0.771%. Isto acontece porque, como o limite máximo do tamanho da janela de não monitoração é pequena, consegue-se detectar a queda brusca na atividade de transição antes e, desta forma, a correção é feita de forma mais rápida.

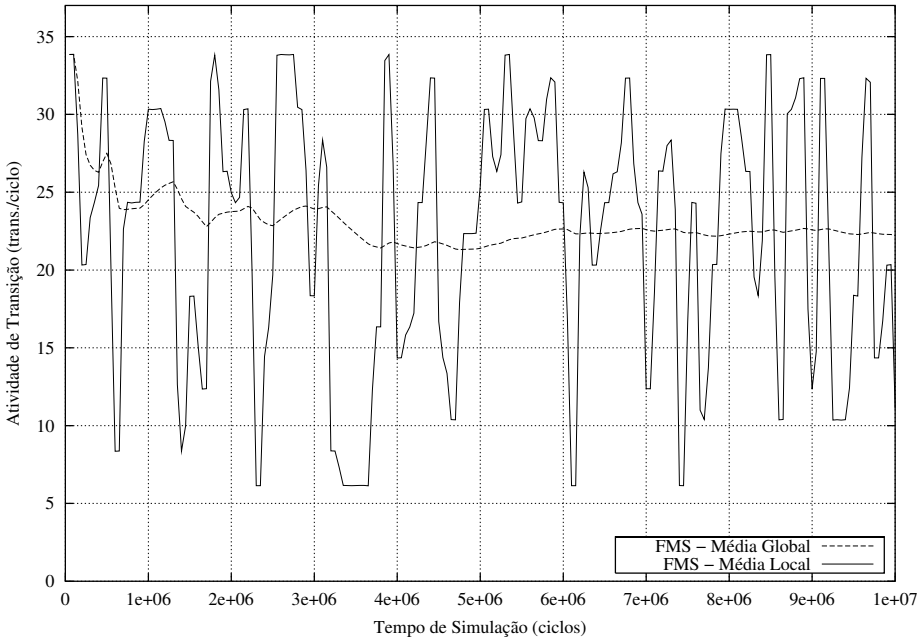
Na tabela 5.12 são mostrados os resultados obtidos para os testes T1–T5 referentes a diversas configurações do parâmetro *threshold*. Como pode ser visto pelos dados, mesmo relaxando o *threshold* para 5% (0.05) não se consegue ter uma diminuição muito significativa do tempo de amostragem e, como esperado, o erro das estimativas aumenta, embora para a maior parte dos casos este erro continue sendo muito pequeno.

Experimento	α	Threshold		Default	
		T.A.	Erro	T.A.	Erro
T1	0,0005	99%	0,003%	99%	0,048%
	0,01	99%	0,053%		
	0,02	98%	0,164%		
	0,05	95%	0,532%		
T2	0,0005	99%	0,005%	99%	0,013%
	0,01	99%	0,021%		
	0,02	99%	0,019%		
	0,05	96%	0,160%		
T3	0,0005	99%	0,004%	99%	0,009%
	0,01	99%	0,011%		
	0,02	99%	0,011%		
	0,05	88%	0,780%		
T4	0,0005	99%	0,004%	99%	0,124%
	0,01	99%	0,110%		
	0,02	99%	0,110%		
	0,05	99%	0,132%		
T5	0,0005	98%	0,017%	73%	2,276%
	0,01	72%	2,468%		
	0,02	71%	3,389%		
	0,05	71%	3,389%		

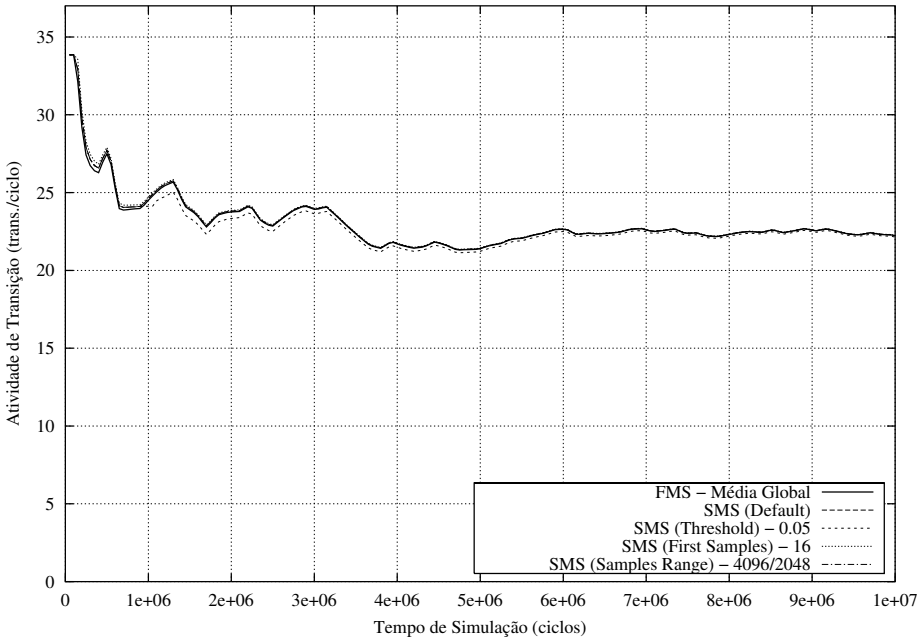
Tabela 5.12: Resultados para testes T1 a T5 usando SMS (*Threshold*)

As figuras 5.15(b)–5.19(b) permitem a visualização de algumas estimativas para os testes de T1 até T5 usando o algoritmo SMS, com algumas variações na configuração dos parâmetros. Note como, durante toda a simulação dos circuitos, as estimativas se mantém sempre muito próximas do resultado obtido utilizando o algoritmo FMS.

Neste capítulo foram apresentados os resultados experimentais da biblioteca PowerSC, primeiramente validando os resultados obtidos, comparando-os com os resultados de uma ferramenta comercial. Após isso, para a técnica de simulação monitoração por amostragem, ou SMS, dois módulos foram utilizados para mostrar sua eficiência: o primeiro é um módulo real, parte de um decodificador de MP3 que, juntamente a um conjunto de entradas reais, demonstrou o potencial do algoritmo proposto. O segundo foi um módulo utilizado para demonstrar que o algoritmo se comporta de forma adequada, mesmo em circuitos altamente instáveis, fornecendo sempre ao usuário uma estimativa confiável.

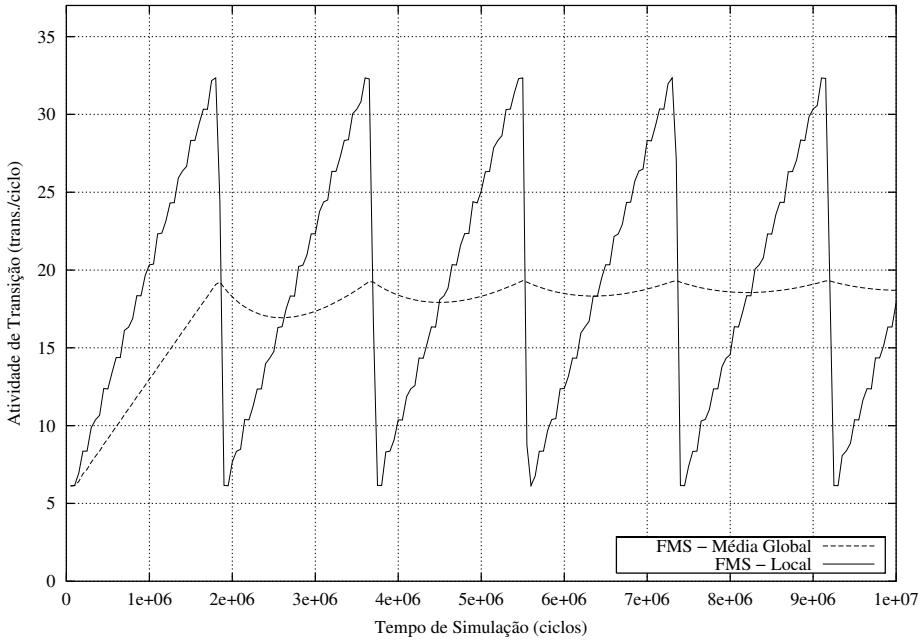


(a) Usando FMS, mostrando as médias global e local

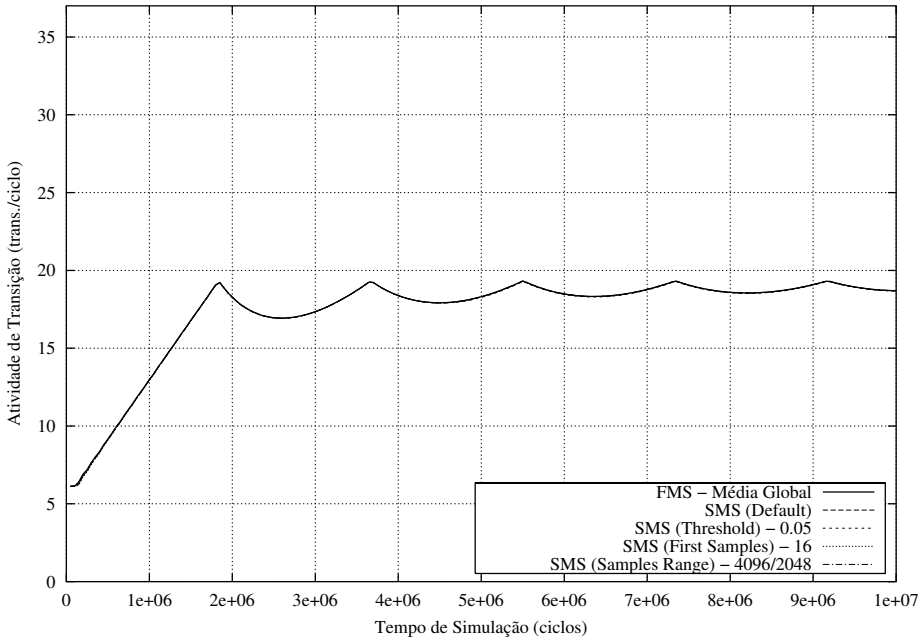


(b) Usando SMS, mostrando algumas configurações de parâmetros

Figura 5.15: Atividade de transição para T1 durante a simulação

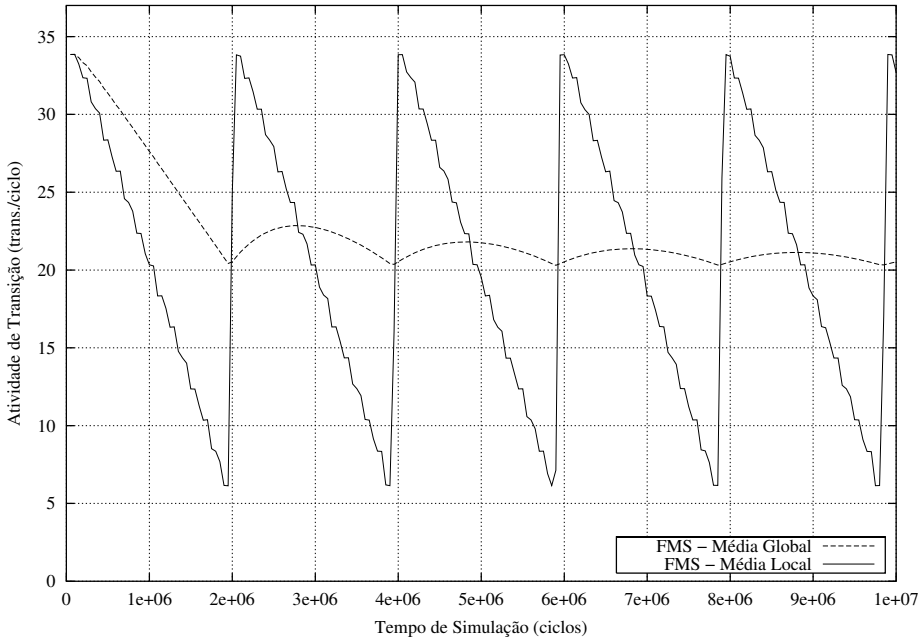


(a) Usando FMS, mostrando as médias global e local

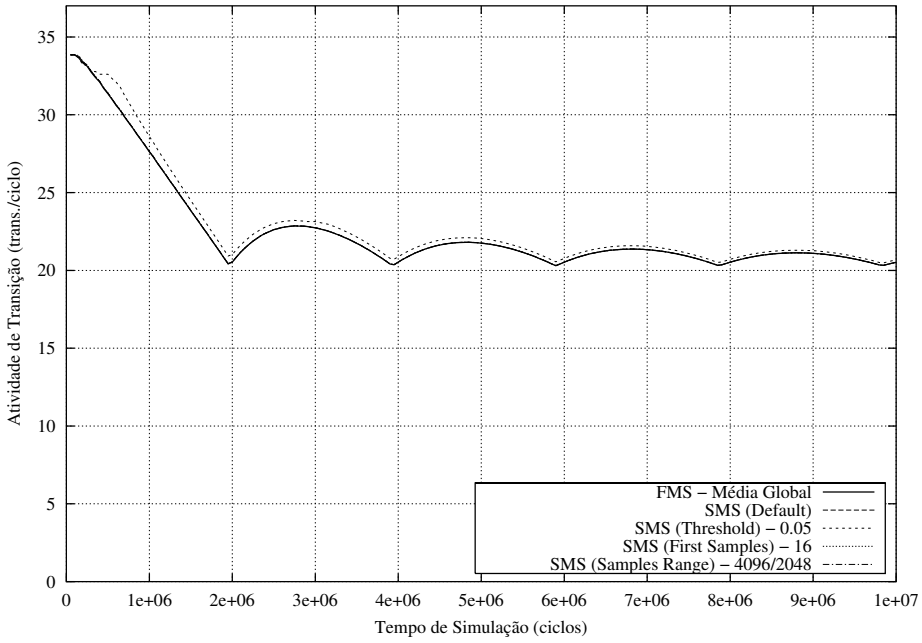


(b) Usando SMS, mostrando algumas configurações de parâmetros

Figura 5.16: Atividade de transição para T2 durante a simulação

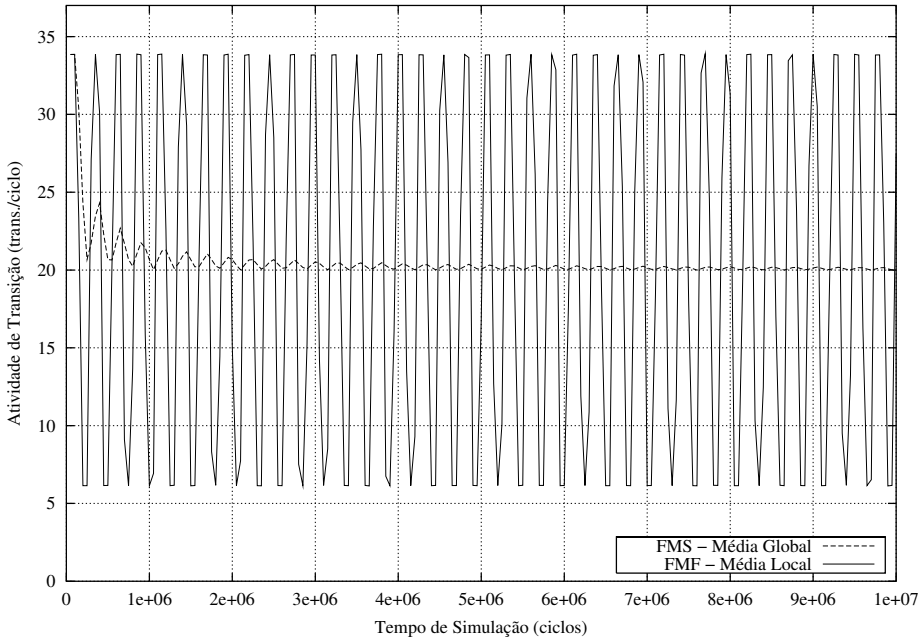


(a) Usando FMS, mostrando as médias global e local

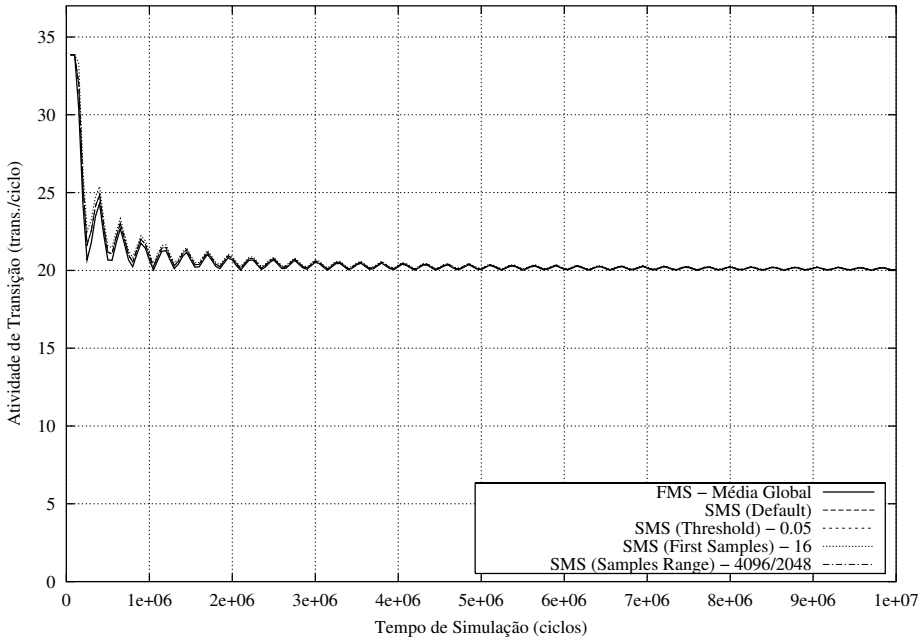


(b) Usando SMS, mostrando algumas configurações de parâmetros

Figura 5.17: Atividade de transição para T3 durante a simulação

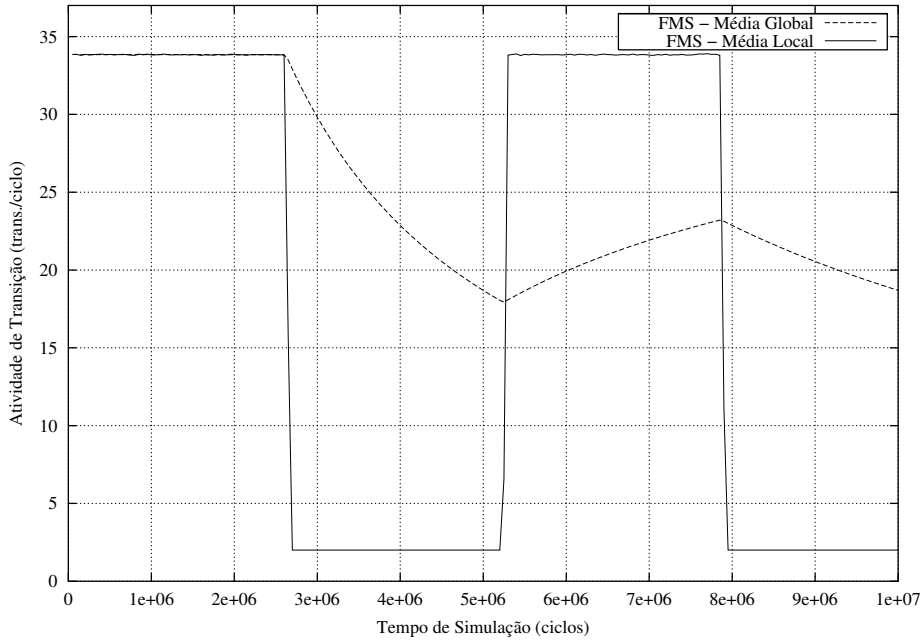


(a) Usando FMS, mostrando as médias global e local

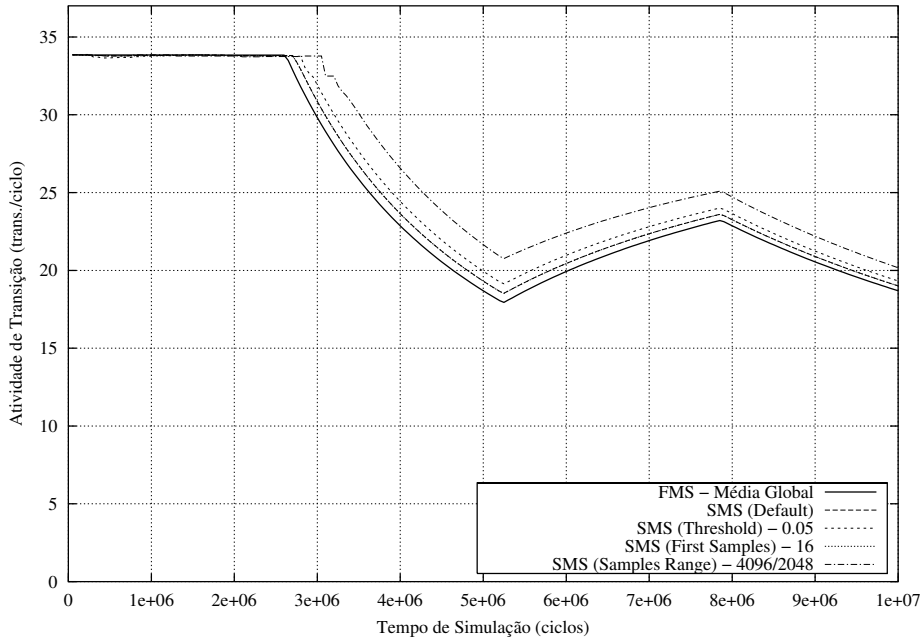


(b) Usando SMS, mostrando algumas configurações de parâmetros

Figura 5.18: Atividade de transição para T4 durante a simulação



(a) Usando FMS, mostrando as médias global e local



(b) Usando SMS, mostrando algumas configurações de parâmetros

Figura 5.19: Atividade de transição para T5 durante a simulação

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho apresentou a PowerSC, uma biblioteca que estende as capacidades da linguagem SystemC, de forma a habilitar a captura da atividade de transição de modelos escritos nesta linguagem, incrementando a especificação executável gerada pelo SystemC.

Juntamente com a biblioteca PowerSC, uma metodologia alternativa à de uma ferramenta comercial foi proposta, reduzindo o número de passos necessários para seu uso. A metodologia tem como *apparatus* do fluxo de projeto apenas as bibliotecas SystemC e PowerSC, com um compilador padrão de C++. Ela é simples e transparente ao projetista, necessitando um esforço mínimo para seu uso. A idéia utilizada para a PowerSC só é possível graças ao poder de expressão de uma linguagem de alto nível como C++ e não seria possível com linguagens de descrição de *hardware* mais tradicionais, como Verilog e VHDL.

Outra contribuição deste trabalho foi o algoritmo SMS, apresentado no capítulo 4, o qual é capaz de detectar, eficientemente, períodos de estabilidade dos circuitos, conseguindo reduzir, de forma expressiva, o tempo necessário de monitoração, como também é capaz de detectar circuitos com comportamento instável ("mal-comportados"), mantendo sempre o compromisso de fornecer ao usuário uma estimativa precisa da atividade de transição.

A informação de atividade de transição capturada utilizando a PowerSC pode facilmente ser utilizada para alimentar os modelos das técnicas de estimativa do nível arquitetural, apresentados no capítulo 3.

Este trabalho resultou na submissão de um artigo para a conferência SOC'05 (*IEEE International Symposium on System-on-Chip 2005*), a ser realizado na Finlândia, em novembro. O artigo foi aceito para publicação, intitulado *High-Level Switching Activity Prediction Through Sampled Monitored Simulation* [39], e estará disponível nos anais desta conferência. Planeja-se a submissão de outros resultados deste trabalho em alguma revista científica da área, ainda em fase de seleção.

As contribuições deste trabalho podem ser resumidas abaixo:

- Desenvolvimento de uma biblioteca que estende uma linguagem de descrição de sistemas, dando suporte à captura de atividade de transição (*switching activity*) a partir de modelos numa descrição de alto nível, em código C++;
- Proposição de uma metodologia de análise de potência alternativa à de uma ferramenta comercial, para ser utilizada com a extensão desenvolvida. Esta metodologia, diferentemente da metodologia da ferramenta comercial, é simples e transparente, exigindo um esforço mínimo do projetista para seu uso;
- Superação de uma limitação da ferramenta comercial, incapaz de monitorar a atividade de transição de bancos de registradores;
- Proposta de um algoritmo de monitoração eficiente, capaz de reduzir o tempo necessário de monitoração durante a simulação do modelo de forma expressiva, com uma perda mínima da precisão.

6.1 Trabalhos Futuros

Ao final deste trabalho sobram algumas lacunas importantes que devem ser preenchidas, como por exemplo, a ausência de informação da tecnologia-alvo na qual o modelo será implementado. Essa informação é necessária para que a geração de estimativas de consumo de potência tornem-se mais precisas, fazendo possível o fornecimento de valores em Watts ao usuário.

No entanto, esta é uma tarefa factível, dado que não seria difícil conectar uma biblioteca de tecnologia à nossa abordagem. Esta possível melhoria em nossa metodologia, que fica como trabalho futuro, é mostrada na figura 6.1. Note as diferenças com relação a figura 4.15 (pág. 40). Esta nova versão da metodologia da PowerSC incluiria agora informações de tecnologia, num formato que poderia ser semelhante ao das bibliotecas de tecnologia caracterizadas para potência da Synopsys [7]. Estas bibliotecas têm informações de potência estática (*leakage power*) e potência dinâmica para células individuais, que nada mais são que blocos RTL definidos na biblioteca.

Assim, utilizando-se da informação contida nesta biblioteca de tecnologia, um *parser* receberia como entrada o modelo SystemC juntamente com a biblioteca de tecnologia apropriada *(i)*, gerando como saída um arquivo, por exemplo, chamado *modelo.psclib (ii)*. Este processo de *parsing* do passo *(iii)* mapearia cada um dos elementos do modelo para as células da biblioteca de tecnologia, anotando as informações para os nodos do modelo. Após isso, este arquivo poderia ser usado como entrada para a simulação RTL, através de uma macro, na seguinte forma:

```
PSC_USE_TECH_LIB("modelo.psclib");
   ...
PSC_REPORT_SWITCHING_ACTIVITY;
```

Isso indicaria à PowerSC para extrair do arquivo `modelo.psclib` as informações específicas de tecnologia para os elementos monitorados. Os relatórios contendo as estimativas *(iv)* poderiam então fornecer ao usuário números mais precisos.

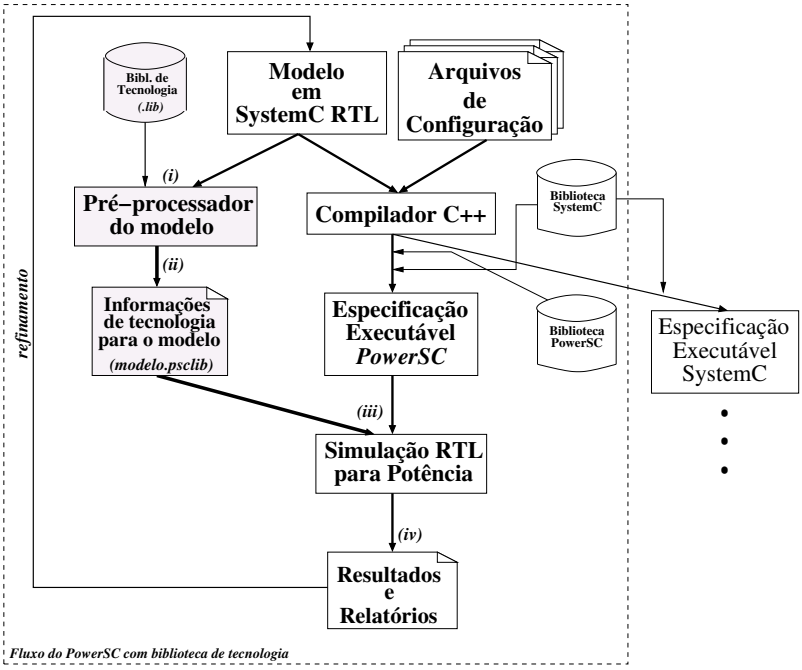


Figura 6.1: PowerSC com informações de tecnologia

Outra melhoria proposta refere-se ao algoritmo de simulação com monitoração por amostragem, ou SMS:

- **Parâmetros auto-ajustáveis**: pretende-se explorar algumas extensões do algoritmo que permitam ajustar automaticamente seus parâmetros durante a simulação, de forma a tirar mais proveito dos períodos de estabilidade e, também, exigir menos intervenção do usuário.

- **Critério de parada**: outra extensão interessante a ser explorada seria criar um critério de parada, utilizando para tal fim, técnicas da teoria de probabilidade, para decidir quando os elementos monitorados de um circuito tenham convergido, reduzindo o número de amostras necessárias para a obtenção das estimativas.

Referências Bibliográficas

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 19 April 1965.

[2] Eedesign online. http://www.eedesign.com. 2005.

[3] S. H Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor consumption. *Intel Technology Journal*, 2001.

[4] L. W. Nagel. Spice2: A computer program to simulate semiconductor circuits. Erlm520, Univ. California, Berkeley, 1975.

[5] Jan Rabaey. Low-power system design. In *EUROCHIP Course on Methods and Tools for Digital System Design*, 1995. Leuven, Netherlands.

[6] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits*. Prentice Hall, 2nd edition, 2003.

[7] Synopsys Inc. *PowerCompiler User Guide*, v-2003.12 edition, December 2003.

[8] Itrs 2003 edition. Technical report, International Technology Roadmap for Semiconductors, 2003.

[9] Charlie X. Huang, Bill Zhang, An-Chang Deng, and Burkhard Swirski. The design and implementation of powermill. In *Proceedings of the 1995 international symposium on Low power design*, pages 105–110. ACM Press, 1995.

[10] Paul Landman. High-level power estimation. In *ISLPED '96: Proceedings of the 1996 international symposium on Low power electronics and design*, pages 29–35. IEEE Press, 1996.

[11] K. D. Müller-Glaser, K. Hirsch, and K. Neusinger. Estimating essential design characteristics to support project planning for ASIC design management. In Louise Goto, Satoshi; Trevillyan, editor, *Proceedings of the IEEE International Conference*

on Computer-Aided Design, pages 148–151, Santa Clara, CA, November 1991. IEEE Computer Society Press.

[12] D. Liu and C. Svensson. Power consumption estimation in CMOS VLSI chips. *IEEE Journal of Solid-State Circuits*, pages 663–670, June 1994.

[13] Mahadevamurty Farid and N. Najm. Towards a high-level power estimation capability. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 588–598. IEEE Press, 1996.

[14] Diana Marculescu, Radu Marculescu, and Massoud Pedram. Information theoretic measures of energy consumption at register transfer level. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 81–86. ACM Press, 1995.

[15] Kwang-Ting Cheng and Vishwani D. Agrawal. An entropy measure for the complexity of multi-output boolean functions. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 302–305. ACM Press, 1990.

[16] Scott R. Powell and Paul M. Chau. Estimating power dissipation of vlsi signal processing chips: The pfa technique. *VLSI Signal Processing IV*, pages 250–259, 1990.

[17] Paul E. Landman and Jan M. Rabaey. Activity-sensitive architectural power analysis. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, pages 571–587. IEEE Computer Society Press, June 1996.

[18] Subodh Gupta and Farid N. Najm. Energy and peak-current per-cycle estimation at rtl. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(4):525–537, 2003.

[19] Anand Raghunathan, Sujit Dey, and Niraj K. Jha. Register-transfer level estimation techniques for switching activity and power consumption. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 158–165. IEEE Computer Society, 1996.

[20] Manuela Anton, Ionel Colonescu, Enrico Macii, and Massimo Poncino. Fast characterization of rtl power macromodels. In *IEEE Proceedings of ICECS 2001*, pages 1591–1594, 2001.

[21] Subodh Gupta and Farid N. Najm. Energy-per-cycle estimation at rtl. In *Proceedings of the 1999 international symposium on Low power electronics and design*, pages 121–126. ACM Press, 1999.

[22] Huzefa Mehta, Robert Michael Owens, and Mary Jane Irwin. Energy characterization based on clustering. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 702–707. ACM Press, 1996.

[23] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 340–345. ACM Press, 2000.

[24] Michael Eiermann and Walter Stechele. Novel modeling techniques for rtl power estimation. In *Proceedings of the 2002 international symposium on Low power electronics and design*, pages 323–328. ACM Press, 2002.

[25] A. Chandrakasan, M. Potkonjak, J. Rabaey, and R. Brodersen. Optimizing power using transformations. In *IEEE Transactions on Computer-Aided Design*, pages 12–31, 1995.

[26] Raul San Martin and John P. Knight. Power-profiler: optimizing asics power consumption at the behavioral level. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 42–47. ACM Press, 1995.

[27] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 384–390. IEEE Computer Society Press, 1994.

[28] Praveen Kalla, Jörg Henkel, and Xiaobo Sharon Hu. Sea: Fast power estimation for micro-architectures. In *ASP-DAC '03: Proceedings of the 2003 conference on Asia South Pacific design automation*, pages 600–605, 2003.

[29] David Lidsky and Jan M. Rabaey. Early power exploration: a world wide web application. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 27–32. ACM Press, 1996.

[30] Reinaldo A. Bergamaschi and Yunjian W. Jiang. State-based power analysis for systems-on-chip. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 638–641. ACM Press, 2003.

[31] Xilinx Inc. *XPower Tutorial – FPGA Design*, v1.2 edition, January 2002.

[32] Sequence Design Inc. See http://www.sequencedesign.com. 2005.

[33] Mentor Graphics Corporation. *ModelSim SE 5.8b User's Manual*, January 2004.

[34] Open SystemC Initiative. *SystemC Language Reference Manual*, revision 1.0 edition, 2003. See http://www.systemc.org.

[35] Open SystemC Initiative. *SystemC 2.0 User's Guide*, version 2.0 edition, 2002. See http://www.systemc.org.

[36] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[37] Free Software Foundation. *Using the GNU Compiler Collection*, May 2004. See http://gcc.gnu.org. 2005.

[38] Synopsys solvnet. http://synopsys.solvnet.com. 2005.

[39] F. Klein, R. Azevedo, and G. Araujo. High-level switching activity prediction through sampled monitored simulation. In *Proc. of International Symposium on System-on-Chip (SOC)*, Tampere, Finland, November 2005. Accepted for publication.