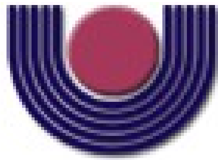

UNIOESTE – Universidade Estadual do Oeste do Paraná

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Informática

Curso de Bacharelado em Informática

VIPRO-MP: uma plataforma virtual multiprocessada baseada na arquitetura SimpleScalar

Maxiwell Salvador Garcia

CASCAVEL

2008

Maxiwell Salvador Garcia

VIPRO-MP: uma plataforma virtual multiprocessada baseada na arquitetura SimpleScalar

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Informática, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Dr. Marcio Seiji Oyamada

CASCAVEL

2008

Maxiwell Salvador Garcia

VIPRO-MP: uma plataforma virtual multiprocessada baseada na arquitetura SimpleScalar

Monografia apresentada como requisito parcial para obtenção do Título de *Bacharel em Informática*, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

____

Dr. Marcio Seiji Oyamada (Orientador)

Colegiado de Informática, UNIOESTE

____

Msc. Guilherme Galante

Colegiado de Informática, UNIOESTE

____

Msc. Anibal Mantovani Diniz

Colegiado de Informática, UNIOESTE

Cascavel, 28 de Julho de 2008.

DEDICATÓRIA

Este trabalho é dedicado à minha família, que sempre me apoiou e me incentivou nesta jornada, agora cumprida, e à todos meus amigos que me deram forças para que eu não desistisse, mudando algumas decisões que havia tomado.

EPÍGRAFE

> "Toda a nossa ciência, comparada com a realidade, é primitiva e infantil – e, no entanto, é a coisa mais preciosa que temos."
>
> Albert Einstein

AGRADECIMENTOS

Apesar do homem ser um ser sociável, podendo cruzar e interagir com milhões de pessoas, o sentimento de solidão é facilmente experimentado, pela complexa natureza do ser e estar. Por isso, devo muito às pessoas que entraram em minha vida e permitiram que tal sentimento fosse menos presente, principalmente nestes últimos tempos de independência psicológica que vivi.

Primeiramente, deixo meu profundo agradecimento aos meus pais, que me ofereceram condições ideais para concluir este curso, apoiando-me sempre nas decisões. Mesmo distantes, os senti mais presentes que antes fora, sendo meu guia e incentivo para continuar a caminhada. Agradeço, também, a Claudinha, "minha amada imortal", que deixou esta estrada menos árdua e mais tranqüila e entendeu meus períodos de ausência por conta de obrigações da faculdade e da música.

Quero agradecer ao meu orientador e amigo Márcio, que participou diretamente na construção desde trabalho, e "se enxerguei mais longe, foi porque estava sobre os ombros de gigantes". Aos meus amigos, que, neste período de festas e estudos, fizeram parte de minha vida, ajudando-me tanto em caráter pessoal, quanto acadêmico.

Enfim, agradeço a todos que contribuíram para meu crescimento neste período de graduação, e o menino que entrou, hoje se despede como homem.

Maxiwell Salvador Garcia

Lista de Figuras

Figura 2.1: Retorno financeiro para o lançamento de um produto com e sem atraso [55]......................6
Figura 2.2: Impacto de novas tecnologias no custo do projeto [22]...7
Figura 2.3: Metodologia de projeto de ES [55]...8
Figura 2.4: Níveis macros de abstração na especificação de projeto [56]..8
Figura 2.5: Fluxo tradicional de integração de hardware e software [50]..10
Figura 3.1: Impacto no consumo de potência conforme a tecnologia [36]...13
Figura 3.2: Paralelismo de *threads*..14
Figura 3.3: Modelo UMA...16
Figura 3.4: Organização da máquina Carnegie-Mellon Cm* [47]...17
Figura 3.5: Freqüência *vs* Voltagem em um StrongARM SA-1100 [45]...19
Figura 3.6: Consumo de potência em um sistema com ARM9 [57]..20
Figura 4.1: Relacionamento das ferramentas utilizadas...24
Figura 4.2: Diagrama da implementação do sim-outorder [5]..25
Figura 4.3: Arquitetura do funcionamento do Sim-Wattch [4]..28
Figura 4.4: Duas metodologias de projeto. (a) metodologia tradicional; (b) metodologia SystemC.....29
Figura 5.1: Sim-Wattch com a biblioteca CACTI 1.0 integrada...32
Figura 5.2: Diagrama de estado para acessar à memória compartilhada...37
Figura 5.3: Organização do VIPRO-MP com dois processadores e timer...40
Figura 6.1: Algoritmo de compressão JPEG paralelizado...43
Figura 6.2: Gráfico com a variação dos tamanhos das caches. (a) versus os ciclos de clock; (b) versus a Potência Dissipada..45
Figura 6.3: Gráfico com variação de IL1 e DL1 fixada em 2x128Kb...45

Lista de Tabelas

Tabela 6.1: Impacto da latência da memória compartilhada no número de ciclos................................42
Tabela 6.2: Dados de configuração e execução de ambientes com um, dois e quatro núcleos.............46

Lista de Quadros

Quadro 5.1: Criação do *clock* e sua ligação com os componentes..35
Quadro 5.2: Pseudocódigo do laço principal do processador..35
Quadro 5.3: Instância de dois processadores..36
Quadro 5.4: Instância da memória compartilhada..37
Quadro 5.5: Variáveis sobre a contenção do barramento de um processador.......................................38
Quadro 5.6: Tratamento da interrupção no processador...39
Quadro 5.7: Salvando o contexto e saltando para a função implementável..39

Lista de Abreviaturas e Siglas

API	Application Programming Interface
CC-NUMA	Cache Coherent - Non-Uniform Memory Access
CMOS	Complementary metal-oxide-semiconductor
COMA	Cache-Only Memory Access
CRC	Classes, Responsabilidades e Colaboradores
DCT	Discrete Cosine Transform
DVD	Digital Versatile Disc
ES	Embedded Systems
IDE	Integrated Development Environment
MMU	Memory Management Unit
MPSoC	Multiprocessor Systems-on-Chip
NC-NUMA	Non Coheren - Non-Uniform Memory Access
NUMA	Non- Uniform Memory Access
PDA	Personal Digital Assistant
PISA	Portable Instruction Set Architecture
PPE	PowerPC Processor Element
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SMP	Symmetric MultiProcessor
SMT	Simultaneous Multithreading
SoC	Systems-on-Chip
SPE	Synergistic Processor Element
TLM	Transaction-Level Messages
UMA	Uniform Memory Access
UML	Unified Modeling Language
VIPRO-MP	Virtual Prototype for Multiprocessor Architectures

Sumário

Lista de Figuras vii

Lista de Tabelas viii

Lista de Quadros ix

Lista de Abreviaturas e Siglas x

Sumário xi

Resumo xiii

1 Introdução 1

1.1 Objetivos gerais....3

1.2 Objetivos específicos....3

1.3 Organização do texto....4

2 Sistemas Embarcados 5

2.1 Projeto de Sistemas Embarcados....7

2.1.1 Metodologia de projeto....7

2.1.2 Níveis de abstração....8

2.1.3 Particionamento entre *software* e *hardware*....9

2.2 Simulação de arquitetura....10

3 Processamento Paralelo 12

3.1 Níveis de Paralelismo....13

3.2 Arquiteturas multiprocessadas....15

3.2.1 Modelos de memória compartilhada....15

3.2.1.1 Modelo UMA....16

3.2.1.2 Modelo NUMA....16

3.2.1.3 Modelo COMA....18

3.2.1.4 Modelo VIPRO-MP....18

3.2.2 Medidas de desempenho e consumo de potência....19

3.3 Software para Sistemas Paralelos....21

3.3.1 O futuro da programação *multicore*....23

4 Ferramentas Utilizadas **24**
4.1 A ferramenta SimpleScalar........25
4.2 A biblioteca CACTI........26
4.3 A ferramenta Sim-Wattch........27
4.4 A biblioteca SystemC........28
4.4.1 Metodologias de projetos........29
5 A ferramenta VIPRO-MP **31**
5.1 Desenvolvimento........32
5.1.1 Metodologia........33
5.1.2 Atualização da biblioteca CACTI........34
5.1.3 A modelagem SystemC........34
5.1.4 Barramento e memória compartilhada........36
5.1.5 Interrupção........38
5.2 Organização final da ferramenta........40
6 Simulações e Resultados **41**
6.1 Estudo de Caso – Ordenação paralela........41
6.1.1 Resultados........42
6.2 Estudo de Caso - Algoritmo JPEG........43
6.2.1 Exploração do espaço de projeto nas caches........43
6.2.2 Resultados........44
6.2.3 Exploração do espaço de projeto com múltiplos processadores........46
6.2.4 Resultados........46
7 Conclusões e Trabalhos Futuros **48**
A Arquivo de Configuração do *sim-outorder* **50**
B Arquivo Resultado do Sim-Wattch **53**
C Utilizando o VIPRO-MP **60**
Referências Bibliográficas **63**

Resumo

O aumento crescente da utilização de *software* embarcado, com requisitos estritos em termos de desempenho, consumo de potência, tempo de projeto, entre outros, requerem plataformas de *hardware* eficientes. Por esse motivo, a arquitetura multiprocessada está sendo adotada, tanto em sistemas embarcados quanto em computação de propósito geral. Um problema possível é que esperar a construção completa de um *hardware* para, enfim, poder testar o *software* embarcado não é viável, posto que atrasos podem ocorrer em ambas as fases, prejudicando o projeto como um todo. Outro agravante é a possibilidade do *hardware* construído não ser eficiente o bastante para acomodar o *software* em construção. Visando sanar esses problemas, plataformas virtuais estão sendo propostas, dando ao projetista a possibilidade de testar várias configurações de componentes, simular e analisar o desempenho e o consumo de potência obtido antes da construção do *hardware*. O objetivo deste trabalho é construir uma plataforma virtual para arquiteturas multiprocessadas, focando-se na arquitetura PISA, utilizando o SimpleScalar para simular processadores *superescalares*. A biblioteca SystemC foi utilizada pela possibilidade de modelar componentes de *hardware* e *software* em C++, agilizando o processo de construção do *hardware* final. A comunicação entre os processadores foi implementada utilizando uma memória compartilhada, que é acessada por meio de um barramento modelado explicitamente. Para a validação da plataforma virtual proposta neste trabalho, dois estudos de caso, ordenação paralela e compressão JPEG, foram realizados, variando-se os mais diferentes aspectos da plataforma como tamanho de *cache* e número de processadores, a fim de comprovar a eficácia da plataforma construída, analisando o desempenho e o consumo de potência das arquiteturas.

Palavras-chave: Sistemas embarcados, Arquitetura multiprocessada, plataforma virtual, SystemC, MIPS, PISA, SimpleScalar, Sim-Wattch.

Capítulo 1
Introdução

A convergência de produtos no mercado de consumo, tais como celulares e PDA com capacidade multimídia, TV digital interativa, automóveis com módulos digitais complexos (navegação e rastreamento), requerem arquiteturas com grande capacidade de processamento e habilitada para suportar diferentes domínios de aplicação. Por outro lado, o aumento de dispositivos alimentados por baterias necessita de um projeto otimizado, com baixo consumo de potência [55].

Atualmente, o projeto de um sistema embarcado utiliza cada vez mais de *software* embarcado, propiciando flexibilidade e extensibilidade para o projeto. Um outro aspecto importante, é o aumento de soluções SoC (System-on-chip) contendo mais de um processador sendo denominadas MPSoC (Multiprocessor System-on-chip) [23]. Considerando o desenvolvimento de tais sistemas, pode ser visto que, devido à grande variabilidade de aplicações, são necessários projetos customizados, tornando evidente a necessidade de modelos de simulação com rápida adaptabilidade [1].

Portanto, torna-se fundamental para o aumento da capacidade de exploração de alternativas de projeto, nesse tipo de sistemas, a simulação não só de um processador executando um dado *software*, mas também de outros componentes envolvidos, como memórias, barramentos, sistemas de interconexão e outros componentes de aplicação específica. Isto proporcionará a um projetista experimentar diversas alternativas de arquitetura de um sistema como um todo, adaptando-o de acordo com o *software* embarcado. Esta coleção de componentes previamente desenvolvida pode ser reutilizada para a integração em um novo sistema embarcado a cada momento [1].

Tais ferramentas também auxiliam e aceleram projetos cujo os requisitos de tempo precisam ser satisfeitos, reduzindo o tempo de prototipação e iniciando o desenvolvimento do *software* embarcado o quanto antes, mesmo na ausência do *hardware* final. Essa antecipação

da implementação é necessária devido à crescente complexidade dos *software*s e, conforme a tecnologia (130nm, por exemplo), o esforço de desenvolvimento do *software* ultrapassa o desenvolvimento da arquitetura [39]. Com isso, questões críticas relacionadas com *time-to-market,* como os consoles de vídeo-game que são escalonados de forma que o lançamento seja realizado antes do Natal, podem ser amenizadas e resolvidas.

A simulação é a técnica normalmente utilizada para validar um projeto antes da sua implementação. Para avaliar arquiteturas de processadores superscalares, Todd Austin com auxílio de colaboradores desenvolveu o SimpleScalar [44][5] enquanto estudante de Ph. D. da Universidade de Wisconsin. Uma empresa foi criada a fim de continuar o projeto, oferecer suporte e licenciar comercialmente o produto. Não obstante, ele possui o código aberto e é *freeware* para acadêmicos quando estes o utilizam sem fins comerciais. Um grupo da Universidade de Princeton, conduzido pela professora Margaret Martonosi, ampliou o SimpleScalar ao incorporar um analisador de consumo de potência, o chamando de Sim-Wattch, sendo possível fazer estatísticas sobre o consumo de potência de uma dada organização do processador [4].

Após verificar a confiabilidade desses projetos, realizando simulações precisas, flexíveis e rápidas de arquiteturas superescalares, empresas e centros de pesquisas ao redor do mundo começaram a utilizá-los em larga escala para avaliação de desempenho [40]. Isso trouxe reduções de custos e mais agilidade aos processos, pois, com um simples comando, uma nova organização e arquitetura pode ser criada e testada, garantindo uma rápida verificação do desempenho do sistema

Porém uma limitação se encontra presente, pois a arquitetura simulada com o SimpleScalar (e conseqüentemente com o Sim-Wattch) é a monoprocessada. Existem projetos que tem como objetivo expandir o SimpleScalar para uma arquitetura multiprocessada. Porém, neste trabalho, além de suportar múltiplos processadores, a arquitetura poderá ser acrescentada com outros componentes de *hardware* (barramentos, memórias, sistemas de interconexão) em módulos individuais. A especificação de cada um dessas partes (*hardware* e *software*) utiliza ferramentas (linguagens de especificação e simulação) tão diferentes que torna o processo de integração uma tarefa delicada. Em vista disso, organizações voltadas para o desenvolvimento de sistemas embarcados disponibilizaram uma biblioteca baseada em C/C++ denominada SystemC [51]. Esta biblioteca tem como objetivo possibilitar ao programador a construção de blocos (módulos) de *hardware*, e simulá-los obedecendo a um relógio (*clock*). O uso de SystemC possibilita a implementação em diversos níveis de abstração, começando de uma aplicação escrita puramente em C++, sendo refinada até o nível RTL (*Register Transfer*

Level) [41]. Assim, com o uso de SystemC, é possível descrever tanto os componentes de *hardware* como os componentes de *software* de um sistema, permitindo ao seu usuário utilizar elementos típicos de *hardware*, tais como portas e sinais, dentro do contexto de C++.

1.1 Objetivos gerais

Construir uma plataforma virtual multiprocessadas com estimativa de desempenho e consumo de potência utilizando o simulador SimpleScalar para modelar os processadores superescalares. A arquitetura base é a PISA, baseada nos processadores MIPS. A comunicação entre os processadores é realizada por meio de uma memória compartilhada, onde um barramento conecta todos os componentes.

Objetiva-se, com isso, avaliar rapidamente o desempenho e consumo de potência sobre diferentes configurações de uma plataforma de hardware em relação a execução de uma determinada aplicação.

1.2 Objetivos específicos

A plataforma virtual multiprocessada a ser desenvolvida neste trabalho deve suportar componentes de *hardware*, e para isso, a linguagem SystemC foi incorporada após serem realizadas as devidas modificações no código do SimpleScalar original. Assim, cada componente da plataforma, processadores, barramento, memória compartilhada, *timer*, é descrito por um módulo SystemC, podendo ser configurado conforme a necessidade do projetista. Por meio da biblioteca SystemC, os componentes foram sincronizados por um *clock*, possibilitando a execução ciclo a ciclo de cada componente da plataforma. O uso do SystemC possibilita também o reuso de componentes já desenvolvidos nesta linguagem, criando assim plataformas com componentes customizados.

Um *timer* foi modelado, e após um intenso estudo do código do SimpleScalar, as interrupções foram implementadas, permitindo que modelos arquiteturais mais complexos possam ser testados futuramente.

1.3 Organização do texto

O trabalho inicia-se apresentando, no Capítulo 2, conceitos e características de Sistemas Embarcados e sua importância na atualidade. Como tais sistemas estão convergindo à soluções com múltiplos processadores em um *chip*, o Capítulo 3 faz uma revisão sobre Processamento Paralelo, abordando o impacto que arquiteturas multiprocessadas traz no consumo de potência, a organização da memória, e a construção de *software*s para ambientes paralelos. O Capítulo 4 detalha cada ferramenta utilizada para a construção do ambiente proposto, sendo elas o SimpleScalar, o Sim-Wattch e as bibliotecas CACTI e SystemC. O Capitulo 5 apresenta a plataforma virtual desenvolvida denominada VIPRO-MP (*Virtual Prototypes for Multiprocessor Architectures*) e suas etapas desenvolvimento. Os dois estudos de caso, ordenação paralela e compressão JPEG, e os resultados são descritos no Capítulo 6 e a conclusão e trabalhos futuros estão expostos no Capitulo 7. O Apêndice A e B são os arquivos de configurações e resultados da ferramenta Sim-Wattch, respectivamente. O Apêndice C apresenta uma descrição de uso do VIPRO-MP, com alguns trechos de código úteis ao entendimento.

Capítulo 2
Sistemas Embarcados

A presença de pequenos dispositivos munidos de processamento lógico está aumentando rapidamente, e, segundo pesquisas [11], em 2010, somarão 16 bilhões (três dispositivos por indivíduo). Nestes, microprocessadores, memórias, barramentos, e outros componentes de *hardware* e *software*, são encapsulados em um único circuito, que recebe o nome de Sistema Embarcado (*Embedded System* – ES). A importância destes sistemas no século XXI, consumindo 95% de todos os processadores produzidos atualmente [11], e suas particularidades de desenvolvimento, fez surgir uma nova área na computação, com milhares de pesquisadores [56].

O termo Sistema Embarcado, inicialmente, era associado a um sistema especialista, dedicado a uma tarefa específica e bem definida, diferentemente dos computadores ditos "convencionais" ou de propósito geral. Porém, as constantes evoluções arquiteturais permitiram a construção de sistemas embarcados robustos, com *software*s complexos, possibilitando a execução de várias tarefas e, inclusive, de *software*s feitos por usuários. Os celulares contemporâneos (*Iphone* da *Apple*, N96 da *Nokia*) são exemplos destes ES.

O sucesso de um projeto de sistema embarcado está diretamente relacionado à definição de seus requisitos para com os objetivos esperados. Dentre os requisitos importantes estão:

i. o desempenho: que visa uma alta capacidade de processamento;
ii. o consumo de energia: que prioriza o tempo de utilização do dispositivo;
iii. o custo de produção: no qual um valor limite é imposto para o produto final;
iv. tamanho e forma: quando a escolha do usuário for, também, baseado em estética;
v. o tempo de projeto: que estabelece limites para o desenvolvimento, muitas vezes, por fatores sazonais;
vi. tolerância à falhas e confiabilidade: quando sistemas críticos e de tempo real são projetados.

O projetista deve estar ciente dos conflitos existentes nas escolhas desses requisitos e um equilíbrio ideal para sua aplicação deve ser atingido. Assim, optar por um sistema embarcado com grande capacidade de processamento, resultará em um alto consumo de potência e um custo final elevado; assim como desenvolver redundâncias para reduzir as falhas que, além de elevar o custo e potência consumida, pode modificar, de maneira não esperada, o tamanho e a forma do produto. O requisito tempo de projeto também deve ser muito bem analisado, pois em um mercado mundial globalizado, com evoluções tecnológicas constantes, o tempo para desenvolvimento está ficando cada vez menor e o ciclo de vida dos novos produtos lançados está diminuindo consideravelmente. Assim, um atraso em uma das fases do projeto pode resultar em uma redução drástica dos lucros, como mostra a figura 2.1, no qual um produto é lançado conforme o planejado e com atraso.

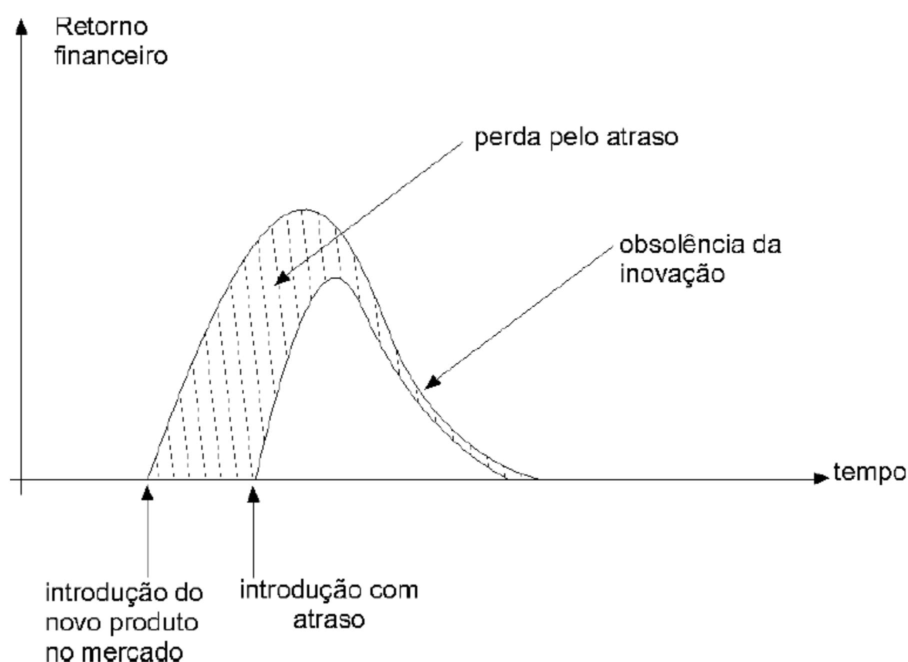

Figura 2.1: Retorno financeiro para o lançamento de um produto com e sem atraso [55]

Alguns requisitos cruciais, como o custo total do projeto e seu tempo de implementação, levaram à criação de novas técnicas de desenvolvimento, entre elas o uso de plataformas virtuais e reuso de componentes. Desta forma, é possível aumentar o nível de complexidade do sistema mantendo o tempo e o custo do projeto em níveis aceitáveis. Na figura 2.2, observa-se que, em 2001, a economia do projeto utilizando novas técnicas de desenvolvimento (linha clara) foi superior a 90% quando comparado com uma estimativa de quanto seria o custo sem tais técnicas.

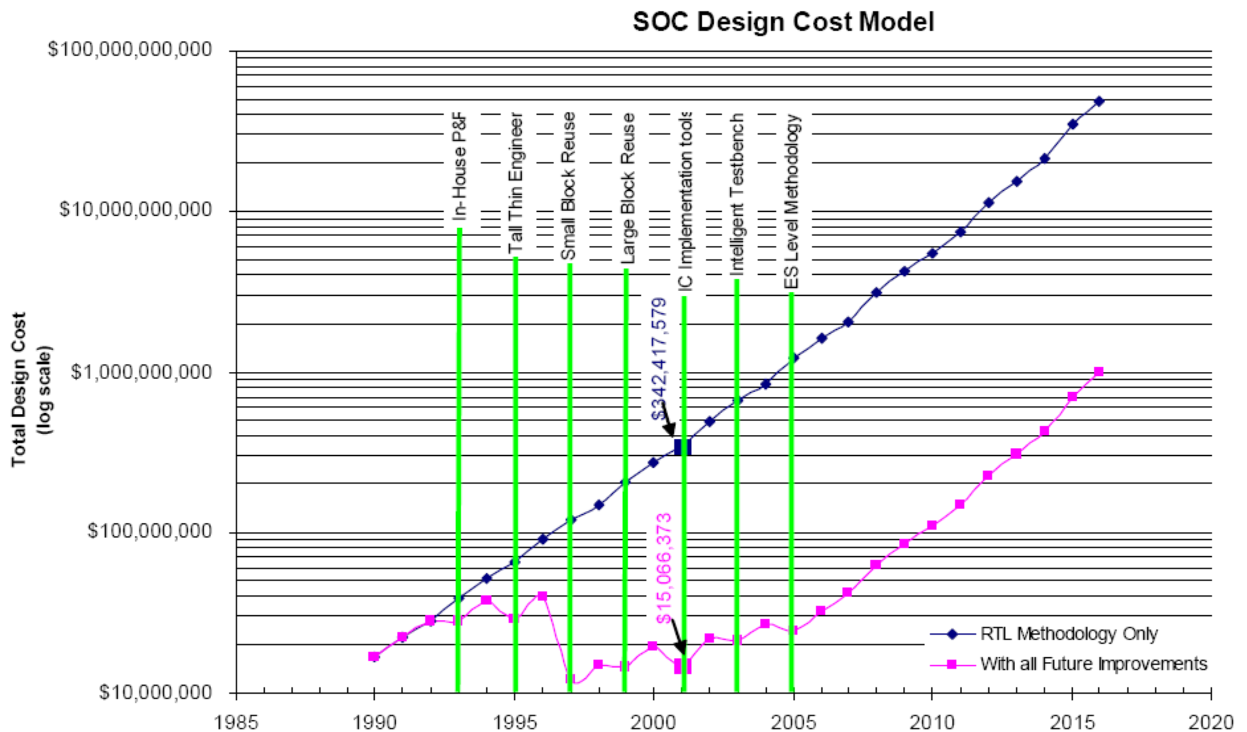

Figura 2.2: Impacto de novas tecnologias no custo do projeto [22]

2.1 Projeto de Sistemas Embarcados

A complexa arquitetura de um sistema embarcado, em que vários componentes de *hardware* e *software* compartilham uma estrutura de comunicação, aliado com a grande variedade de soluções possíveis, faz do projeto a parte mais importante do desenvolvimento. É nesta etapa que os requisitos do sistema são estabelecidos conforme o objetivo, a data do lançamento é estimada visando o máximo de lucro e as técnicas e ferramentas a utilizar são definidas.

2.1.1 Metodologia de projeto

Em um projeto de sistema embarcado, devido à sua complexidade, é recomendado iniciar a especificação em um alto nível de abstração, como descrições de funcionalidades, e seguir detalhando cada nível inferior até alcançar a síntese do leiaute final do circuito. Este cenário está ilustrado na Figura 2.3.

Uma metodologia bem definida impõe regras, padroniza os procedimentos e coordena de maneira eficaz as equipes (e as pessoas) envolvidas; utilizá-las, permite que todos os envolvidos tenham ciência do que está sendo feito, dos avanços já realizados e das subdivisões do projeto, facilitando a comunicação e a troca de idéias entre os membros [56].

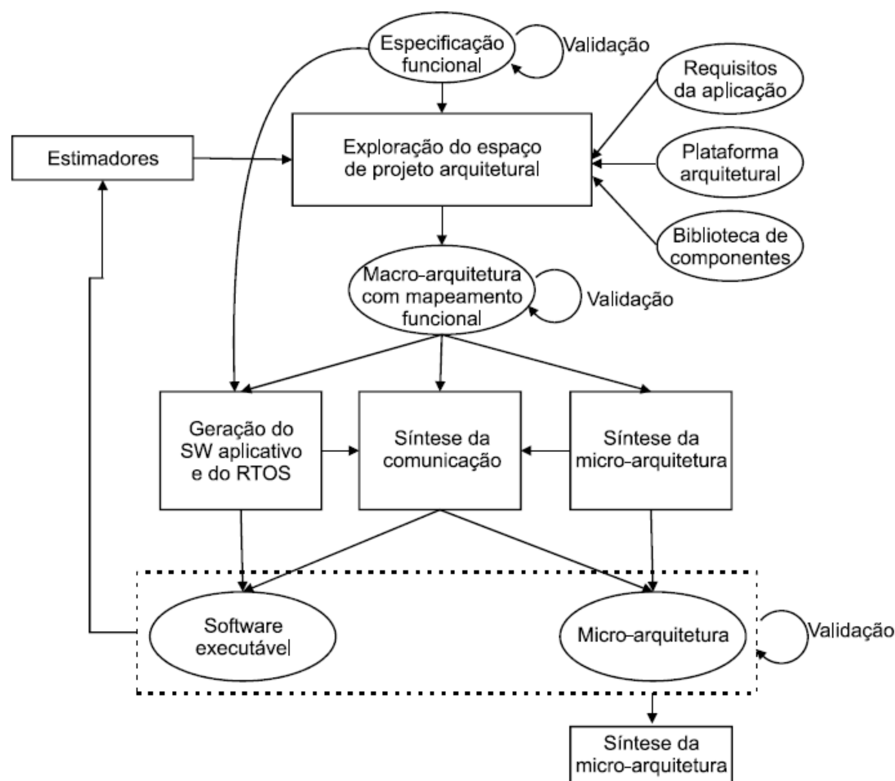

Figura 2.3: Metodologia de projeto de ES [55]

2.1.2 Níveis de abstração

As metodologias indicam que diferentes níveis de abstrações devem ser utilizados na especificação do projeto, porém, não existe uma padronização plena destes níveis, apesar dos esforços patrocinado por diversas empresas [55]. A tentativa de padronização segue o fluxo básico apresentado na figura 2.4.

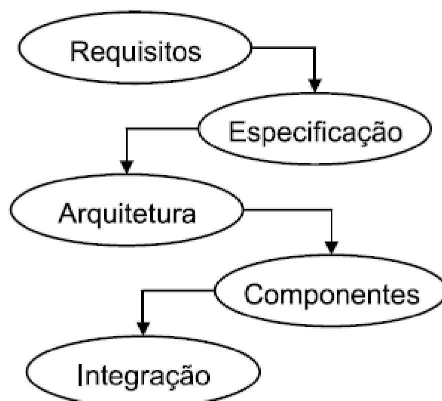

Figura 2.4: Níveis macros de abstração na especificação de projeto [56]

A coleta dos requisitos (nível de abstração mais elevado) permite especificar o sistema de maneira puramente funcional, determinando, por exemplo, se o desempenho ou o consumo de potência, será priorizado. Nesta etapa, não há nenhuma informação estrutural ou dependente da arquitetura-alvo a ser implementada. Na segunda etapa, a especificação, as informações levantadas na fase anterior são descritas de uma maneira mais formal, utilizando linguagens de especificação como UML. Assim, antes de passar à terceira etapa, deve-se estar claro o objetivo do sistema e como será seu comportamento no ambiente.

A fase da arquitetura estabelece os componentes de *hardware* e *software* que serão utilizados, suas funções e como eles serão interconectados. Tudo é documentado, com cartões CRC (Classes, Responsabilidades e Colaboradores) por exemplo, e passado à etapa 4, responsável pelas implementações dos componentes. Nesta etapa, com base nos estudos e discussões que foram realizados para esclarecer detalhes e dúvidas sobre o projeto, a construção do *hardware* e do *software* é conduzida de forma concorrente, aumentando a produtividade. Devido à técnica do reuso, alguns componentes poderão estar prontos, ou precisarão de alguns ajustes para se adequar ao projeto, que também é feito nesta fase. A integração é a fase onde grande parte dos erros são descobertos, tudo é encaixado e trabalhos de diferentes equipes são unidos. Por isso, é considerada a etapa mais desafiadora [56].

2.1.3 Particionamento entre *software* e *hardware*

O particionamento ocorre na fase arquitetural do projeto e objetiva equilibrar o desempenho esperado com o custo do produto. A entrada para este processo é a especificação funcional do sistema e a saída é uma macro-arquitetura composta por processadores e blocos de *hardware* dedicado e um mapeamento das funcionalidades em cada componente da arquitetura.

Uma abordagem clássica supõe inicialmente que o sistema será inteiramente desenvolvido em *software*, sobre um processador conhecido e não configurável. Uma avaliação de desempenho através de um simulador, por exemplo, permite identificar as partes críticas da aplicação para serem movidas à blocos dedicados de *hardware*. O processo é repetido até que uma solução aceitável seja encontrada [55], lembrando que fabricar vários elementos de *hardwares* dedicados aumenta o desempenho e eleva consideravelmente o preço do projeto e do produto final.

Algumas abordagens para particionamento automático estão sendo estudadas [6][8][25] [38], porém, por conta da complexidade computacional, o escopo de atuação ainda é pequeno e limitado.

2.2 Simulação de arquitetura

Simulação através de plataformas virtuais é uma técnica muito utilizada atualmente para encontrar arquiteturas eficientes para aplicações de diversos domínios. Isso é possível devido à possibilidade de reconfigurar facilmente parâmetros como tamanho das *caches*, políticas de atualização, número de unidades funcionais, quantidade de processadores, tipo de barramentos, entre outras opções, e dessa forma, efetuar um estudo detalhado das propriedades do sistema e observar seu funcionamento. Conseqüentemente, tal técnica diminui o risco de fracassos no projeto, pois antes de gastar tempo e dinheiro em implementações de mais baixo nível (na etapa componentes da Figura 2.4), pode-se verificar se a aplicação objetivo atenderá os requisitos estabelecidos inicialmente, como desempenho e consumo de potência, validando, assim, tanto o *software* quanto o *hardware*.

Com o uso de plataformas virtuais, então, uma configuração arquitetural próxima do esperado pode ser conseguida e além disso, a validação do *software* embarcado pode ocorrer bem antes da síntese do *hardware*. Essa independência é necessária atualmente devido à complexidade algorítmica que as soluções estão exigindo das aplicações, e esperar a construção física do protótipo para realizar os testes iniciais e intermediários, pode demandar tempo, atrasando o projeto.

A figura 2.5 mostra o fluxo tradicional de desenvolvimento e integração de componentes de *hardware* e *software*. Através da simulação de arquitetura, a **espera pelo protótipo** não precisa acontecer, pois os arquivos binários já desenvolvidos até o momento podem ser testados na plataforma virtual. Com isso, ganha-se em paralelismo de trabalho, pois, pelo fluxo tradicional, se um atraso no **desenvolvimento do *hardware*** acontecer, a continuação do **desenvolvimento do *software*** é comprometida. Segundo Zivojnovic [59], uma economia de até 40% no tempo total de desenvolvimento pode ser obtida através do uso de simulações.

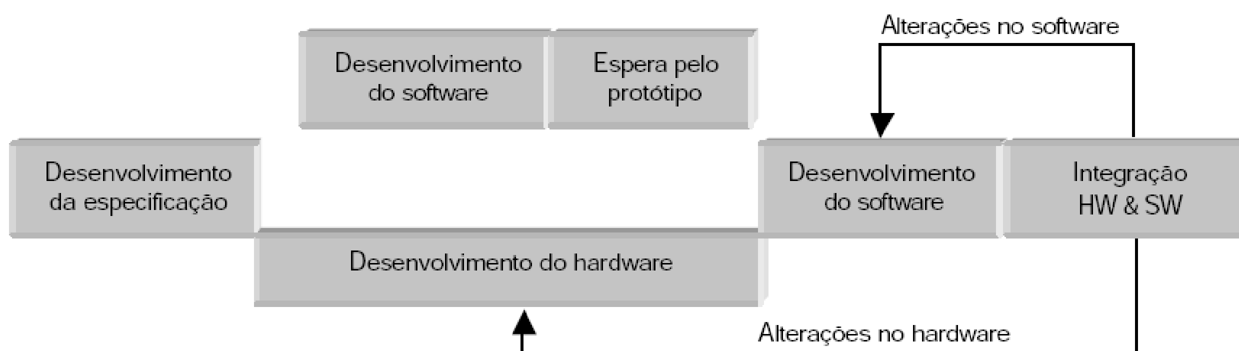

Figura 2.5: Fluxo tradicional de integração de *hardware* e *software* [50]

O nível de detalhe incluído em uma ferramenta de simulação tem que ser considerado, pois se poucos detalhes estão implementados, os resultados podem não ser confiáveis. Não obstante, detalhar demasiadamente um simulador faz sua execução ser demorada, podendo, dependendo da aplicação simulada, não trazer resultados em tempos hábeis. Tais aspectos são importantes quando as arquiteturas a serem testadas não possuem um simulador ou não está corretamente validado (trazendo resultados insatisfatórios), sendo necessário seu desenvolvimento.

Ferramentas de simulação podem ser construídas em qualquer linguagem de propósito geral. Contudo, existem várias linguagens e bibliotecas que dão um suporte mais apropriado quando um mapeamento de *hardware* em *software* está sendo realizado, oferecendo elementos típicos de *hardware*, como portas, sinais e *clock*. Uma delas é o SystemC, uma biblioteca C++ utilizada na implementação do simulador VIPRO-MP e descrita na Seção 4.3.

Capítulo 3
Processamento Paralelo

Há 40 anos, Gordon Moore propôs, observando o ritmo de evolução da tecnologia, que seria possível duplicar o número de transistores numa mesma área de silício aproximadamente a cada 18 meses, mantendo o mesmo custo de produção. Com a miniaturização, aumenta-se a freqüência e conseqüentemente o desempenho do processador. Desta forma, um processador produzido no começo de 2004 teria o dobro de desempenho de um produzido em julho de 2002. Essa declaração, conhecida como Lei de Moore, ditou os passos da indústria de processadores até os dias atuais, elevando o número de transistores de 29.000, em processadores do começo da década de 80, para 1,7 bilhões, em processadores contemporâneos [26].

Por mais que essa lei promoveu o progresso e o desempenho dos núcleos ao longo das décadas, estamos chegando na capacidade atômica do silício, e não teremos mais o ganho em freqüência que tínhamos a poucos anos. Além disso, quanto menores são os transistores em um circuito integrado, maior sua densidade (seu número por unidade de área), aumentando a produção e a concentração de calor devido à dissipação de potência. Assim, para o funcionamento correto e maior vida-útil, meios para refrigerar o sistema devem ser utilizados.

Na Figura 3.1, o consumo de potência, em Watts, aumenta consideravelmente à cada nova minimização dos transistores; o termo **tecnologia**, neste contexto, refere-se à escala utilizada na fabricação dos transistores, atualmente, em nanômetros. A dissipação média de potência em um circuito pode ser decomposta em duas parcelas: (i) potência estática (*leakage power*) e (ii) potência dinâmica (*dynamic power*). A parcela estática é a função da corrente de fuga (em inglês, *leakage current*) dos transistores e é obtida basicamente nos períodos em que não ocorrem chaveamentos [12]. Percebe-se que esta parcela está ficando cada vez mais significativa com as novas tecnologias. A parcela dinâmica é decorrente da atividade do sistema, ou seja, da execução dos aplicativos, e com 70nm, é superada pela parcela estática.

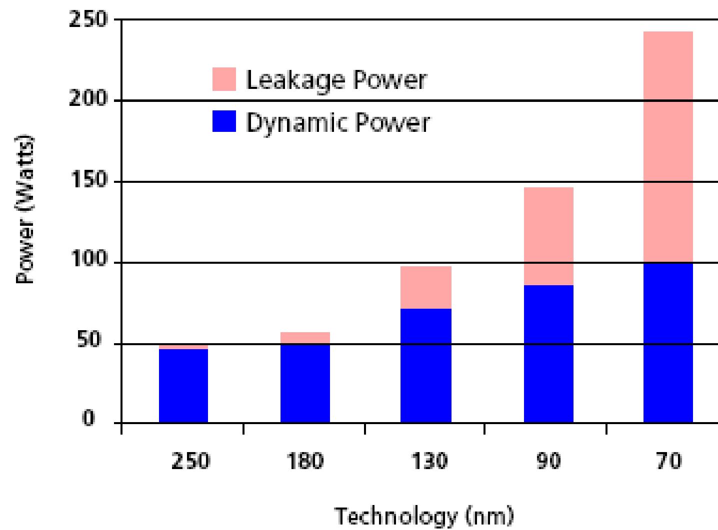

Figura 3.1: Impacto no consumo de potência conforme a tecnologia [36]

Para sistemas embarcados, tais fatores são críticos, pois na maioria das vezes, o sistema é alimentado por bateria, sendo o consumo de potência e energia um requisito importante. Por isso, a técnica de conseguir desempenho aumentando a freqüência está enfrentando sérios problemas e o processamento paralelo, com múltiplos núcleos de menor freqüência, está sendo uma alternativa à, quase derrotada, lei de Moore.

3.1 Níveis de Paralelismo

O paralelismo pode ser explorado em diversos níveis nas máquinas mais recentes. Em seu menor nível de granularidade, ele se apresenta no nível de instruções de máquina, sendo denominado ILP (*Instruction Level Paralelism*).

Em qualquer aplicação é possível identificar algumas instruções, ou trechos de código, que são independentes entre si, podendo, portanto, ser executados concorrentemente sem que isto prejudique a semântica da aplicação. Este nível já é bastante explorado, tendo em vista sua absoluta necessidade para que possa haver execução *pipeline*. Após alguns estudos e avaliações, o *pipeline* foi reestruturado e ampliado para o que atualmente é conhecido como **arquitetura superescalar** [40]. Estas arquiteturas possuem a capacidade de executar instruções com paralelismo simultâneo, além daquele com sobreposição parcial, no qual uma instrução é quebrada em pequenas partes, e cada uma delas consome uma fração do tempo necessário para executar o trabalho todo [40].

Porém, o paralelismo de instrução necessita de mecanismo para análise de dependências e suas resoluções, o que, conforme a gama de instruções a executar, é complexo ou impossível. Em virtude disto, a extração de paralelismo neste nível é limitado. Para obter ganhos maiores, faz-se também necessária a exploração de paralelismo de mais alto nível, como os de *thread* e de aplicação.

O paralelismo de *thread,* ilustrado na Figura 3.2, ocorre quando, em uma aplicação, existem diferentes fluxos de execução, ou *threads*, que podem ser executadas completamente em paralelo, podendo haver um ou mais pontos de sincronismo entre eles. Conforme ilustra a figura, o **Programa principal**, dispara *threads* independentes entre si para diferentes processadores, e espera as respostas para continuar, deixando a execução mais eficiente e ocupando os três processadores disponíveis. O paralelismo de aplicação é ainda mais simples: duas ou mais aplicações, totalmente independentes, são executadas em paralelo, não havendo entre elas nenhum ponto comum.

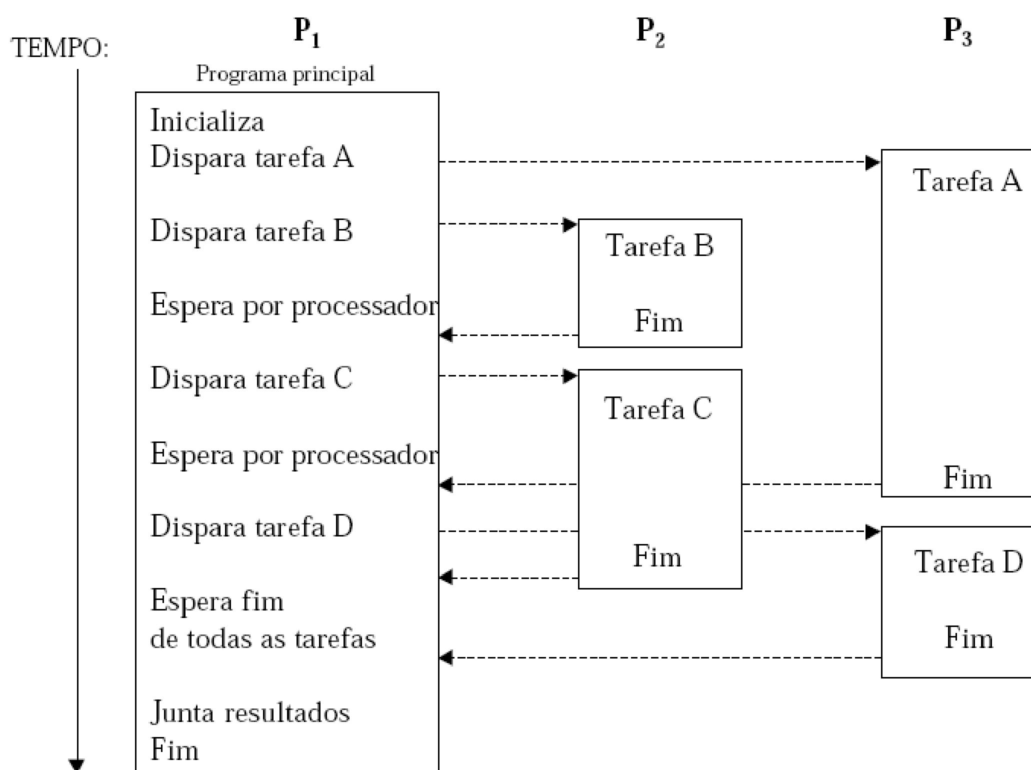

Figura 3.2: Paralelismo de *threads*

Nestes cenários, sistemas contendo apenas um núcleo de processamento se tornam, muitas vezes, ineficientes, pois, mesmo com todo o esforço do programador em dividir a execução dos aplicativos em *threads* e de desenvolver *softwares* que executam paralelamente, na

realidade, apenas um fluxo de instrução atua sobre um fluxo de dados, utilizando, no máximo, *pipeline* quando possível. Assim, várias trocas de contexto são necessárias para atender este nível de paralelismo, prejudicando o desempenho global do sistema. Visando resolver este impasse, múltiplos núcleos, em vez de um único, estão sendo utilizados por processador, dando origem às **arquiteturas multiprocessadas**. Dentro deste conceito está, também, incluída a opção similar de encapsular vários processadores em um único *chip,* o que, em sistemas embarcados, é denominado MPSoC (*multiprocessor system-on-chip*).

Uma outra organização de máquina paralela é interligar vários computadores por uma rede (*cluster*), o que comumente recebe o nome de **multicomputador**. Devido à organização, cada processador (ou máquina) tem sua própria memória privada, e acessa apenas ela, sendo chamado de modelo de memória distribuída. Isso requer atenção especial do programador, pois uma distribuição malfeita dos dados, entre as memórias, pode levar o sistema a uma perda considerável de desempenho, pois trocas de dados adicionais compromete a rede de interconexão e o processamento dos processos envolvidos. Porém, tais sistemas são amplamente utilizados por causa de sua alta escalabilidade, podendo adicionar ou retirar pontos de processamento sem suspender os pontos em funcionamento.

3.2 Arquiteturas multiprocessadas

Os chamados processadores *multicore* e os MPSoC se tornaram os principais produtos para computação de propósito geral e para sistemas embarcados, respectivamente. O uso de múltiplos núcleos, apesar de ser um nicho relativamente novo a ser explorado, não é uma idéia nova, pois Gelsinger e colaboradores, em 1989, escreveram o artigo *Microprocessors Circa 2000* [14], em que já se previa o aparecimento de processadores multinúcleos na virada do ano 2000. O motivo desta convergência se deve à necessidade de alcançar maiores desempenhos através da execução de diversas tarefas simultâneas, sem o conseqüente aumento no consumo de energia e da geração de calor, como será relatado na Seção 3.2.2. Uma questão importante, ao utilizar essas arquitetura, é quanto ao gerenciamento da memória principal, e será discutida na seção subseqüente.

3.2.1 Modelos de memória compartilhada

O termo multiprocessador e arquitetura multiprocessada está intimamente vinculado com este modelo, pelo fato de todos os processadores existentes no sistema compartilhar um espaço de

endereçamento. Apesar de ser, reconhecidamente, mais fácil de programar, e de oferecer recursos que os multicomputadores não dispõe, tal modelo reduz a escalabilidade do sistema, pois à medida que novos processadores vão sendo incluídos, torna-se mais complexo o gerenciamento da memória e aumenta o conflito no barramento.

Os sistemas multiprocessador se dividem de acordo com a maneira de implementação de sua memória compartilhada. As três classes são UMA (*Uniform Memory Access*), NUMA (*Non- Uniform Memory Access*) e COMA (*Cache-Only Memory Access*). Todos os sistemas de uma determinada classe possuem características específicas, tanto na implementação quanto na programação.

3.2.1.1 Modelo UMA

Em sistemas que seguem tal modelo, o acesso a qualquer módulo de memória é feito de forma uniforme entre todos os processadores, ou seja, o tempo de acesso à memória é constante. Para conseguir essa característica, os múltiplos processadores são ligados por meio de uma rede de interconexão (normalmente barramento) a uma memória global centralizada. A Figura 3.3 ilustra este modelo, no qual o M abaixo da rede de interconexão é o bloco de memória compartilhada.

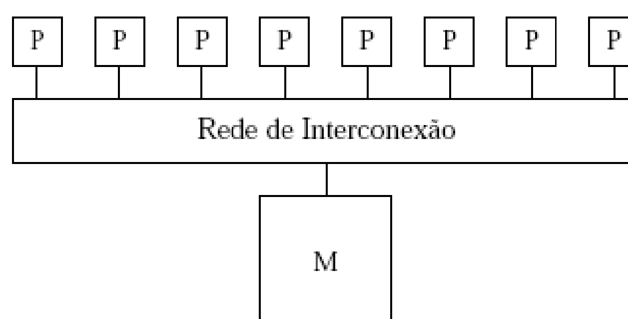

Figura 3.3: Modelo UMA

Os sistemas SMP (*Symmetric MultiProcessor*), composto por processadores de capacidade de processamento equivalente e que compartilham o mesmo conjunto de instruções, utilizam-se deste modelo. Como exemplo, tem-se os processadores Core 2 Duo [21], da Intel, em que dois núcleos idênticos são encapsulados, acessando a memória de maneira compartilhada.

3.2.1.2 Modelo NUMA

Neste modelo, cada processador possui uma memória local, a qual é agregada ao espaço de endereçamento global da máquina. Dessa forma, existe o conceito de memória próxima

(acessada localmente) e memória distante (acessada remotamente), cada uma oferecendo tempos de acessos diferentes. Assim, a não-exigência de tempo uniforme no acesso à memória elimina a grande limitação de escalabilidade dos sistemas UMA.

Um agravante neste sistema é com relação às *caches* dos processadores, utilizadas em sistemas NUMA complexos, conhecidos por CC-NUMA (*Cache Coherent - Non-Uniform Memory Access*). Como todos os processadores podem modificar a memória local de todos, o risco de a *cache* de um certo núcleo estar com uma versão desatualizada do dado é grande. Assim, mecanismos de coerência de *cache* devem ser implementados, dificultando a construção do modelo. Uma alternativa, que simplifica a construção, é não implementar tais mecanismos de coerência, impondo a retirada das *caches* dos processadores, sendo denominado NC-NUMA (*Non-Coherent - Non-Uniform Memory Access*) [40].

Uma das primeiras máquinas NC-NUMA foi a **Carnegie-Mellon Cm*** [47], composta por uma coleção de CPUs LSI-11 [9], cada uma com uma memória acessível diretamente através de um barramento local. Os nós eram interconectados por um barramento de sistema. A Figura 3.4 mostra como tal máquina era organizada.

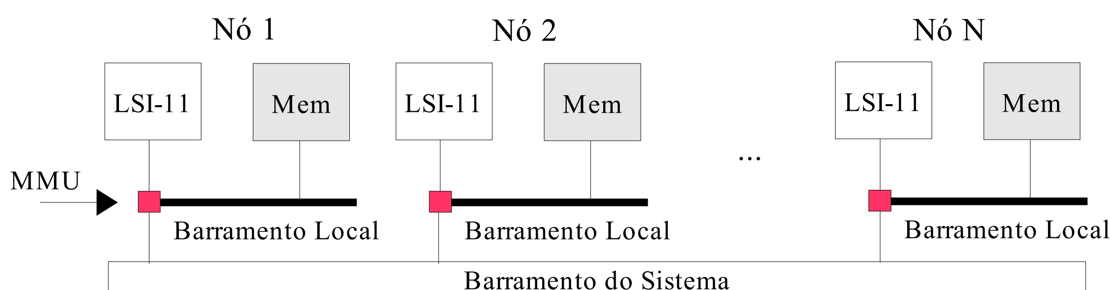

Figura 3.4: Organização da máquina Carnegie-Mellon Cm* [47]

A MMU (*Memory Management Unit*), em cada referência à memória, verificava se o endereço pertencia à memória local. Se sim, a requisição era feita à memória local, via barramento local. Caso contrário era feita ao nó remoto, através do barramento de sistema [47].

Uma outra variante do modelo NUMA [32] acrescenta, além das memórias locais aos processadores, um bloco de memória global, conectada diretamente na rede de interconexão (barramento do sistema). Isso deixa cada processador com três tipos de acessos:

- local: quando interage com a própria memória;
- remoto: ao solicitar um endereço de memória referente à memória local de outro processador; e
- global: quando o endereço solicitado está na memória compartilhada global.

3.2.1.3 Modelo COMA

Como demonstrado anteriormente, as máquinas UMA possuem bom desempenho, mas escalabilidade limitada, enquanto que as máquinas NUMA possuem grande escalabilidade, mas o desempenho pode ser inferior por causa da distribuição dos dados e da coerência de *cache* [41]. O modelo COMA advém da alternativa radical de se eliminar a memória principal, transformando-a em grandes *caches* privadas. Essa descentralização no acesso aos dados permite às máquinas COMA um grande potencial em ganho de desempenho, mas, apesar disto, poucas máquinas COMA foram construídas até o momento devido ao alto custo de produção.

3.2.1.4 Modelo VIPRO-MP

O modelo implementado no VIPRO-MP não pertence especificamente às classes supracitadas. Isto porque, cada processador da plataforma possui uma memória privada (local), no qual nenhum outro processador tem acesso, mas a arquitetura oferece uma memória compartilhada (global) para troca de informação. Quando o processador solicita endereços abaixo de 0x80000000, a memória privada é acessada, e endereços acima, a memória compartilhada é requisitada. Assim, não pertence ao modelo UMA pois os tempos de acessos podem ser diferentes entre a memória local e a global e, como cada processador acessa sua memória privada em uma faixa de endereço independente, também não pertence ao NUMA.

Este modelo é amplamente implementado em sistemas embarcados, como no processador Nexperia [17] que, além da memória privada, possui uma memória compartilhada, utilizada para envio de mensagens entre os processadores. O processador CELL [19], para sistemas embarcados, da parceria IBM, Toshiba e Sony, também implementa esta variante sobre os núcleos SPEs (*Synergistic Processor Element* - processador RISC especializado em computação vetorial). Este processador possui 9 núcleos: 8 SPEs e um PPE (*PowerPC Processor Element* – processador RISC de 64 bits responsável pelo controle de tarefas do sistema operacional). Cada núcleo tem completo acesso à memória global (semelhante à memória compartilhada do VIPRO-MP) e as instruções de leitura e escrita podem ser feitas simultaneamente e de maneira independente [19]. Os SPEs possuem, ainda, uma pequena memória local (*Local Store* – LS), para o armazenamento de instruções e de dados. O Playstation 3, da Sony, faz uso deste processador embarcado.

3.2.2 Medidas de desempenho e consumo de potência

Nos multiprocessadores e *multicores*, o desempenho não é mensurável somente através das velocidades dos núcleos, mas, também, na vazão de processamento que esta nova abordagem oferece. O motivo é o aumento dos recursos disponíveis, executando mais funções em paralelo. Então, é possível que um processador de 3GHz tenha um desempenho menor que um processador de 4 núcleos à 500MHz. Isso ocorre, pelo fato da tecnologia permitir realizar mais tarefas ao mesmo tempo e, se preciso, dedicar alguns núcleos à tarefas mais dispendiosas e de maior prioridade.

A utilização de soluções multiprocessadas apresenta vantagens significativas, também, em requisitos como consumo de energia se comparada a soluções monoprocessadas. De acordo com Givargis [15], para os circuitos integrados CMOS (*complementary metal-oxide-semiconductor*), pode-se adotar a seguinte equação para o consumo de potência:

$$P = \frac{C \times A \times F \times V^2}{2} \tag{3.1}$$

onde C é a capacitância média de chaveamento, A é um termo de 0.0 a 1.0 que determina a atividade média de chaveamento, F é a freqüência de relógio aplicada no circuito e V é a tensão aplicada para que ocorra o chaveamento das portas lógicas.

Sendo a freqüência F uma aproximação linear da função que determina V, visto empiricamente em um StrongARM SA-1100 (Figura 3.5), o consumo de potência pode ser minimizado através da redução da freqüência em conseqüência da redução da voltagem. E, devido à característica da fórmula 3.1, uma redução na voltagem traz um minimização quadrática da potência consumida.

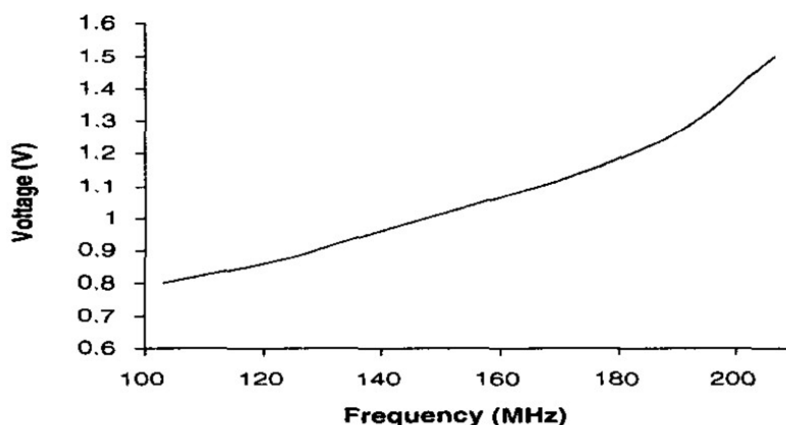

Figura 3.5: Freqüência *vs* Voltagem em um StrongARM SA-1100 [45]

Assim, se o *software* for corretamente paralelizado, utilizar mais núcleos com freqüências e voltagens menores, reduzirá o consumo de potência sem prejudicar o desempenho. Esta é a solução adotada por diversos processadores de propósito geral atualmente, como os Core 2, da Intel, e os Athlon X2, da AMD.

Outros recursos podem ser utilizados para aumentar ainda mais a economia de energia dissipada, como desativar unidades funcionais que são utilizadas por um curto espaço de tempo e reduzir o tamanho das *caches*, responsável por grande parte da potência consumida em um processador. A Figura 3.6 exibe as porcentagens do consumo de potência em um sistema utilizando um processador ARM9, sendo a D-Cache (*Data Cache*) e a I-Cache (*Instruction Cache*) responsável por 44% do total consumido.

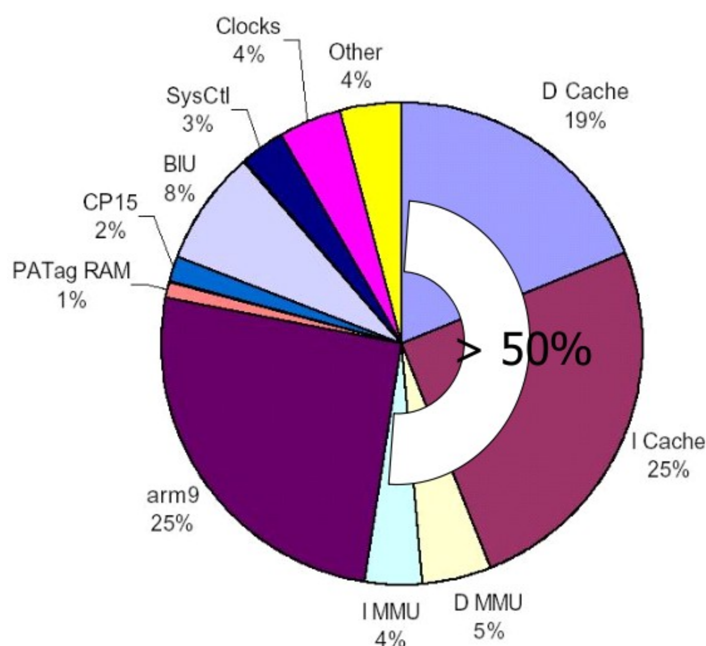

Figura 3.6: Consumo de potência em um sistema com ARM9 [57]

Tais fatores justificam a intensa pesquisa industrial e acadêmica para com os sistemas *multicore,* no qual o desempenho é medido pela eficiência e não mais pela freqüência, o que da início à era da computação medida pela eficiência [20].

3.3 *Software* para Sistemas Paralelos

Uma aplicação em *software* é a ponte entre o usuário e a plataforma de *hardware*. Por isso, uma mudança arquitetural impacta diretamente na construção do *software*, necessitando de novos mecanismos para conseguir desfrutar plenamente dos recursos da nova arquitetura. Assim, migrar de uma arquitetura monoprocessada para uma multiprocessada, requer que os aplicativos construídos sob a forma seqüencial também sejam modificados.

O principal problema é que aplicações que contém apenas um fluxo de instrução, que na literatura são comumente chamadas de *single-threaded,* não conseguem tirar proveito das características das novas arquiteturas *multicore* ou multiprocessadas. Em outras palavras, o desempenho de uma aplicação desse tipo não melhora simplesmente por executarmos o programa em um sistema multiprocessado. Linguagens como C e C++ são muito comuns no desenvolvimento de aplicações de propósito geral e para sistemas dedicados, mas não oferecem, na própria linguagem, mecanismos para ajudar no particionamento da aplicação. Daí o surgimento de bibliotecas e APIs (*Application Programming Interface*) especiais para suportar o que é conhecido como programação paralela ou programação *multithread.* Já a linguagem Java, que vem despontando também no ramo de embarcados, possui suporte nativo à este tipo de programação [48]. Desta forma, programação paralela permite que o programador informe os pontos que podem ser executados concorrentemente dentro de uma aplicação, deixando a arquitetura multiprocessada eficiente.

Alguns pontos devem ser cuidadosamente analisados para a implementação de um *software* que explore o fator concorrência [1]:

- Informar a concorrência: é responsabilidade do programador saber os trechos de códigos que poderão ser executados paralelamente. Essa tarefa pode ser simples em alguns problemas que são claramente paralelizáveis, porém, em termos gerais, é uma tarefa difícil e, para determinados problemas, impossível.
- Projeto do algorítimo: projetar desde o início ou alterar o projeto de um algoritmo existente de maneira que ele se torne paralelo, isto é, explicitamente concorrente. Como já dito, trivial para alguns casos e complexo para outros.
- Comunicação: sincroniza e faz a troca de informações entre os vários processadores existentes no sistema. Para projetar mecanismos e algoritmos eficientes para este fim é necessário a consideração de alguns fatores, como: tempo para estabelecimento de uma comunicação, quantidade de dados na transmissão, e previsão de tempo de

resposta. A eficiência do sistema depende da infra-estrutura de *hardware* que está disponível para comunicação, mas também de como o *software* foi projetado e particionado.

Uma ferramenta (linguagem, API ou biblioteca) que ofereça apoio à programação paralela é utilizada pelo programador para traduzir o projeto do algoritmo paralelo da solução, para uma aplicação paralela (concorrente).

Um aspecto importante a ser considerado é a modelagem da memória no sistema, podendo ser compartilhada ou distribuída, caso sistemas multicomputadores estejam, também, sendo considerados. No segundo modelo, a comunicação e sincronização são realizadas por trocas de mensagens, carregando dados e informações semânticas. Já no primeiro, é através de variáveis globais, declaradas no espaço compartilhado, para que todos os núcleos tenham acesso. Algumas aplicações são melhores executadas em um dos modelos, e mapeá-las para o outro, pode trazer ineficiência. Assim, antes de projetar o algoritmo e de escolher a ferramenta à utilizar, deve-se verificar se o modelo será adequado ao *software* (máximo de desempenho), ou se o *software*, ao modelo (reuso de estruturas e componentes).

Dentre as ferramentas existentes para programação *multithread,* estão: a biblioteca *pthread* [46] (memória compartilhada) para C e Fortran, a API OpenMP [49] (memória compartilhada) para C/C++ e Fortran e a biblioteca MPI [37] (memória distribuída) para C e Fortran

Ao implementar um algoritmo paralelo em alguma ferramenta que ofereça suporte, alguns paradigmas podem ser seguidos [2], sendo os mais utilizados descritos à seguir:

- Mestre/Escravo: uma *thread* mestre executa as tarefas essenciais do programa paralelo e divide o resto das tarefas para processos escravos. Quando um processo escravo termina sua tarefa, ele informa o mestre que lhe atribui uma nova tarefa. Este paradigma é bastante simples, visto que o controle está centralizado num processo mestre; sua desvantagem é que o mestre torna-se um gargalo na comunicação.
- *Pool* de Trabalho: um *pool* (conjunto) de tarefas é disponibilizado por uma estrutura compartilhada (global) e os processos, executados em núcleos distintos, buscam nesta estrutura o que deve processar. O programa paralelo termina quando o pool de trabalho fica vazio. Este tipo de modelo facilita o balanceamento da carga; por outro lado, além da sua dificuldade de implementação em memória distribuída, é difícil obter um acesso eficiente e homogêneo aos múltiplos processos .

- Divisão e conquista: dividir uma tarefa em diversas partes menores e atribuí-las aos processadores ociosos. O resultado do problema inicial é a combinação dos vários subresultados encontrados. O paradigma Mestre/Escravo é englobado por este modelo, porém, os processos podem se comunicar ao longo da execução diretamente, sem interferência do mestre.

3.3.1 O futuro da programação *multicore*

Apesar dos modelos de algoritmos e das ferramentas de apoio à programação paralela, construir aplicativos eficientes neste estilo de programação ainda é altamente complexo; mesmo com poucos núcleos sendo considerados. Com o fracasso da lei de Moore, sistemas com dezenas de núcleos serão projetados afim de suprir a exacerbada procura por desempenho [18]. Porém, isto só será alcançado se o *software* conseguir tirar proveito de todos os recursos, fato que a programação paralela atual não está apta a fazer, se mostrando altamente difícil e complicada.

Assim, de nada adianta ter um poder computacional gigantesco, se o *hardware* está à frente da capacidade dos programadores em fazer programas que os possam utilizar de forma eficiente [52]. Em outras palavras, os chips *multicore*, embora sejam o sonho do alto poder de processamento, estão se tornando o pesadelo da programação [52].

Como será resolvido esta questão ainda é incerto. Segundo Kirsch [apud 52],

> “nós programamos em linguagens seqüenciais. Iremos precisar de expressar nossos algoritmos em um nível mais alto de abstração? Pesquisas nesta área são críticas para nosso sucesso."

Capítulo 4

Ferramentas Utilizadas

Para a construção do simulador de arquiteturas multiprocessadas VIPRO-MP, algumas ferramentas já desenvolvidas foram utilizadas. As relações entre elas estão expostas na Figura 4.1, no qual as partes pontilhadas foram as implementações realizadas neste trabalho.

O SimpleScalar [5] é um conjunto de ferramentas de simulação e modelagem de arquiteturas de processadores que vem sendo amplamente utilizado por diversos grupos de pesquisas. No entanto, a potência que o aplicativo irá consumir não é estimada, o que, em um projeto de sistema embarcado, pode ser crucial. O simulador Sim-Wattch [4], desenvolvido sobre a plataforma SimpleScalar e utiliza a primeira versão da biblioteca CACTI [24] para estimar a potência, supre esta necessidade.

A biblioteca SystemC [51] foi utilizada devido aos recursos típicos oferecidos quando elementos de *hardware* são modelados. A biblioteca CACTI utilizada no Sim-Wattch, desenvolvida em 1994, foi atualizada para a versão 4.1 [53], de 2006, que oferece mais precisão no consumo de potência dos elementos.

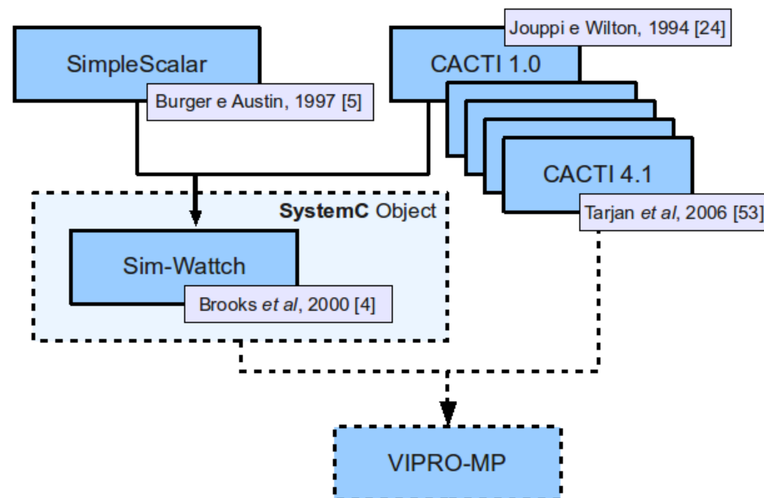

Figura 4.1: Relacionamento das ferramentas utilizadas

4.1 A ferramenta SimpleScalar

O SimpleScalar [5] é fruto de um trabalho iniciado na Universidade de Wisconsin-Madison, em 1997, e foi validado por diversos centros de pesquisas, tanto de caráter acadêmico quanto industrial. Essa ferramenta consiste em grupo de simuladores de processadores superscalares, compiladores e depuradores, e pode ser configurada para simular arquiteturas Alpha, PISA, ARM ou x86 [44]. Sua distribuição é por código aberto, facilitando a adaptação ao projeto em pesquisa, e gratuita para fins educacionais.

Devido ao objetivo do trabalho, o SimpleScalar foi compilado para simular arquiteturas PISA (*Portable Instruction Set Architecture*), executando, portanto, apenas binários no formato ECOFF [5]. Esta arquitetura foi escolhida devido ao seu conjunto de instruções ser derivado da ISA do MIPS-IV.

Dentre os oito simuladores do pacote, o mais completo é o *sim-outorder,* que incorpora previsão de desvios, renomeação de registradores e execução fora-de- ordem, em um *pipeline* de 6 estágios: busca, despacho, escalonamento, execução, retorno e conclusão. Este simulador permite a configuração de praticamente todos os recursos, como tamanho da fila de busca de instruções, a largura de decodificação, o número de estações de reserva da *Register Update Unit* (RUU), a largura da remessa, fila de *load/store* (LSQ), a quantidade de unidades funcionais, as *caches* de instruções e de dados dos níveis 1 e 2. A Figura 4.2 exibe um diagrama da implementação do *sim-outorder* e seus estágios do *pipeline.*

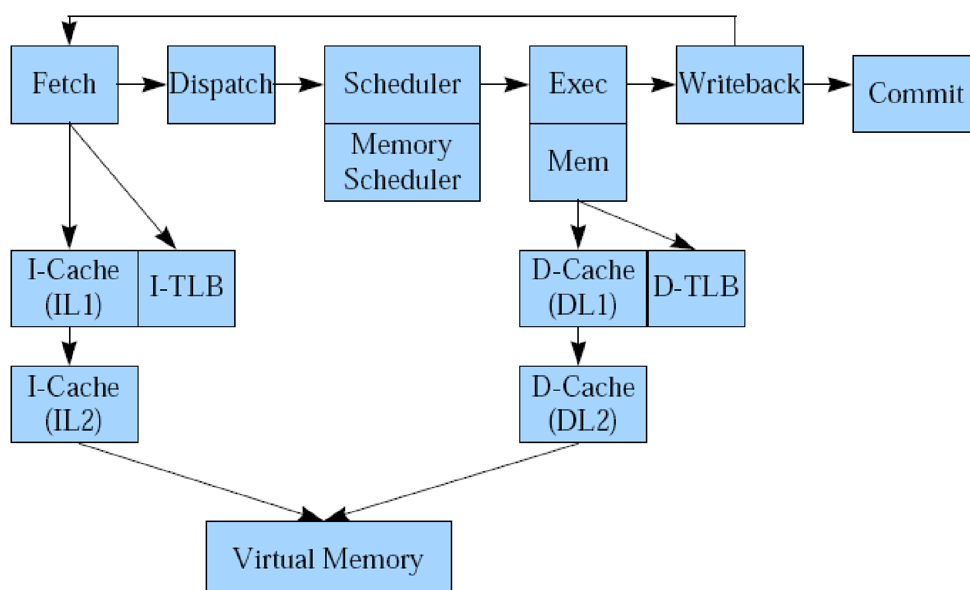

Figura 4.2: Diagrama da implementação do *sim-outorder* [5]

No primeiro estágio a busca de instruções é realizada junto com a previsão de desvios, e, se algum desvio é detectado e previsto como tomado, a busca é bloqueada e redirecionada, de acordo com o endereço alvo. Em *Dispatch* a decodificação da instrução e a renomeação dos registradores é realizada. A cada ciclo, o estágio de despacho encaminha um determinado número de instruções para a fila de escalonamento.

No estágio *Scheduler,* é realizado o escalonamento das instruções e, a cada ciclo, as instruções cujos operandos já se encontram disponíveis são encaminhadas para as unidades funcionais especializadas. As instruções podem ser executadas fora da ordem original. Na execução, as instruções são executadas de acordo com a disponibilidade de cada tipo de unidade funcional, as quais ficam ocupadas pelo tempo de latência da instrução. Quando uma instrução é finalizada, o próximo estágio (*Writeback*) verifica na fila eventuais dependências existentes e caso não haja nenhuma, disponibiliza a instrução para o estágio de gravação dos resultados (*commit*); as instruções executadas erroneamente por uma previsão de desvios incorreta são descartadas. O *pipeline* se completa com o *Commit*, que atualiza os registradores, as *caches* de dados, e sinaliza sucesso na execução desta instrução.

4.2 A biblioteca CACTI

A biblioteca CACTI foi implementada com o intuito de oferecer, através de formulações, uma estimativa do consumo de potência de alguns elementos de *hardware*. A versão 1.0 [24] modela apenas *caches* SRAM e o consumo da potência dinâmica é calculado em função do tempo de acesso, pela função *calculate_time().* Esta função recebe como parâmetros as características físicas da *cache*, como tamanho em bytes, tipo de associação, tamanho do bloco e tipo da tecnologia, e retorna os tempos de acesso e a energia consumida por ciclo.

A versão 2.0 [29] incorporou a modelagem da potência, adicionando o parâmetro *voltagem* na estrutura de entrada. A versão 3.0 [42] ampliou a precisão do retorno ao modelar a área, dado as características de entrada.

A versão 4.0 e a 4.1 [53] adicionou, ao conjunto anterior, o modelo de potência estática (*leakage model*) baseando-se em implementações alternativas da versão 3.0, como eCACTI [31] e Hotleakage [58]. A implementação de algumas funções herdadas da versão anterior foram modificadas e a função *calculate_time()* foi substituída pela *cacti_interface()*, adicionando alguns novos parâmetros para melhor refletir os avanços da tecnologia dos semicondutores [16].

A versão mais recente, a 5.1 [54], apresenta algumas melhorias à versão 4.1, tendo como conseqüência o aumento da complexidade nas chamadas das funções de interface. Além da SRAM, esta versão permite que outros tipos de memória, como a RAM, tenham seu consumo de potência mensurável.

4.3 A ferramenta Sim-Wattch

O Sim-Wattch [4] é um simulador desenvolvido em 2000, pela Universidade de Princeton, que estende o *sim-outorder*, do conjunto de ferramentas SimpleScalar, adicionando um estimador de consumo de potência. Antes da integração com o *sim-outorder*, um *framework,* denominado Wattch [4], foi construído se apoiando na biblioteca CACTI 1.0, oferecendo funções para mensurar o consumo de potência de elementos utilizados em um processador. Este *framework* é independente da arquitetura utilizada, podendo ser integrado com outros simuladores arquiteturais, sem ser o SimpleScalar.

O Wattch é cerca de 1000 vezes mais rápido que as ferramentas existentes para estimativa de potência em nível de leiaute e diferindo, em média, 30% de resultados de processadores reais, utilizando dados da Intel [4]. Tal taxa é referente às potências absolutas, porém, quando pesquisas neste escopo são realizadas, às potências relativas tem maior importância, e a taxa de erro, neste caso, é de 10% [4].

O funcionamento do Sim-Wattch é baseado na quantidade de ciclos executados e na freqüência de acesso às estruturas, e, para isso, inicialmente, dada as configurações de *hardware*, a biblioteca CACTI calcula os valores de referência para as estruturas, como mostra a Figura 4.3. Após, a execução do binário inicia-se, e as funções e as variáveis do *framework* Wattch são acessadas a cada ciclo do *sim-outorder*, atualizando a estimativa da potência consumida.

Ao observar o arquivo resultado gerado pelo *sim-outorder* integrado com o Wattch (Apêndice B), percebe-se três estimativas de potência: CC1, CC2 e CC3. O CC1 considera que qualquer acesso a um bloco arquitetural consome a potência do bloco todo, mesmo quando o bloco permite mais do que um acesso simultâneo; CC2 considera que o consumo de potência em um bloco arquitetural que permite mais do que um acesso simultâneo é proporcional ao número de acessos; CC3 se parece com CC2, entretanto, considera um consumo mínimo de 10% (a título de manutenção) mesmo quando um bloco arquitetural não é acessado [16]. Os autores afirmam que o tipo CC3 produz as estimativas mais próximas dos valores obtidos em situações reais.

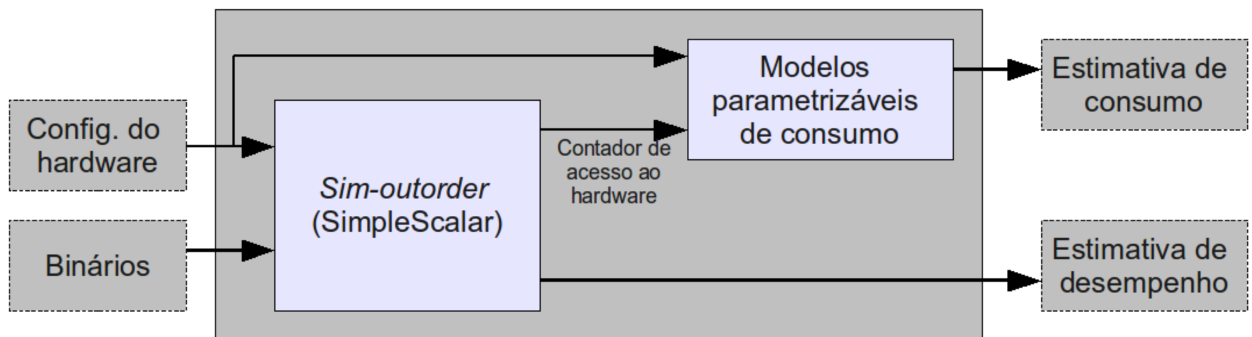

Figura 4.3: Arquitetura do funcionamento do Sim-Wattch [4]

A modelagem proposta no Sim-Wattch obedece a Fórmula 3.1. A variável C, é estimada baseando-se no circuito e nos tamanhos dos transistores, sendo que a voltagem (V) e a freqüência (F) dependem da tecnologia assumida. O parâmetro tecnológico utilizado por padrão é 350 nanômetros. O termo de atividade média de chaveamento depende do binário à executar, porém, considera-se um circuito que pré-carrega e descarrega em todo ciclo, atribuindo, portanto, 1 para A.

4.4 A biblioteca SystemC

A especificação de *hardware* e *software* utiliza ferramentas (linguagens de especificação e simulação) tão diferentes que torna o processo de integração uma tarefa delicada. Assim, muito dos desafios na criação de SoC e MPSoC estão na incompatibilidade entre a linguagem de modelagem, como VHDL e Verilog, e a linguagem de criação de *software*. A linguagem C++, por exemplo, não possui noção sobre o tempo e não implementa concorrência, o que é fundamental para a especificação de sistemas embarcados..

Uma das linguagens propostas para a especificação de sistemas *hardware*/*software* é a biblioteca baseada em C/C++ denominada SystemC [51]. O uso desta biblioteca possibilita a implementação em diversos níveis de abstração, começando de uma aplicação escrita puramente em C++, sendo refinada até o nível RTL [19]. Assim, é possível descrever tanto os componentes de *hardware* como os componentes de *software* de um sistema, permitindo ao usuário utilizar elementos típicos de *hardware*, tais como portas e sinais, dentro do contexto de C++.

A maior desvantagem do uso de SystemC era a falta de resultados de síntese, como área ocupada, freqüência máxima de relógio e consumo de energia. Porém, já existem ferramentas que fazem a síntese de uma especificação SystemC para silício [41].

4.4.1 Metodologias de projetos

Há diferenças significativas entre a metodologia tradicional e a utilizando SystemC. A possibilidade de refinamento de uma linguagem de alto nível para níveis mais baixo trás vários benefícios, como evitar inconsistências, erros e interpretação ambígua da especificação. É possível, ainda, validar a funcionalidade do sistema e verificar seu desempenho antes de sua implementação física. A verificação também se torna bem mais simples, pois os mesmo casos de testes podem ser utilizados durante todo o projeto, independente do nível de abstração em que se encontra [41].

A Figura 4.4(a) mostra a metodologia tradicional, onde após o Modelo C/C++ ser analisado, testado e refinado, é migrado para HDL. Percebe-se que esta metodologia se torna limitada, pois:

- a conversão manual pode resultar em erros, além de ser tediosa;
- o modelo HDL é desconectado do modelo C/C++, e uma modificação, após a conversão, não será implementada em C/C++;
- novos testes precisam ser elaborados após a conversão, pois, normalmente, os realizados para validar o modelo C/C++ não executam em HDL.

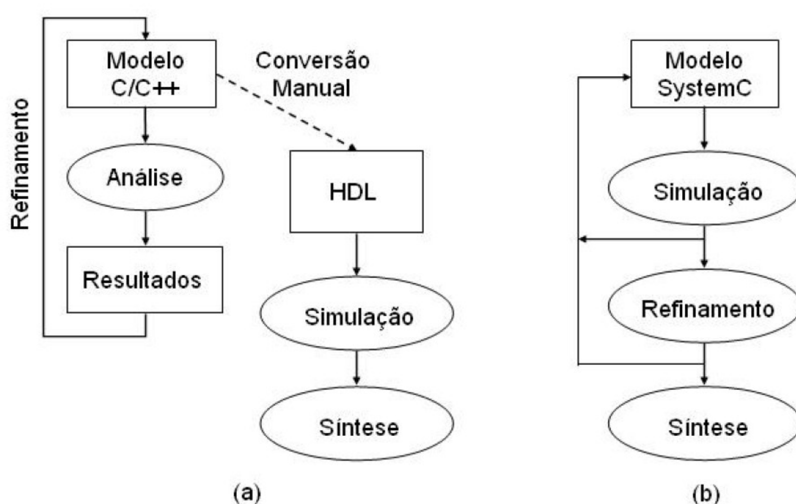

Figura 4.4: Duas metodologias de projeto. (a) metodologia tradicional; (b) metodologia SystemC

A metodologia SystemC, representada pela Figura 4.4(b), foi a proposta para superar os limites da tradicional, tendo como principais vantagens:

- o refinamento gradual, em pequenos passos, facilitando adaptações no projeto e detecção de erros durante o processo;
- a utilização de uma mesma linguagem durante todo o processo, não necessitando de profissionais com conhecimentos profundos em diversas linguagens em diferentes níveis de abstração; e
- utilizar os mesmos testes durante o processo, como já citado.

Capítulo 5
A ferramenta VIPRO-MP

VIPRO-MP (*Virtual Prototype for Multiprocessor Architectures*) é o ambiente proposto neste trabalho para construção de plataformas virtuais multiprocessadas. O SimpleScalar foi utilizado como simulador do processador, junto com o *framework* para cálculo de potência consumida Wattch (Sim-Wattch), sendo os demais componentes de *hardware* modelados utilizando SystemC.

O modelo arquitetural implementado no VIPRO-MP permite a execução de aplicações descritas em linguagem C e compiladas para conjunto de instrução PISA, semelhante à arquitetura MIPS. A escolha desta arquitetura se justifica por ser uma empresa que vende *cores*, ou seja descrições de processadores que podem ser configurados de acordo com a necessidade do cliente e por sua crescente utilização nos sistemas embarcados atuais, como DVD Recorders, Blue-Ray Decoders, câmeras digitais Canon, Samsung, JVC, Pentax e Fujifilm, e na maioria dos produtos VoIP existentes (cerca de 70%) [35].

Cada processador possui uma memória privada limitada em 2Gb, e a comunicação entre eles é feita apenas pela memória compartilhada, que é mapeada no endereço base 0x80000000 e endereço final 0x8FFFFFFF. O barramento e a memória compartilhada são interligados utilizando canais de transferência no nível de transações (TLM - *Transaction-Level Messages*) e estão modelados em SystemC. Na versão atual do VIPRO-MP os dados da memória compartilhada não são armazenados em *cache*, evitando-se assim a implementação de protocolos de coerência. Tal modelo, em que a memória privada utiliza a *cache* e a memória compartilhada não a utiliza, é comumente adotado em arquiteturas embarcadas tais como o Nexperia [17], ou mesmo o processador Cell [19] em relação a memória dos SPE (*Synergistic Processor Element*). O VIPRO-MP também possui mecanismos de interrupção, permitindo que um único processador execute mais de uma tarefa. Para isso, um *timer* foi

implementado, no qual a cada período interrompe o processador que assume a tarefa pendente alojada no manipulador de interrupções.

Os parâmetros de configuração, para cada processador, continuam sendo os mesmo do *sim-outorder* do pacote SimpleScalar (Apêndice A), facilitando para quem já trabalha com a ferramenta. Porém, para aumentar a flexibilidade da plataforma, outros parâmetros foram inseridos: (i) tecnologia CMOS utilizada em cada processador, (ii) ativação do modo *debug* do barramento, (iii) a latência, em ciclos, da memória compartilhada e (iv) a definição do endereço base e final da memória compartilhada, caso o projetista deseje alterar.

O VIPRO-MP é uma ferramenta de código aberto e respeita os termos das licenças das ferramentas que serviram como base para o seu desenvolvimento.

5.1 Desenvolvimento

O desenvolvimento passou por estágios bem definidos, em que versões de desenvolvimento foram construídas, testadas e evoluídas até atingir maturidade suficiente para a distribuição da versão final.

A primeira versão *dev* (versão de desenvolvimento) do VIPRO-MP foi obtida migrando todo o código originalmente em C do Sim-Wattch (*sim-outorder* + *Wattch*) para C++, resultando em uma classe para cada componente do processador, e por conseqüência, melhorando a organização do código. O *framework* Wattch foi encapsulado em um objeto denominado **Power**, e nesta versão, a biblioteca CACTI 1.0 foi fundida com este objeto, fazendo com que as funções CACTI virassem métodos de Power, como mostra a Figura 5.1. Isto trouxe complicações, pois estudos futuros mostraram que esta biblioteca é independente, e ao ser substituída, neste modelo, uma parte do código do VIPRO-MP, deveria ser alterado.

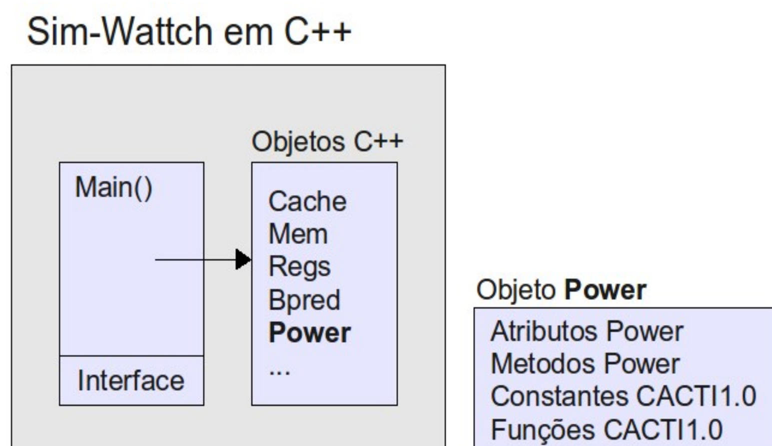

Figura 5.1: Sim-Wattch com a biblioteca CACTI 1.0 integrada

Durante a fase de validação desta primeira versão *dev*, a mesma foi experimentada exaustivamente e os resultados das simulações foram comparados com aqueles obtidos nas ferramentas originais de base, garantindo que erros não fossem inseridos no processo de modificação.

A segunda versão *dev* permitiu que testes iniciais utilizando uma arquitetura multiprocessada fosse realizado, contando com uma memória compartilhada de 16Kb e o processador já possuindo integração com o SystemC, permitindo ao projetista instanciar quantos processadores desejasse na plataforma. Devido às mudanças internas no código do processador (migração para C++ e uso do SystemC), denominamos esta nova versão de SS++. Quando um SS++ (processador) solicita a memória compartilhada, neste modelo, a *cache* não interfere, evitando a implementação de mecanismos de coerência.

Para validar esta versão, foi executado vários binários idênticos em um ambiente com vários processadores e comparado os resultados de desempenho e consumo de potência com o Sim-Wattch original; o sincronismo e a comunicação entre os processadores também foi cuidadosamente analisado. Porém, o barramento não era explicitamente modelado, e a memória compartilhada não era implementada no nível transacional (TLM).

Após estas implementações iniciais (versões *dev*), o projeto foi evoluindo e se solidificando, e após intensa codificação, chegou-se à versão final, que será descrita nas seções subseqüentes.

5.1.1 Metodologia

Como o VIPRO-MP se apóia em várias ferramentas, estudá-las detalhadamente foi fundamental antes de começar as implementações. A princípio, as ferramentas SimpleScalar e Sim-Wattch foram exploradas e validadas, e para isso, um *cross-compiler* GCC, de versão 2.7.2.3, foi compilado para gerar binários ECOFF (MIPS).

As versões de desenvolvimento, citadas na seção anterior, foram construídas para validar o projeto por etapas, até chegar, por fim, na versão final. Todas as versões foram desenvolvidas para o ambiente GNU/Linux sobre a IDE de programação Eclipse CDT [10], que facilitou em muito o mapeamento e rastreamento das estruturas complexas do código, aumentando a produtividade. Os compiladores utilizados foram o GCC-3.4 e G++-3.4, e para gerar o VIPRO-MP, um Makefile foi construído para atender os requisitos da ferramenta. Os dois compiladores foram utilizados pois a biblioteca CACTI, na versão final, foi mantida em C, como será explicado na próxima seção. O SystemC utilizado foi a versão 2.0.1 com um

aumento no tamanho da pilha das *threads,* pois o tamanho padrão era insuficiente para executar as *threads* dos processadores.

5.1.2 Atualização da biblioteca CACTI

A biblioteca CACTI 1.0 baseia-se apenas no tempo de acesso às estruturas para estimar o consumo de potência, como já mencionado na Seção 4.2. Outro agravante é o fato da tecnologia ser fixa dentro da biblioteca, e mesmo o Sim-Wattch permitindo a alteração da tecnologia de fabricação, a CACTI 1.0 ignora tais parâmetros[1]. Devido a estas restrições, surgiu a necessidade de atualizar a biblioteca CACTI para a versão 4.1, objetivando maior precisão na estimativa do consumo de potência.

Porém, converter todas as funções da versão 4.1 de C para C++, como foi feito com a versão 1.0, e mesclá-las ao objeto Power, é custoso, e o trabalho é perdido caso uma nova versão da biblioteca fosse escolhida para integrar o VIPRO-MP. Por isso, a abordagem adotada foi manter esta biblioteca nativamente em C, e assim, substituí-la por outra versão simplesmente impactaria na atualização das chamadas das funções na classe Power, caso alguma assinatura se modifique.

Na CACTI 4.1, toda interação com a biblioteca ocorre por meio da função *cacti_interface()*, em que vários parâmetros novos são considerados e utilizados dentro da biblioteca para calcular dinamicamente valores que na CACTI 1.0 eram estáticos. Para validação desta nova biblioteca, comparou-se *benchmarks* executados no VIPRO-MP e na ferramenta PowerSMT [16], que também realiza esta atualização e foi cuidadosamente validada. Como a segunda ferramenta é um simulador SMT, considerou-se apenas os resultados onde um único fluxo de instrução é utilizado no processador.

5.1.3 A modelagem SystemC

A primeira integração do VIPRO-MP com o SystemC aconteceu na segunda versão *dev*, no qual apenas o processador foi modelado. Na versão final, além do processador, o barramento, a memória compartilhada, o árbitro do barramento e o *timer* também estão modelados em SystemC (Figura 5.3, página 40), permitindo uma integração lógica entre os componentes, assemelhando-se a blocos de *hardware*.

---

[1] Esta observação também está presente em [16], na descrição da ferramenta PowerSMT.

Após a instanciação dos módulos SystemC, pode-se optar por sincronizá-los por um sinal de relógio (*clock*), facilitando a construção do *hardware* posteriormente. O Quadro 5.1 exibe o código onde o *clock* é instanciado e ligado, ao barramento, à memória compartilhada e ao *timer*, respectivamente.

Quadro 5.1: Criação do *clock* e sua ligação com os componentes

```
/* instanciando o clock da simulacao */
sc_clock clock("CLOCK", 10, 0.5, 0.0);

/* ligando o sinal de clock na porta de clock dos componentes */
bus->clock(clock);
mshared->clock(clock);
it_timer->clock(clock);
```

Em cada ciclo, os componentes executam um método definido ou uma interação de um laço, utilizando a função *wait()* para sincronização. O Quadro 5.2 possui o pseudocódigo de um processador, implementado no arquivo **sim-outorder.cpp**, onde a cada ciclo: as variáveis de consumo de potência são iniciadas, todos os estágios do *pipeline* são executados e os valores das variáveis de consumo de potência no objeto Power são atualizados.

Quadro 5.2: Pseudocódigo do laço principal do processador

```
for (;;)
{
    /* safe state for interrupt */
    if ((interrupt)&&(processorSafe)){
           executeInterruptHandler()
    }

    /* Added for Wattch to clear hardware access counters */
    POWER->clear_access_stats();

    /* main simulator loop, NOTE: the pipe stages are traverse in reverse order
       to eliminate this/next state synchronization and relaxation problems */
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();

    /* Added by Wattch to update per-cycle power statistics */
    POWER->update_power_stats();

    /* wait for next cycle (SystemC) */
    wait();
}
```

Logo, os processadores do ambiente também são ligados ao *clock,* porém para melhor organização, isto é feito no momento em que eles são instanciados, como mostra o Quadro 5.3. Neste quadro, além do *clock,* é possível observar que o processador é ligado a outros

componentes, como o barramento e o *timer*. Nestas ligações, o SystemC adota o seguinte critério:

```
master.port(slave);
```

onde *master* é o componente que terá o controle da porta, e o *slave*, ao receber um sinal nesta porta, executará um método pré-definido. Exemplificando, o *timer* é quem comandará a porta *interrupt_port*, que o conecta ao processador. Ao receber um sinal nesta porta, o processador executa o método *setInterruptMode(),* como será explicado na Seção 5.1.5, que descreve o tratamento de interrupções.

Quadro 5.3: Instância de dois processadores

```
/* Processador */
simplescalar scsp("PROC1", 1);
scsp.setFd(fd);                     // atribuindo o arquivo output
scsp.CLK(clock);                    // ligando o sinal de clock no procesador
scsp.bus_port(*bus);                // ligando a porta do barramento com o barramento
it_timer->interrupt_port(scsp);     // ligando a porta interrupt no timer
scsp.init(argc1, argv1, environ);  // passando os parametros de PROC1
scsp.reset(reset);

/* Processador */
simplescalar scsp2("PROC2", 2);
scsp2.setFd(fd);
scsp2.CLK(clock);
scsp2.bus_port(*bus);
it_timer->interrupt_port(scsp2);
scsp2.init(argc2, argv2, environ);
scsp2.reset(reset);
```

Outras ligações que precisam ser feitas, como o árbitro com o barramento e a memória compartilhada com o barramento, são apresentadas em detalhes no Apêndice C, que apresenta na íntegra o arquivo **main_sc.cpp** do VIPRO-MP[2].

5.1.4 Barramento e memória compartilhada

A comunicação entre os processadores é feita pela memória compartilhada. Para acessá-la, os processadores dispõem de um barramento modelado em nível transacional que, a cada ciclo, verifica uma estrutura que armazena as requisições de acesso ao barramento.

Uma requisição pode operar em quatro modos: REQUEST, WAIT, ERROR e OK; o primeiro modo é atribuído inicialmente à qualquer requisição de barramento. Como várias requisições podem chegar simultaneamente, um árbitro é necessário para decidir qual assumirá. Para essa escolha, o árbitro possui dois critérios de desempate: (i) elege as requisições em modo WAIT antes das requisições em modo REQUEST e (ii) sempre opta

[2] Ao utilizar SystemC, o arquivo principal *main.cpp* é substituído por *main_sc.cpp*.

pela requisição oriunda do processador com menor ID. O critério (i) possui maior precedência que o (ii). Após essa eleição, o componente *slave* que estiver mapeado no endereço informado, é acionado. Para requisitar a memória compartilhada, o endereço deve estar entre 0x80000000 e 0x8FFFFFFF, como já citado.

Porém, ao instanciar a memória compartilhada, alguns de seus parâmetros podem ser configurados, como o endereço base e final e a latência (em ciclos), como exibe o Quadro 5.4. Na ausência de um quarto parâmetro, a latência padrão é 18 ciclos.

Quadro 5.4: Instância da memória compartilhada

```
memshared   *mshared  = new memshared ("mem", /*base*/ 0x80000000,
                                              /*final*/ 0x8fffffff,
                                              /*latency*/ 25);
```

Ao repassar a requisição à memória compartilhada, esta requisição fica em modo WAIT na estrutura do barramento até a memória compartilhada sinalizar sucesso ou falha. O modo WAIT é necessário pois normalmente a memória compartilhada não consegue completar a operação em um ciclo devido à alta latência ou pela limitação no número de linhas do barramento (32 linhas), necessitando, por exemplo, de dois ciclos para transferir um *double* (64 bits). Isso explica a maior precedência do WAIT sobre o REQUEST. Caso algum erro ocorra, a requisição fica em modo ERROR e é descartada pelo barramento. Se a latência atribuída já foi consumida e a operação obteve sucesso, a requisição se torna OK, e é retirada da fila de requisições do barramento. O diagrama de estado da Figura 5.2 mostra uma *thread* processador e uma *thread* barramento acessando a memória compartilhada. Como já foi dito, quando o processador faz uma requisição de barramento, esta é armazenada na estrutura.

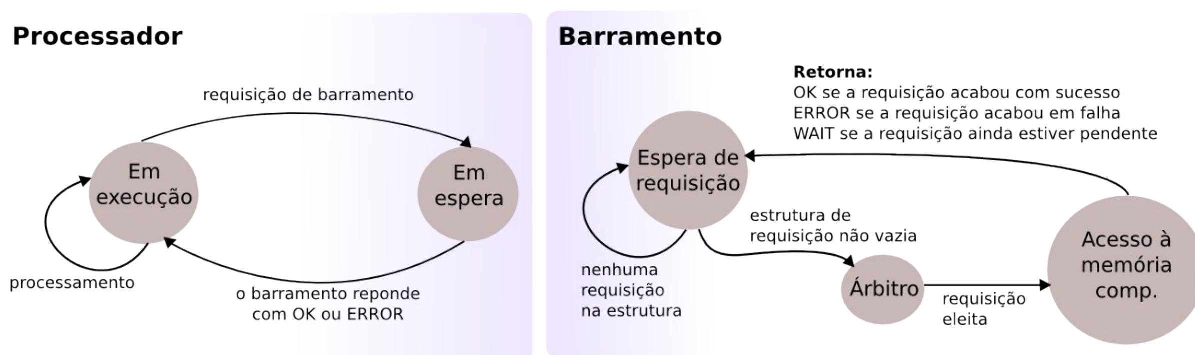

Figura 5.2: Diagrama de estado para acessar à memória compartilhada

Percebe-se que, quanto maior a latência da memória compartilhada, mais ciclos o barramento demorará para permitir outra requisição, resultando em uma grande contenção. A fim de analisar essa característica, três variáveis foram acrescentadas no arquivo final de estimativa de desempenho, mostradas no Quadro 5.5. Utilizando estes dados como exemplo, percebe-se que o barramento foi acessado 129587 vezes e o número total de ciclos que o processador esperou foi 411030. Deste total, 22269 ciclos foram desperdiçados pela contenção.

Quadro 5.5: Variáveis sobre a contenção do barramento de um processador

```
sim_cycle          4399829    # total simulation time in cycles
num_bus_access      129587    # total number of access bus
cycle_wait_bus      411030    # total cycle waiting for bus
cycle_bus_busy       22269    # total cycle waiting (wasted) for bus busy
```

5.1.5 Interrupção

Possibilitar a execução de duas tarefas independentes em um único processador permite que aplicações e arquiteturas mais complexas possam ser utilizadas futuramente. Por isso, o VIPRO-MP permite que seus processadores sejam interrompidos pelo *timer*, um componente modelado em SystemC que, a cada período de tempo, sinaliza existência de uma interrupção através da porta de interrupção existente em cada processador.

Como os processadores modelados são superescalares com instruções fora de ordem, cuidados especiais foram necessários para que o suporte a interrupção fosse implementado. Voltando ao Quadro 5.2, da página 34, percebe-se que, mesmo com o modo **interrupt** ativo, é necessário que o processador atinja um estado seguro para chamar o método *execute InterruptHandler()*. Este estado seguro é alcançado quando todas as instruções que já estavam no estágio de busca e despachadas no momento da interrupção terminaram de executar. Para isso, duas estruturas foram estudadas: a LSQ (*Load/Store Queue*) e a RUU (*Register Update Unit*). Juntas, elas formam a estrutura da estação de reserva do SimpleScalar, espaço onde as instruções aguardam pela disponibilidade dos recursos necessários ou resolução das dependências para, enfim, executar e finalizar a execução da instrução (estágio *commit()*). Assim, quando o modo *interrupt* é acionado, o estágio de busca (*ruu_fetch()*, Quadro 5.2) interrompe a busca por novas instruções e começa a inserir a instrução NOP (*no operation*) para os estágios seguintes. Isso é realizado até as estruturas LSQ e RUU ficarem vazias, ou seja, até o processador alcançar um estado seguro.

Armazenar qual seria a próxima instrução à executar desta tarefa interrompida e definir qual instrução executar imediatamente está codificado no Quadro 5.6. Como os registradores R26 e R27 do *mips* são reservados para o sistema operacional, é no R26 que foi armazenado o *NextPC* da tarefa interrompida, ou seja, o endereço da próxima instrução a ser executada quando voltar da interrupção. O salvamento do contexto e o *jump* para o *interruptHandler* são feitos por meio de *software*, implementado em *assembler.* No VIPRO-MP foi definido que a área de código desse *software* tem como endereço base 0x1100.

Quadro 5.6: Tratamento da interrupção no processador

```
regs0.regs_R[26] = regs0.regs_NPC;
regs0.regs_NPC   = 0x1100;
printf("interrupt actived\n");
```

O Quadro 5.7 exibe a parte do código *assembler* responsável por copiar os registradores para a pilha (da linha 15 à 42) e por chamar a função na qual o usuário implementará a nova tarefa (linha 45). Essa implementação deve estar em uma função C, denominada **__it_handle()**. Para a criação do binário executável, portanto, deve-se ligar o arquivo objeto contendo a função *main()* e a função *__it_handle()* com o arquivo objeto do *assembler.*

Quadro 5.7: Salvando o contexto e saltando para a função implementável

```
15.    sw $1, 0-4($29)
16.    sw $2, 0-8($29)
17.    sw $3, 0-12($29)
18.    sw $4, 0-16($29)
19.    sw $5, 0-20($29)
20.    sw $6, 0-24($29)
21.    sw $7, 0-28($29)
22.    sw $8, 0-32($29)
23.    sw $9, 0-36($29)
24.    sw $10, 0-40($29)
25.    sw $11, 0-44($29)
26.    sw $12, 0-48($29)
27.    sw $13, 0-52($29)
28.    sw $14, 0-56($29)
29.    sw $15, 0-60($29)
30.    sw $16, 0-64($29)
31.    sw $17, 0-68($29)
32.    sw $18, 0-72($29)
33.    sw $19, 0-76($29)
34.    sw $20, 0-80($29)
35.    sw $21, 0-84($29)
36.    sw $22, 0-88($29)
37.    sw $23, 0-92($29)
38.    sw $24, 0-96($29)
39.    sw $25, 0-100($29)
40.    sw $29, 0-104($29)
41.    sw $30, 0-108($29)
42.    sw $31,0-112($29)
43.    addi $29, $29, -112
44.    addi $4, $0, 0x2
45.    jal     __it_handle
46.    add $0, $0, $0
```

Após executar a função *__it_handle(),* a tarefa interrompida será retomada. Mas para isso, o contexto é restaurado e o registrador *NextPC* recebe o conteúdo do registrador R26. Esta etapa também é implementada em *software* e seu código *assembler* segue o exposto no Quadro 5.7.

Caso o projetista deseje organizar uma arquitetura sem interrupção nos processadores, é necessário apenas comentar as linhas em que o componente *timer* aparece no arquivo *main_sc.cpp* (Apêndice C), ou simplesmente não conectar a porta de interrupção dos processadores no *timer.*

5.2 Organização final da ferramenta

Para relacionar os conceitos apresentados, um diagrama de blocos do VIPRO-MP é apresentado na Figura 5.3, no qual um ambiente com dois processadores (SS++) está modelado. As linhas contínuas e o sinal de *clock* são ligações de portas em SystemC. As linhas pontilhadas são ligações funcionais, em que o objeto Power solicita funções da biblioteca CACTI 4.1.

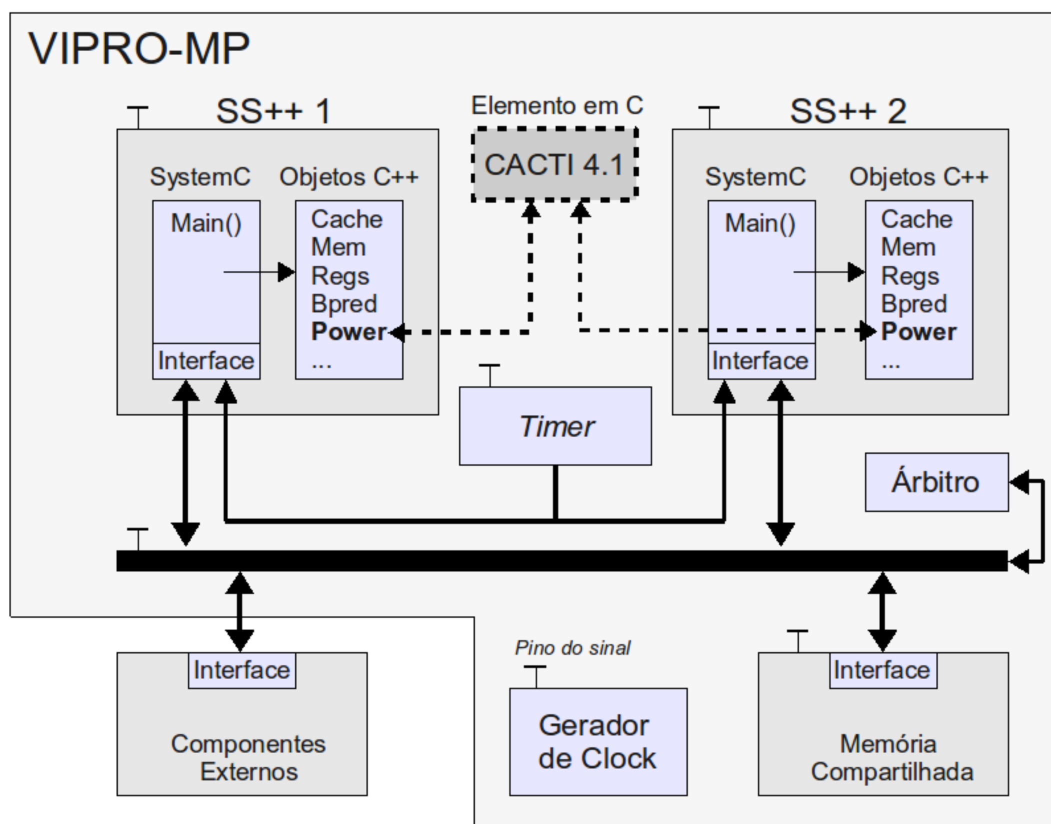

Figura 5.3: Organização do VIPRO-MP com dois processadores e *timer*

Capítulo 6
Simulações e Resultados

Este capítulo tem por objetivo mostrar alguns testes realizados com a plataforma virtual VIPRO-MP com o intuito de avaliar algumas características que a ferramenta oferece. Três simulações foram propostas, na qual a primeira explora a contenção no barramento por meio de um algoritmo de ordenação paralelo [13], a segunda avalia o uso de *caches* na execução de um algoritmo de compressão JPEG paralela, e a terceira explora o espaço de projeto com múltiplos processadores fazendo uso, também, do algoritmo de compressão JPEG paralelo [13].

6.1 Estudo de Caso – *Ordenação paralela*

Um problema conhecido, quando vários processadores estão executando um certo algoritmo, é a sobrecarga no barramento. Isto ocorre porque trocas de mensagens para sincronização ou compartilhamento de dados entre os processadores devem ser realizados e, para isso, a memória compartilhada é utilizada por meio do barramento. Quando existem requisições em demasia para adquiri-lo, muitos processadores esperam vários ciclos para, enfim, conseguir transferir seu dado, o que é denominado de contenção.

Nesta simulação, a contenção do barramento foi analisada sob diferentes arquiteturas e para isso, foi alterado o número de instâncias de processadores na plataforma VIPRO-MP. O impacto da latência da memória compartilhada também foi analisada com os seguintes parâmetros 3, 6 e 18 ciclos. Os valores de latência foram escolhidos pois são iguais as latências da memória *cache* L1 (3 ciclos), memória *cache* L2 (6 ciclos) e memória principal (18 ciclos).

A aplicação paralela é responsável por ordenar um vetor de 4000 elementos gerados de forma randômica. Para isso, um processador é eleito mestre e carrega o vetor na memória

compartilhada, para enfim, sinalizar aos processadores escravos que a ordenação pode começar. Cada processador fica encarregado de ordenar, pelo método *QuickSort,* uma faixa de endereço. Ao perceber que todos os escravos terminaram, o processador mestre realiza o *MergeSort* para junção e geração do vetor final [13].

6.1.1 Resultados

A tabela abaixo mostra o número de ciclos que cada processador consumiu para realizar a ordenação. O mestre é sempre o processador com maior ID, possuindo, obviamente, mais ciclos de execução.

Tabela 6.1: Impacto da latência da memória compartilhada no número de ciclos

Latência Memória Compartilhada	Quantidade de Processadores	Número de Ciclos			
		Proc 1	*Proc 2*	*Proc 3*	*Proc 4*
3 ciclos	1 Proc	2393815			
	2 Proc	1087806	1273829		
	4 Proc	532631	544900	639068	823553
6 ciclos	1 Proc	3031909			
	2 Proc	1422677	1656787		
	4 Proc	750407	936557	1054714	1287216
18 ciclos	1 Proc	5584416			
	2 Proc	3136674	3562594		
	4 Proc	1716979	3349584	3566959	3991504

Visto que os dados da memória compartilhada não são armazenados em *cache,* cada acesso tem um *overhead* no resultado final. Como exemplo, a execução com 4 processadores e memória compartilhada de 18 ciclos de latência é 350% mais lento que em uma com latência igual a 3.

Outro resultado importante é o fato de dois processadores com memória compartilhada de 3 ciclos executar em um tempo muito próximo a quatro processadores com memória compartilhada de 6 ciclos (sendo 1273829 e 1287216 ciclos, respectivamente). Tal resultado indica para o projetista que, neste problema, o objetivo do projeto poderia ser fixado em melhorar a latência ao invés de aumentar a quantidade de processadores.

6.2 Estudo de Caso - *Algoritmo JPEG*

O algoritmo de compressão JPEG, adaptado do *benchmark* MiBench [34], foi utilizado neste estudo de caso. Como inicialmente o algoritmo era seqüencial, foi necessário estudar o funcionamento deste método de compressão para paralelizá-lo corretamente. O algoritmo é implementado em duas fases conforme apresentado na Figura 6.1. Na primeira fase, após o carregamento da imagem a ser comprimida na memória compartilhada, feito pelo processador mestre, cada processador do ambiente fica incumbido de realizar a transformada DCT (*Discrete Cosine Transform*) de N/P blocos JPEG e reescrever na memória compartilhada (N é o número total de blocos JPEG e P o número de processadores). Quando todos os processadores terminam a primeira fase, o mestre executa a fase final e seqüencial do algoritmo, a compressão (*entropy encoder*), e gera o arquivo de saída. Para a sincronização entre os processadores variáveis de espera são utilizadas na memória compartilhada, sendo que os processadores utilizam a espera ativa.

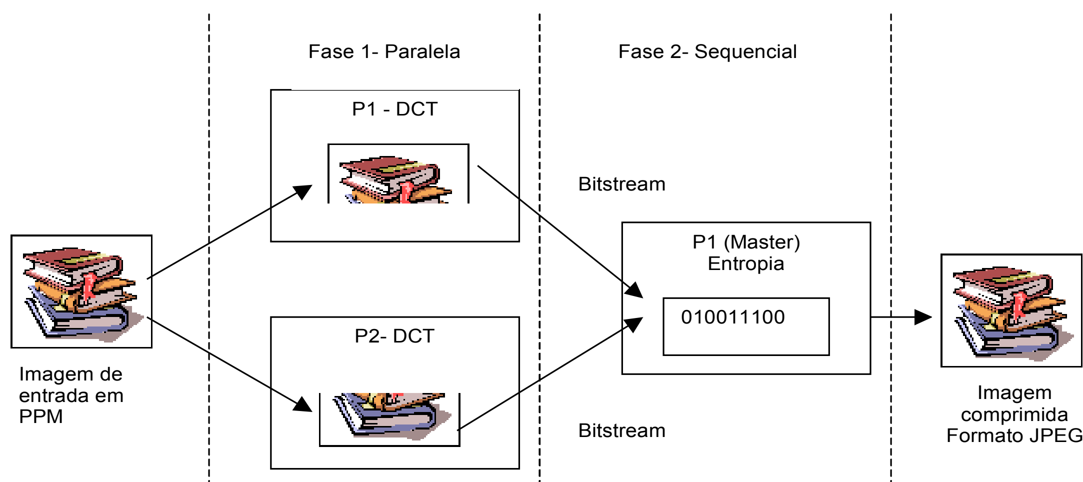

Figura 6.1: Algoritmo de compressão JPEG paralelizado

6.2.1 Exploração do espaço de projeto nas caches

Como foi visto na Seção 3.2.2, na Figura 3.6, a memória *cache* é responsável por grande parte do consumo de um processador. Por isso, esta simulação avalia o impacto que o aumento da memória *cache* traz na quantidade de ciclos e na potência média consumida. Foi utilizado um algoritmo de compressão JPEG [34], muito utilizado em câmeras fotográficas e celulares.

Duas instâncias do processador foram utilizadas e as *caches Instruction* L1 (IL1) e *Data* L1 (DL1) foram manipuladas de modo independente, permitindo uma análise de ambas quanto ao desempenho da aplicação. A variação das *caches* foi apenas em relação ao seu tamanho (quantidades de entradas), fixando a associação em 4 *Ways,* para DL1, e 1 *Way* (mapeamento direto), para IL1; o tamanho do bloco de ambas foi de 32 bytes. O tamanho da *cache* IL1 variou de 4 à 256Kbytes e da *cache* DL1, de 4 à 1024Kbytes, e as variáveis resultantes analisadas foram o número de ciclo para execução do aplicativo e a potência média dissipada por ciclo. A tecnologia utilizada foi a de 350nm, sendo a configuração padrão do Sim-Wattch. Apesar de ser ultrapassada, os resultados obtidos são úteis para uma comparação relativa entre as diferentes soluções.

É importante frisar que as configurações da cache podem ser informadas diretamente na linha de comando ou no arquivo de configurações (Apêndice A), portanto não é necessário gerar um novo simulador para cada configuração da arquitetura, mas somente informar a configuração e coletar os resultados. Para auxiliar o projetista, uma ferramenta auxiliar foi implementada em Matlab [33] para coletar um conjunto de arquivos de resultados provenientes das simulações e construir os gráficos para a análise das diferentes arquiteturas, permitindo que se informe quais parâmetros dos resultados serão utilizados, como o número de ciclos da simulação (*sim_cycle*) e a potência média consumida por ciclo (*avg_total_power_cycle_cc3*).

6.2.2 Resultados

Os gráficos da Figura 6.2 mostram os resultados em termos de tamanho total da *cache*, contabilizando-se todos os processadores. Por exemplo, no eixo IL1, 32 significa que cada processador tem 16Kbytes de *cache* de instruções (2x16Kbytes). Na Figura 6.2(a), para cada configuração da *cache*, o número de ciclos mais alto entre os dois processadores foi utilizado; em 6.2(b), a potência média dissipada é a soma da potência média dissipada dos dois processadores.

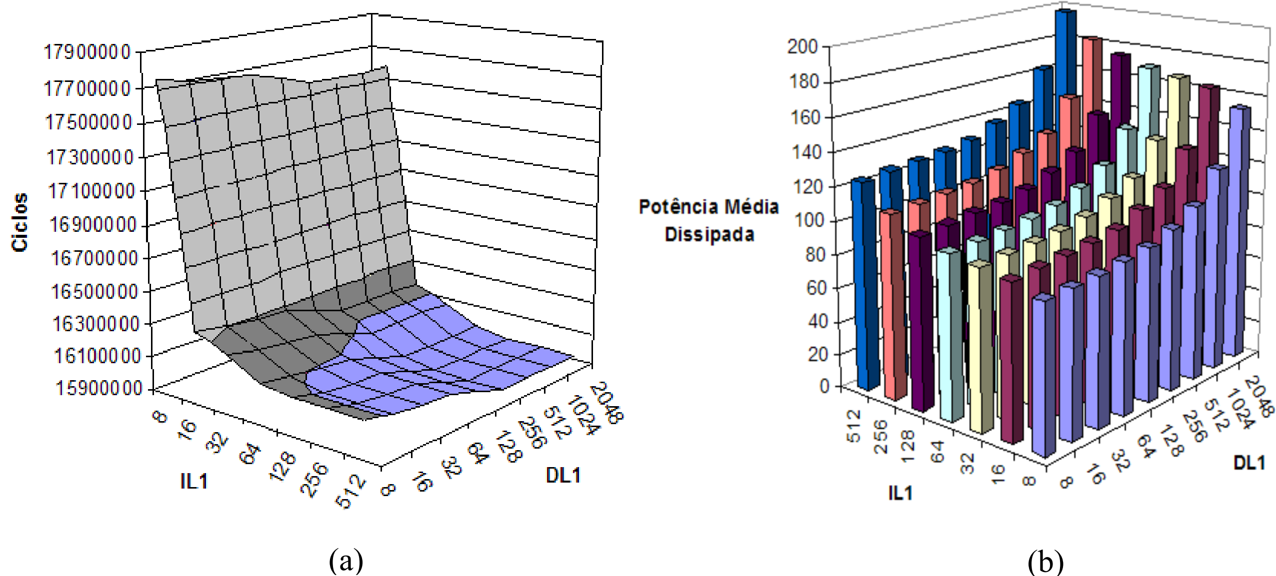

Figura 6.2: Gráfico com a variação dos tamanhos das *caches*. (a) *versus* os ciclos de *clock;*
(b) *versus* a Potência Dissipada

Observa-se que, devido ao tipo de problema resolvido, a *cache* de dados influencia minimamente na diminuição da quantidade de ciclos, pois repetições de dados não são freqüentes, porém dissipa uma quantidade de potência considerável. Por isso, foi fixado um valor de 256Kbytes (128Kbytes em cada processador) na *cache* DL1 e comparou-se a potência dissipada e os ciclos, com relação à *cache* IL1 (Figura 6.3). Nota-se um ponto ótimo, com a *cache* de instruções (IL1) de tamanho 16Kbytes (2x8Kbytes). Neste ponto, o número de ciclos é de 16.146.453 e a potência média é 103,6 *watts*. Comparando-se com a solução que utiliza a *cache* de instruções em 512Kbytes (2x256Kbytes), verifica-se um ganho de 1,3% no desempenho, porém com um aumento de 39% na potência consumida.

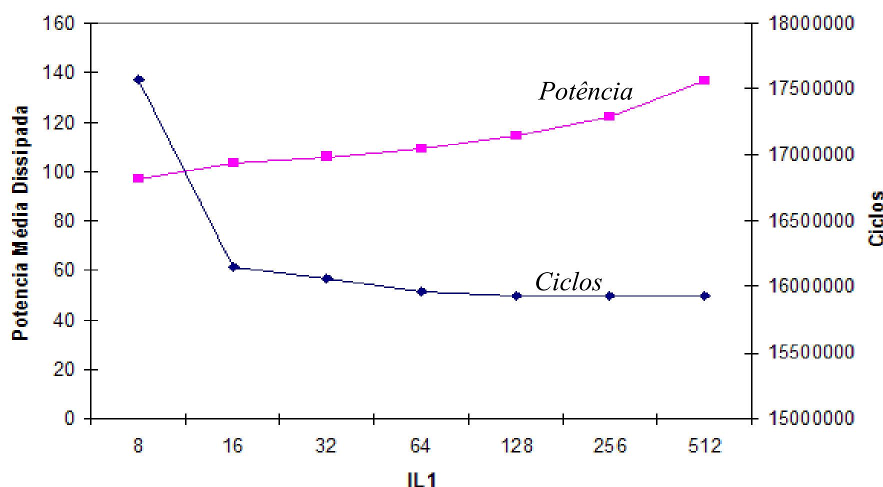

Figura 6.3: Gráfico com variação de IL1 e DL1 fixada em 2x128Kb

6.2.3 Exploração do espaço de projeto com múltiplos processadores

Nesta simulação, o objetivo principal é comparar o ganho em eficiência energética quando múltiplos núcleos são utilizados. O algoritmo utilizado foi a compressão JPEG paralela, explicado na Seção 6.2.

Foi utilizado um núcleo a 1.200MHz, dois à 600Hz, e 4 à 300Hz, e a tensão foi reduzida linearmente com a freqüência, fato válido para circuitos CMOS (*Complementary Metal-Oxide-Semiconductor)*, conforme foi apresentado na Seção 3.2.2. O tamanho total das *caches* de dados e de instruções no ambiente foi mantido em 512Kb e 256Kb, respectivamente. Assim, o aumento na quantidade de núcleos no ambiente é inversamente proporcional ao tamanho das *caches* em cada núcleo, como pode ser visto na Tabela 6.2, em configuração.

Outro aspecto importante é o monitoramento do barramento, no qual se verifica a quantidade de ciclos desperdiçados por um processador esperando ganhar a vez para acessar a memória compartilhada. A latência de acesso padrão do SimpleScalar para a memória privada é de 18 ciclos, e por isso, o acesso à memória compartilhada foi configurado também com esta latência.

6.2.4 Resultados

A Tabela 6.2, exibe a configuração de cada ambiente e a estimativa de desempenho e consumo de potência da execução. A contenção no barramento indica a quantidade de ciclos que um processador ficou esperando pelo acesso, além dos 18 ciclos usuais. O processador mestre é sempre o primeiro (**#1**).

Tabela 6.2: Dados de configuração e execução de ambientes com um, dois e quatro núcleos

		1 Núcleo	2 Núcleos		4 Núcleos			
Configuração			**#1**	**#2**	**#1**	**#2**	**#3**	**#4**
Frequência	Mhz	1.200	600	600	300	300	300	300
Escala de Voltagem	-	1	0,5	0,5	0,25	0,25	0,25	0,25
Cache de Dados L1	Kb	512	256	256	128	128	128	128
Cache de Instrução L1	Kb	256	128	128	64	64	64	64
Execução								
Número de Ciclos	-	16.130.586	10.903.904	7.605.377	10.303.912	4.891.482	5.987.316	6.989.351
Contenção no Barramento	ciclos	0	558.098	591.943	1.081.213	558.547	1.654.657	2.659.138
Potência Media por Ciclo	watt	611,2744	53,3414	70,3985	6,0182	8,4541	6,9062	5,9139
Tempo de Execução	seg.	0,0134	0,0182		0,0343			
Energia	w*t	8,1910	2,2520		0,9361			

Devido a necessidade da execução da fase seqüencial, o mestre sempre terá o maior número de ciclos, e portanto, este número foi utilizado para calcular o tempo de execução dado a freqüência. Percebe-se que a solução de 2 Núcleos, que trabalha a 600MHz cada, o seu tempo de execução foi aproximadamente 35% maior que a solução com 1 núcleo. Porém, a potência média por ciclo na solução com 1 processador é 611,2744 *watts*, enquanto que para a solução com 2 processadores a soma da potência média é 123,7399, o que traz uma redução de 80%. Já a solução com 4 núcleos apresentou uma grande contenção no barramento, e o fato dos processadores com ID maiores possuir um desperdício maior de ciclos, se deve ao árbitro do barramento, que seleciona o menor ID como critério de desempate. Apesar de seu tempo de execução ter sido 150% maior que a solução de um núcleo, sua economia energética foi cerca de 8 vezes à de um núcleo e 2,5 vezes à de 2 núcleos, se mostrando uma opção interessante quando a ênfase do projeto está na economia de potência dissipada.

Capítulo 7
Conclusões e Trabalhos Futuros

A crescente preocupação dos projetistas em construir sistemas embarcados com consumo eficiente de potência tem feito, das plataformas virtuais, ferramentas indispensáveis para exploração do espaço de projeto. Sua rápida configuração associada aos resultados confiáveis permite que se teste, sem esforço, arquiteturas de diversos tipos, avaliando o desempenho e o consumo de potência do protótipo.

A ferramenta VIPRO-MP encontra-se na gama de plataformas virtuais para sistemas multiprocessados, permitindo que diferentes modelos arquiteturais sejam avaliados rapidamente, possibilitando a maximização da relação custo/benefício. Isto porque dado um aplicativo, escolher uma configuração de *hardware* que seja adequada em relação aos requisitos de desempenho, consumo de potência e custos para o projeto pode ser uma tarefa não trivial.

Nas simulações realizadas, várias arquiteturas foram analisadas, permitindo uma visualização sumária da capacidade de exploração de projeto do VIPRO-MP. Como o VIPRO-MP é baseado no SimpleScalar, todos os parâmetros podem ser configurados diretamente através de arquivos de configuração ou linha de comando, possibilitando uma rápida exploração de diferentes arquiteturas.

Nos estudos de caso, o projetista pôde avaliar o impacto de diferentes componentes da arquitetura no desempenho geral de uma aplicação. Por exemplo, no caso da aplicação JPEG notou-se claramente que após um determinado limite, aumentar a *cache*, dependendo da aplicação, não garante melhor desempenho, mas tem um impacto importante no consumo de potência, o que, se não visto com antecedência, pode arruinar o projeto. Com relação à diminuição da freqüência e da tensão com múltiplos núcleos, demonstrou-se a vantagem da arquitetura multiprocessada em relação a um único núcleo, alcançando baixo consumo de potência sem grandes perdas no desempenho. A contenção do barramento também é um fator

decisivo para o desempenho, e isto pode ser visto com antecedência no VIPRO-MP. Desta forma, pode-se, para uma dada aplicação, escolher uma organização eficiente, balanceando poder de processamento com consumo de potência, observando os pontos fracos e fortes que cada configuração escolhida possui.

Como continuidade para este trabalho, melhorias certamente podem ser realizadas. Além de estimar o consumo de potência dos processadores, os outros componentes, como barramento, memória compartilhada, *timer,* também poderiam ser avaliados, deixando a estimativa de potência consumida mais precisa. Adaptar um *micro kernel* para controlar cada processador instanciado e a comunicação com outros componentes inseridos possibilitaria que ambientes mais complexos fossem explorados. Um protocolo de coerência de *caches*, e a adoção de outros tipos de estruturas de comunicação tais como redes intra-chips (NoCs - *Network-on-Chip*) também poderiam ser implementados, aumentando a capacidade de exploração de projeto.

Apêndice A

Arquivo de Configuração do *sim-outorder*

```
# random number generator seed (0 for timer seed)
-seed                             1

# simulator scheduling priority
-nice                             0

# maximum number of inst's to execute
-max:inst                         0

# number of insts skipped before timing starts
-fastfwd                          0

# operate in backward-compatible bugs mode (for testing only)
-bugcompat                    false

# extra branch mis-prediction latency
-fetch:mplat                      3

# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred                        bimod

# bimodal predictor config (<table size>)
-bpred:bimod           512

# 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:2lev            1 1024 8 0

# combining predictor config (<meta_table_size>)
-bpred:comb            1024

# return address stack size (0 for no return stack)
-bpred:ras                        8

# BTB config (<num_sets> <associativity>)
-bpred:btb             512 1

# speed of front-end of machine relative to execution core
-fetch:speed                      1

# instruction fetch queue size (in insts)
-fetch:ifqsize                    4
```

```
# instruction decode B/W (insts/cycle)
-decode:width                     4

# instruction issue B/W (insts/cycle)
-issue:width                      4

# run pipeline with in-order issue
-issue:inorder                false

# issue instructions down wrong execution paths
-issue:wrongpath               true

# instruction commit B/W (insts/cycle)
-commit:width                     4

# register update unit (RUU) size
-ruu:size                        16

# load/store queue (LSQ) size
-lsq:size                        16

# total number of integer ALU's available
-res:ialu                         2

# total number of integer multiplier/dividers available
-res:imult                        1

# total number of memory system ports available (to CPU)
-res:memport                      2

# total number of floating point ALU's available
-res:fpalu                        2

# total number of floating point multiplier/dividers available
-res:fpmult                       1
# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
#-cache:il1             il1:16384:32:1:l

# l1 instruction cache hit latency (in cycles)
-cache:il1lat                     1

# l1 data cache config, i.e., {<config>|none}
#-cache:dl1             dl1:16384:32:4:l

# l1 data cache hit latency (in cycles)
-cache:dl1lat                     1

# l2 data cache config, i.e., {<config>|none}
-cache:dl2             ul2:8192:64:4:l

# l2 data cache hit latency (in cycles)
-cache:dl2lat                     6

# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                      dl2

# l2 instruction cache hit latency (in cycles)
-cache:il2lat                     6
```

```
# flush caches on system calls
-cache:flush                  false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress              false

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat                        18 2

# memory access bus width (in bytes)
-mem:width                        8

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:4096:4096:4:l

# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:4096:4096:4:l

# inst/data TLB miss latency (in cycles)
-tlb:lat          70
```

Apêndice B
Arquivo Resultado do *Sim-Wattch*

```
sim: ** simulation statistics **
sim_num_insn              27272195 # total number of instructions
committed
sim_num_refs               7620769 # total number of loads and stores
committed
sim_num_loads              5817175 # total number of loads committed
sim_num_stores        1803594.0000 # total number of stores committed
sim_num_branches           4138046 # total number of branches committed
sim_elapsed_time               215 # total simulation time in seconds
sim_inst_rate          126847.4186 # simulation speed (in insts/sec)
sim_total_insn            30038032 # total number of instructions executed
sim_total_refs             8252986 # total number of loads and stores
executed
sim_total_loads            6303710 # total number of loads executed
sim_total_stores      1949276.0000 # total number of stores executed
sim_total_branches         4670325 # total number of branches executed
sim_cycle                 17763754 # total simulation time in cycles
sim_IPC                     1.5353 # instructions per cycle
sim_CPI                     0.6514 # cycles per instruction
sim_exec_BW                 1.6910 # total instructions (mis-spec +
committed) per cycle
sim_IPB                     6.5906 # instruction per branch
IFQ_count                 54669054 # cumulative IFQ occupancy
IFQ_fcount                12047338 # cumulative IFQ full count
ifq_occupancy               3.0776 # avg IFQ occupancy (insn's)
ifq_rate                    1.6910 # avg IFQ dispatch rate (insn/cycle)
ifq_latency                 1.8200 # avg IFQ occupant latency (cycle's)
ifq_full                    0.6782 # fraction of time (cycle's) IFQ was
full
RUU_count                223658793 # cumulative RUU occupancy
RUU_fcount                11626101 # cumulative RUU full count
ruu_occupancy              12.5907 # avg RUU occupancy (insn's)
ruu_rate                    1.6910 # avg RUU dispatch rate (insn/cycle)
ruu_latency                 7.4459 # avg RUU occupant latency (cycle's)
ruu_full                    0.6545 # fraction of time (cycle's) RUU was
full
LSQ_count                 61937506 # cumulative LSQ occupancy
LSQ_fcount                   41345 # cumulative LSQ full count
lsq_occupancy               3.4867 # avg LSQ occupancy (insn's)
lsq_rate                    1.6910 # avg LSQ dispatch rate (insn/cycle)
lsq_latency                 2.0620 # avg LSQ occupant latency (cycle's)
```

```
lsq_full                      0.0023 # fraction of time (cycle's) LSQ was
full
bpred_bimod.lookups         4862699 # total number of bpred lookups
bpred_bimod.updates         4138046 # total number of updates
bpred_bimod.addr_hits       3857835 # total number of address-predicted
hits
bpred_bimod.dir_hits        3867023 # total number of direction-predicted
hits (includes addr-hits)
bpred_bimod.misses           271023 # total number of misses
bpred_bimod.jr_hits           61190 # total number of address-predicted
hits for JR's
bpred_bimod.jr_seen           61994 # total number of JR's seen
bpred_bimod.jr_non_ras_hits.PP          209 # total number of address-
predicted hits for non-RAS JR's
bpred_bimod.jr_non_ras_seen.PP          265 # total number of non-RAS JR's
seen
bpred_bimod.bpred_addr_rate    0.9323 # branch address-prediction rate
(i.e., addr-hits/updates)
bpred_bimod.bpred_dir_rate    0.9345 # branch direction-prediction rate
(i.e., all-hits/updates)
bpred_bimod.bpred_jr_rate    0.9870 # JR address-prediction rate (i.e., JR
addr-hits/JRs seen)
bpred_bimod.bpred_jr_non_ras_rate.PP    0.7887 # non-RAS JR addr-pred rate
(ie, non-RAS JR hits/JRs seen)
bpred_bimod.retstack_pushes        66892 # total number of address pushed
onto ret-addr stack
bpred_bimod.retstack_pops        62542 # total number of address popped off
of ret-addr stack
bpred_bimod.used_ras.PP        61729 # total number of RAS predictions used
bpred_bimod.ras_hits.PP        60981 # total number of RAS hits
bpred_bimod.ras_rate.PP    0.9879 # RAS prediction rate (i.e., RAS
hits/used RAS)
il1.accesses               31293993 # total number of accesses
il1.hits                   30907424 # total number of hits
il1.misses                   386569 # total number of misses
il1.replacements             386313 # total number of replacements
il1.writebacks                    0 # total number of writebacks
il1.invalidations                 0 # total number of invalidations
il1.miss_rate                0.0124 # miss rate (i.e., misses/ref)
il1.repl_rate                0.0123 # replacement rate (i.e., repls/ref)
il1.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
dl1.accesses                7751651 # total number of accesses
dl1.hits                    7644459 # total number of hits
dl1.misses                   107192 # total number of misses
dl1.replacements             106936 # total number of replacements
dl1.writebacks                22626 # total number of writebacks
dl1.invalidations                 0 # total number of invalidations
dl1.miss_rate                0.0138 # miss rate (i.e., misses/ref)
dl1.repl_rate                0.0138 # replacement rate (i.e., repls/ref)
dl1.wb_rate                  0.0029 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ul2.accesses                 516387 # total number of accesses
ul2.hits                     510983 # total number of hits
ul2.misses                     5404 # total number of misses
ul2.replacements                  0 # total number of replacements
ul2.writebacks                    0 # total number of writebacks
ul2.invalidations                 0 # total number of invalidations
ul2.miss_rate                0.0105 # miss rate (i.e., misses/ref)
```

```
ul2.repl_rate                0.0000 # replacement rate (i.e., repls/ref)
ul2.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
ul2.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
itlb.accesses              31293993 # total number of accesses
itlb.hits                  31293955 # total number of hits
itlb.misses                      38 # total number of misses
itlb.replacements                 0 # total number of replacements
itlb.writebacks                   0 # total number of writebacks
itlb.invalidations                0 # total number of invalidations
itlb.miss_rate               0.0000 # miss rate (i.e., misses/ref)
itlb.repl_rate               0.0000 # replacement rate (i.e., repls/ref)
itlb.wb_rate                 0.0000 # writeback rate (i.e., wrbks/ref)
itlb.inv_rate                0.0000 # invalidation rate (i.e., invs/ref)
dtlb.accesses               7755701 # total number of accesses
dtlb.hits                   7755623 # total number of hits
dtlb.misses                      78 # total number of misses
dtlb.replacements                 0 # total number of replacements
dtlb.writebacks                   0 # total number of writebacks
dtlb.invalidations                0 # total number of invalidations
dtlb.miss_rate               0.0000 # miss rate (i.e., misses/ref)
dtlb.repl_rate               0.0000 # replacement rate (i.e., repls/ref)
dtlb.wb_rate                 0.0000 # writeback rate (i.e., wrbks/ref)
dtlb.inv_rate                0.0000 # invalidation rate (i.e., invs/ref)
rename_power           7424636.0862 # total power usage of rename unit
bpred_power            23591570.0483 # total power usage of bpred unit
window_power           39156987.8808 # total power usage of instruction
window
lsq_power              23736717.3275 # total power usage of load/store
queue
regfile_power          63460515.6364 # total power usage of arch. regfile
icache_power           890078646.6573 # total power usage of icache
dcache_power           1780557351.1744 # total power usage of dcache
dcache2_power          496325542.3906 # total power usage of dcache2
alu_power              168233190.1491 # total power usage of alu
falu_power             126842484.6966 # total power usage of falu
resultbus_power        24086352.7737 # total power usage of resultbus
clock_power            364178500.4261 # total power usage of clock
avg_rename_power             0.4180 # avg power usage of rename unit
avg_bpred_power              1.3281 # avg power usage of bpred unit
avg_window_power             2.2043 # avg power usage of instruction window
avg_lsq_power                1.3362 # avg power usage of lsq
avg_regfile_power            3.5725 # avg power usage of arch. regfile
avg_icache_power            50.1064 # avg power usage of icache
avg_dcache_power           100.2354 # avg power usage of dcache
avg_dcache2_power           27.9404 # avg power usage of dcache2
avg_alu_power                9.4706 # avg power usage of alu
avg_falu_power               7.1405 # avg power usage of falu
avg_resultbus_power          1.3559 # avg power usage of resultbus
avg_clock_power             20.5012 # avg power usage of clock
fetch_stage_power      913670216.7056 # total power usage of fetch stage
dispatch_stage_power   7424636.0862 # total power usage of dispatch stage
issue_stage_power      2532096141.6961 # total power usage of issue stage
avg_fetch_power             51.4345 # average power of fetch unit per cycle
avg_dispatch_power           0.4180 # average power of dispatch unit per
cycle
avg_issue_power            142.5429 # average power of issue unit per cycle
total_power            3880830010.5503 # total power per cycle
avg_total_power_cycle      218.4690 # average total power per cycle
```

```
avg_total_power_cycle_nofp_nod2     183.3882 # average total power per
cycle
avg_total_power_insn       129.1972 # average total power per insn
avg_total_power_insn_nofp_nod2     108.4512 # average total power per insn
rename_power_cc1       5315035.1238 # total power usage of rename unit_cc1
bpred_power_cc1        4486137.8176 # total power usage of bpred unit_cc1
window_power_cc1       31267970.3761 # total power usage of instruction
window_cc1
lsq_power_cc1          5021081.4002 # total power usage of lsq_cc1
regfile_power_cc1      42051287.0077 # total power usage of arch.
regfile_cc1
icache_power_cc1       676936532.3149 # total power usage of icache_cc1
dcache_power_cc1       608529287.7029 # total power usage of dcache_cc1
dcache2_power_cc1      13766141.6611 # total power usage of dcache2_cc1
alu_power_cc1          33955611.8363 # total power usage of alu_cc1
resultbus_power_cc1    17511424.1247 # total power usage of resultbus_cc1
clock_power_cc1        173498743.2156 # total power usage of clock_cc1
avg_rename_power_cc1         0.2992 # avg power usage of rename unit_cc1
avg_bpred_power_cc1          0.2525 # avg power usage of bpred unit_cc1
avg_window_power_cc1         1.7602 # avg power usage of instruction
window_cc1
avg_lsq_power_cc1            0.2827 # avg power usage of lsq_cc1
avg_regfile_power_cc1        2.3673 # avg power usage of arch. regfile_cc1
avg_icache_power_cc1        38.1077 # avg power usage of icache_cc1
avg_dcache_power_cc1        34.2568 # avg power usage of dcache_cc1
avg_dcache2_power_cc1        0.7750 # avg power usage of dcache2_cc1
avg_alu_power_cc1            1.9115 # avg power usage of alu_cc1
avg_resultbus_power_cc1       0.9858 # avg power usage of resultbus_cc1
avg_clock_power_cc1          9.7670 # avg power usage of clock_cc1
fetch_stage_power_cc1  681422670.1324 # total power usage of fetch
stage_cc1
dispatch_stage_power_cc1 5315035.1238 # total power usage of dispatch
stage_cc1
issue_stage_power_cc1  710051517.1012 # total power usage of issue
stage_cc1
avg_fetch_power_cc1         38.3603 # average power of fetch unit per
cycle_cc1
avg_dispatch_power_cc1       0.2992 # average power of dispatch unit per
cycle_cc1
avg_issue_power_cc1         39.9719 # average power of issue unit per
cycle_cc1
total_power_cycle_cc1  1612339252.5808 # total power per cycle_cc1
avg_total_power_cycle_cc1      90.7657 # average total power per cycle_cc1
avg_total_power_insn_cc1      53.6766 # average total power per insn_cc1
rename_power_cc2       3137470.9876 # total power usage of rename unit_cc2
bpred_power_cc2        2747814.5118 # total power usage of bpred unit_cc2
window_power_cc2       21352449.7182 # total power usage of instruction
window_cc2
lsq_power_cc2          2920946.1651 # total power usage of lsq_cc2
regfile_power_cc2      9279820.1141 # total power usage of arch.
regfile_cc2
icache_power_cc2       676936532.3149 # total power usage of icache_cc2
dcache_power_cc2       388494998.6023 # total power usage of dcache_cc2
dcache2_power_cc2      7214017.3169 # total power usage of dcache2_cc2
alu_power_cc2          32077851.8801 # total power usage of alu_cc2
resultbus_power_cc2    9172643.5590 # total power usage of resultbus_cc2
clock_power_cc2        140579702.1020 # total power usage of clock_cc2
avg_rename_power_cc2         0.1766 # avg power usage of rename unit_cc2
avg_bpred_power_cc2          0.1547 # avg power usage of bpred unit_cc2
```

```
avg_window_power_cc2         1.2020 # avg power usage of instruction
window_cc2
avg_lsq_power_cc2            0.1644 # avg power usage of instruction
lsq_cc2
avg_regfile_power_cc2        0.5224 # avg power usage of arch. regfile_cc2
avg_icache_power_cc2        38.1077 # avg power usage of icache_cc2
avg_dcache_power_cc2        21.8701 # avg power usage of dcache_cc2
avg_dcache2_power_cc2        0.4061 # avg power usage of dcache2_cc2
avg_alu_power_cc2            1.8058 # avg power usage of alu_cc2
avg_resultbus_power_cc2       0.5164 # avg power usage of resultbus_cc2
avg_clock_power_cc2          7.9139 # avg power usage of clock_cc2
fetch_stage_power_cc2  679684346.8267 # total power usage of fetch
stage_cc2
dispatch_stage_power_cc2 3137470.9876 # total power usage of dispatch
stage_cc2
issue_stage_power_cc2  461232907.2416 # total power usage of issue
stage_cc2
avg_fetch_power_cc2         38.2624 # average power of fetch unit per
cycle_cc2
avg_dispatch_power_cc2       0.1766 # average power of dispatch unit per
cycle_cc2
avg_issue_power_cc2         25.9648 # average power of issue unit per
cycle_cc2
total_power_cycle_cc2  1293914247.2719 # total power per cycle_cc2
avg_total_power_cycle_cc2      72.8401 # average total power per cycle_cc2
avg_total_power_insn_cc2      43.0759 # average total power per insn_cc2
rename_power_cc3       3348431.0842 # total power usage of rename unit_cc3
bpred_power_cc3        4658378.5867 # total power usage of bpred unit_cc3
window_power_cc3       21769075.7199 # total power usage of instruction
window_cc3
lsq_power_cc3          4774085.0098 # total power usage of lsq_cc3
regfile_power_cc3      10799022.8077 # total power usage of arch.
regfile_cc3
icache_power_cc3       698250743.6588 # total power usage of icache_cc3
dcache_power_cc3       506610443.4514 # total power usage of dcache_cc3
dcache2_power_cc3      55471185.3895 # total power usage of dcache2_cc3
alu_power_cc3          45505609.7049 # total power usage of alu_cc3
resultbus_power_cc3    9635512.5460 # total power usage of resultbus_cc3
clock_power_cc3        159547889.2657 # total power usage of clock_cc3
avg_rename_power_cc3         0.1885 # avg power usage of rename unit_cc3
avg_bpred_power_cc3          0.2622 # avg power usage of bpred unit_cc3
avg_window_power_cc3         1.2255 # avg power usage of instruction
window_cc3
avg_lsq_power_cc3            0.2688 # avg power usage of instruction
lsq_cc3
avg_regfile_power_cc3        0.6079 # avg power usage of arch. regfile_cc3
avg_icache_power_cc3        39.3076 # avg power usage of icache_cc3
avg_dcache_power_cc3        28.5193 # avg power usage of dcache_cc3
avg_dcache2_power_cc3        3.1227 # avg power usage of dcache2_cc3
avg_alu_power_cc3            2.5617 # avg power usage of alu_cc3
avg_resultbus_power_cc3       0.5424 # avg power usage of resultbus_cc3
avg_clock_power_cc3          8.9817 # avg power usage of clock_cc3
fetch_stage_power_cc3  702909122.2455 # total power usage of fetch
stage_cc3
dispatch_stage_power_cc3 3348431.0842 # total power usage of dispatch
stage_cc3
issue_stage_power_cc3  643765911.8216 # total power usage of issue
stage_cc3
```

```
avg_fetch_power_cc3         39.5699 # average power of fetch unit per
cycle_cc3
avg_dispatch_power_cc3       0.1885 # average power of dispatch unit per
cycle_cc3
avg_issue_power_cc3         36.2404 # average power of issue unit per
cycle_cc3
total_power_cycle_cc3  1520370377.2246 # total power per cycle_cc3
avg_total_power_cycle_cc3      85.5883 # average total power per cycle_cc3
avg_total_power_insn_cc3      50.6148 # average total power per insn_cc3
total_rename_access        30026125 # total number accesses of rename unit
total_bpred_access          4138046 # total number accesses of bpred unit
total_window_access       100102270 # total number accesses of instruction
window
total_lsq_access            7767809 # total number accesses of load/store
queue
total_regfile_access       36367381 # total number accesses of arch.
regfile
total_icache_access        31309135 # total number accesses of icache
total_dcache_access         7751651 # total number accesses of dcache
total_dcache2_access         516387 # total number accesses of dcache2
total_alu_access           27533866 # total number accesses of alu
total_resultbus_access     29882896 # total number accesses of resultbus
avg_rename_access            1.6903 # avg number accesses of rename unit
avg_bpred_access             0.2329 # avg number accesses of bpred unit
avg_window_access            5.6352 # avg number accesses of instruction
window
avg_lsq_access               0.4373 # avg number accesses of lsq
avg_regfile_access           2.0473 # avg number accesses of arch. regfile
avg_icache_access            1.7625 # avg number accesses of icache
avg_dcache_access            0.4364 # avg number accesses of dcache
avg_dcache2_access           0.0291 # avg number accesses of dcache2
avg_alu_access               1.5500 # avg number accesses of alu
avg_resultbus_access         1.6822 # avg number accesses of resultbus
max_rename_access                 4 # max number accesses of rename unit
max_bpred_access                  3 # max number accesses of bpred unit
max_window_access                16 # max number accesses of instruction
window
max_lsq_access                    6 # max number accesses of load/store
queue
max_regfile_access               12 # max number accesses of arch. regfile
max_icache_access                 4 # max number accesses of icache
max_dcache_access                 4 # max number accesses of dcache
max_dcache2_access                5 # max number accesses of dcache2
max_alu_access                    3 # max number accesses of alu
max_resultbus_access             10 # max number accesses of resultbus
max_cycle_power_cc1        272.5637 # maximum cycle power usage of cc1
max_cycle_power_cc2        267.4101 # maximum cycle power usage of cc2
max_cycle_power_cc3        269.2339 # maximum cycle power usage of cc3
sim_invalid_addrs                 0 # total non-speculative bogus addresses
seen (debug var)
ld_text_base             0x00400000 # program text (code) segment base
ld_text_size                 225760 # program text (code) size in bytes
ld_data_base             0x10000000 # program initialized data segment base
ld_data_size                  21904 # program init'ed `.data' and uninit'ed
`.bss' size in bytes
ld_stack_base            0x7fffc000 # program stack segment base (highest
address in stack)
ld_stack_size                 16384 # program initial stack size
ld_prog_entry            0x00400140 # program entry point (initial PC)
```

```
ld_environ_base          0x7fff8000 # program environment base address
address
ld_target_big_endian              0 # target executable endian-ness, non-
zero if big endian
mem.page_count                  133 # total number of pages allocated
mem.page_mem                   532k # total size of memory pages allocated
mem.ptab_misses                 136 # total first level page table misses
mem.ptab_accesses          79811551 # total page table accesses
mem.ptab_miss_rate           0.0000 # first level page table
miss rate
```

Apêndice C
Utilizando o VIPRO-MP

O VIPRO-MP segue o mesmo padrão de entrada de parâmetros do SimpleScalar, por meio de um arquivo de configuração (Apêndice A), linha de comando, ou ambos. Abaixo está uma chamada ao VIPRO-MP que simulará a execução de *prog*, onde parte das configurações está no arquivo *configure* e parte em linha de comando, como a configuração da *cache* de dados (DL1).

```
vipro-mp -cache:dl1 dl1:128:32:4:l -config configure prog
```

O que será feito com cada parâmetro de entrada é definido pelo projetista no arquivo principal *main_sc.cpp.* Na Seção 1 desse Apêndice, um exemplo deste arquivo, que modela um ambiente com barramento, *timer*, árbitro, memória compartilhada e dois processadores, está exposto na íntegra, e entre as linhas 21 e 30 os parâmetros são facilmente transferidos aos processadores instanciados.

Percebe-se, portanto, que este arquivo principal é responsável pela configuração das características referentes à plataforma multiprocessada. A instanciação e organização dos objetos diretamente no *main_sc.cpp* permite que a plataforma se adapte às necessidades do projetista, montando a arquitetura desejada com facilidade. Os componentes **barramento**, **memória compartilhada** e **processador** foram modelados a fim de permitir que alguns parâmetros sejam informados na sua criação, além de seu nome (parâmetro utilizado pelo SystemC).

Ao instanciar o barramento, é possível ativar o modo *debug* (linha 18, segundo argumento), que armazena em *log* toda movimentação ocorrida, informando o estado de cada requisição. A memória compartilhada possui latência, endereço base e final pré-estabelecidos (18 ciclos, 0x80000000, 0x8fffffff, respectivamente), porém eles podem ser alterados na

criação do objeto (linha 16), onde o quarto argumento é a latência. Quando mais de um processador é instanciado, o árbitro necessita de um mecanismo de desempate para eleger quem ganhará o barramento e para isso, um ID único para cada processador foi utilizado (linha 47 e 56). Como os menores ID terão preferência quando um empate ocorrer (Seção 5.1.4), é aconselhável que o processador mestre seja associado ao menor valor da faixa adotada.

Para automatizar o processo de simulação de várias arquiteturas, com aplicativos diferentes ou não, utiliza-se arquivos escritos em *shell script* [43], denominados arquivos *scripts*. Em um *script*, é possível definir um laço no qual, a cada iteração, uma configuração é atribuída ao VIPRO-MP, que gera um arquivo de resultados. Ao término do laço, o projetista contará com vários arquivos resultados que serão analisados posteriormente. A Seção 2 deste Apêndice exibe um exemplo de *script,* que foi utilizado no primeiro estudo de caso sobre o algoritmo JPEG, Seção 6.2.1, onde as *caches* IL1 e DL1 variaram independentemente. A linha 8 insere as configurações dos processadores, que serão tratadas no arquivo *main_sc.cpp*; na linha 9 e 10 estão os binários à executar em cada processador e o futuro nome do arquivo de resultado, respectivamente.

Obviamente, analisar os 63 arquivos resultados gerados por este *script*, manualmente, é inviável e oneroso. Por isso, foi desenvolvido um aplicativo em MatLab no qual explora todos os arquivos presentes em uma pasta, montando gráficos para análise. Para saber quais valores utilizar na construção do gráfico, o projetista informa os campos desejados, como por exemplo *sim_cycle* e *LSQ_count* (Apêndice A).

1. Arquivo *main_sc.cpp*:

```
1.      #include <systemc.h>
2.      #include <unistd.h>
3.      #include "simplescalar.hpp"
4.      /*MSG*/
5.      #include "memoryshared.hpp"
6.      #include "bus/my_bus.hpp"
7.      #include "bus/simple_bus_arbiter.hpp"
8.      #include "int_time.hpp"
9.
10.     /*MSO (06/06/2008) Variable declared to obtain the environ directly
11.       In SystemC programs we dont have the main, but sc_main*/
12.     extern char **environ;
13.
14.     int sc_main(int argc, char *argv[]){
15.
16.         memshared             *mshared = new memshared("mem", 0x80000000, 0x8fffffff,32);
17.         simple_bus_arbiter       *arbiter = new simple_bus_arbiter("arbiter");
18.         my_bus                    *bus     = new my_bus("bus", false);
19.         int_time       *int_generator     = new int_time("int_generator");
20.
```

```
21.         /* atribuindo os paramentros dos processadores */
22.         // PROC1
23.         const int argc1 = 14;
24.         char *argv1[argc1] = {argv[0],argv[1],argv[2],argv[3], argv[4], argv[5], argv[7],
25.                        argv[9],argv[11],argv[12],argv[14],argv[15],argv[16], argv[19]};
26.
27.         // PROC2
28.         const int argc2 = 13;
29.         char *argv2[argc2] = {argv[0],argv[1],argv[2],argv[3], argv[4], argv[5],argv[7],
30.                        argv[18],argv[11],argv[12],argv[14],argv[15],argv[16]};
31.
32.         FILE *fd= fopen(argv[20], "w+" );
33.         sc_signal<bool> reset;
34.
35.         /* instanciando o clock da simulaÃ§Ã£o */
36.         sc_clock clock("CLOCK", 10, 0.5, 0.0);
37.
38.         /* ligando o sinal de clock na porta de clock dos componentes */
39.         bus->clock(clock);
40.         mshared->clock(clock);
41.         int_generator->clock(clock);
42.
43.         bus->slave_port(*mshared);
44.         bus->arbiter_port(*arbiter);
45.
46.         /* Processador */
47.         simplescalar scsp("PROC1", 1);
48.         scsp.setFd(fd);                       // atribuindo o arquivo output
49.         scsp.CLK(clock);                      // ligando o sinal de clock no procesador
50.         scsp.bus_port(*bus);          // ligando a porta do barramento com o barramento
51.         int_generator->interrupt_port(scsp); // ligando a porta interrupt no timer.
52.         scsp.init(argc1, argv1, environ);    // passando os parametros de PROC1
53.         scsp.reset(reset);
54.
55.         /* Processador */
56.         simplescalar scsp2("PROC2", 2);
57.         scsp2.setFd(fd);
58.         scsp2.CLK(clock);
59.         scsp2.bus_port(*bus);
60.         int_generator->interrupt_port(scsp2);
61.         scsp2.init(argc2, argv2, environ);
62.         scsp2.reset(reset);
63.
64.         cout << "Starting";
65.         sc_start(clock,3e8);                  // iniciando o clock e todas as threads
66.         cout << "Finished SystemC simulation" << endl;
67.         sc_stop();
68.
69.         /* imprime a memoria compartilhada, dado o intervalo */
70.         //mshared->print_mem(0x80090000, 0x8fffffff);
71.
72.         return 0;
73.     }
```

2. Arquivo *script_jpeg_cache.sh*:

```
1.      #!/bin/bash
2.
3.      for ((i = 64; i <= 16384; i = i+i))
4.      do
5.          for ((j=256; j <= 16384; j = j+j))
6.          do
7.              ./sim-outorder-mp \
8.              -cache:dl1 dl1:$i:32:4:l -cache:il1 il1:$j:32:1:l -config configure \
9.              jpeg1 jpeg2 \
10.             dump_$i-$j.txt
11.         done
12.     done
```

Referências Bibliográficas

[1] AZEVEDO, R.; RIGO, S.; ARAÚJO, G. **Projeto e Desenvolvimento de Sistemas Dedicados Multiprocessados.** In: Livro das Jornadas de Atualização em Informática by Karin Breitman e Ricardo Anido, 2006, Campo Grande.

[2] BARRETO, M. E.; ÁVILA, R. B., OLIVEIRA, F. A. D. **Execução de aplicações em ambientes concorrentes.** Trabalho Independente. Porto Alegre: UFRGS. Disponível em: http://www.inf.ufrgs.br/~avila/download/Navaux:EAA-ERAD01.pdf. Acessado em: 11/nov/2008.

[3] BELANOVIC, P. et al. **Automatic Generation of Virtual Prototypes.** In: IEEE International Workshop on Rapid System Prototyping, 2004, Genebra. **Proceedings…** Washington: IEEE Press, 2004. p 114-118.

[4] BROOKS, D.; TIWARI, V.; MARTONOSI, M. **Wattch: A Framework for Architectural-Level Power Analysis and Optimizations.** In: International Symposium on Computer Architecture, ISCA: 2000, Vancouver, B.C, p. 83-94.

[5] BURGER, D.; AUSTIN, T. M. **The SimpleScalar Tool Set: Version 2.0.** Technical Report, n.1342. Madison: University of Wisconsin, 1997.

[6] CANCIAN, R. L.; STEMMER, M. R.; SCHULTER, A.; FROLICH, A. A. **Ferramenta de Suporte ao Projeto Automatizado de Sistemas Computacionais Embarcados .** In: XXVII Congresso da SBC, Rio de Janeiro, 2007, p. 805-815.

[7] CONSTANTINI, U. **Simulação de sistemas embarcados multiprocessadores utilizando a linguagem SystemC.** Cascavel: UNIOESTE, Novembro, 2007. Monografia.

[8] DE MICHELL, G. D.; GUPTA, R.K. **Hardware/software co-design**. In: Proceedings of the IEEE. Volume 85, Issue 3, 1997. p 349 – 365.

[9] DOTZEL, G. **LSI-11, RT-11, Megabytes of Memory and Modula-2/VRS.** In: DEC PROFESSIONAL, The Magazine for DEC Users. Spring House, PA. 1986.

[10] **ECLIPSE C/C++ Development Tooling**. Disponível em: http://www.eclipse.org/cdt/. Acessado em: 07/nov/2008

[11] FISHER, J. A., FARABOSCHI, P., YOUNG, C. **Embedded Computing: A Vliw Approach to Architecture, Compilers and Tools**. 1st Edition. New York: Morgan Kaufmann, 2005.

[12] FRÖHLICH, A. A. M. **Gerência do Consumo de Energia Dirigida pela Aplicação em Sistemas Embarcados .** Florianópolis: UFSC, Fevereiro, 2007. Dissertação.

[13] GARCIA, M. S., SCHUCK, M., OYAMADA, M. S. **VIPRO-MP: uma plataforma virtual multiprocessada com estimativa de consumo de potência** . In: VIII FITEM – Fórum de Informática e Tecnologia de Maringá, Maringá - PR, 2008, p. 35-45.

[14] GELSINGER, P. P., GARGINI, P. A., PARKER, G. H., YU, A. Y. C. **Microprocessors Circa 2000**. In: IEEE Spectrum, 1989, p. 43-47. Disponível em http://www.intel.com/ research/silicon/ieee/circa2000.pdf. Acessado em: 07/nov/2008

[15] GIVARGIS, T; VAHID, F. **PLATUNE: A Tuning Framework for System-on-a-Chip Platforms**. In: IEEE Tranactions on Computer – Aided Design of integrated circuits and systems, Vol. 21, No. 11, 2002, p. 1317-1327.

[16] GONÇALVES, R. A. **PowerSMT: FERRAMENTA PARA ANÁLISE DE CONSUMO DE POTÊNCIA EM ARQUITETURAS SMT**. Maringá: UEM, 2008. Dissertação.

[17] GOOSSENS, K., et al. **Service-Based Design of Systems on Chip and Networks on Chip.** In: Dynamic and Robust Streaming in and Between Connected Consumer-Electronics Devices, Springer, 2005. v.3, p. 37-60.

[18] HENNESSY, J., PATTERSON, D. A. **Computer Architecture: A Quantitative Approach.** 4th Edition. CA: Morgan Kaufmann, 2006.

[19] IBM. **IBM Cell Broadband Engine Architecture**, 2007. Disponível em: http://www.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA. Acessado em: 07/nov/2008

[20] INTEL. **Intel Multi-Core Processors: Leading the Next Digital Revolution.** Disponível em: www.intel-inside.ro/technology/magazine/computing/multi-core-0905.pdf. Acessado em: 22/jul/2008.

[21] **Intel Core 2 Duo.** Disponível em : http://www.intel.com/products/processor/core2duo/index.htm. Acessado em: 07/nov/2008.

[22] ITRS. **International Technology Roadmap for Semiconductors, version 2001. Design**. Disponível em: http://www.itrs.net/Links/2001ITRS/Design.pdf. Acesssado em: 22/jul/2008.

[23] JERRAYA, A. A. Long Term Trends for Embedded System Design. In: EUROMICRO Symposium on Digital System Design, 2004, Rennes. **Proceedings….** Washington: IEEE Press, 2004, p. 20-26.

[24] JOUPPI, N. and WILTON, S. **An enhanced access and cycle time model for on-chip caches.** Technical Report TR-93-5, Compaq WRL, 1994.

[25] KALAVADE, A.; LEE, E. A. **Hardware / Software Co-Design Using Ptolemy – A Case Study.** In: Proceedings of the First Intl. Workshop on Hardware/Software Codesign, Colorado, 1992, p 1-18.

[26] KAY, R. AND THIBODEAU, P. Counting Cores. **Computer World.** June 20, 2005. Disponível em: http://www.computerworld.com. Acessado em: 26/jul/2008

[27] KRASNER, J. **Embedded Development Issues and Challenges**. In: Embedded Market Forecasts, 2003. Disponível em http://www.embeddedforecast.com. Acessado em: 28/mar/2008.

[28] **RealView SoC Designer.** Disponível em http://www.arm.com/products/DevTools/SoCDesigner.html. Acessado em: 28/mar/2008.

[29] REINMAN, G. and JOUPPI, N. **CACTI 2.0: An integrated cache timing and power model.** WRL Research Report 2000/7, 2000.

[30] MAEURER, T. **IBM Systems and Technology Group.** Austin, Texas. Cell Architecture and Broadband Engine Processor. 2005.

[31] MAMIDIPAKA, M. and DUTT, N. **eCACTI: An Enhanced Power Estimation Model for On-chip Caches**. Technical Report TR-04-28. University of California. Irvine Center for Embedded Computer Systems, 2004.

[32] MELLO, A. V., MÖLLER, L. H. **Arquitetura Multiprocessada em SoCs: Estudo de Diferentes Topologias de Conexão.** Porto Alegre: PUC-RS, Junho, 2003. Monografia.

[33] **MatLab Version 7 (R14).** Disponível em: http://www.mathworks.com. Acessado em: 11/11/2008.

[34] **MiBench**. Disponível em: www.eecs.umich.edu/mibench. Acessado em: 28/out/2008.

[35] **Mips.** Disponível em: http://www.mips.com. Acessado em: 28/mar/2008.

[36] MOSAID Virtual Silicon. Power Islands: The Evolving Topology of SoC Power Management. **Design Reuse.** Disponível em: http://www.designreuse.com/articles/9150/power-islands-the-evolving-topology-of-soc-power-management.html

[37] **MPI.** Disponível em: http://www-unix.mcs.anl.gov/mpi. Acessado em: 22/jul/2008.

[38] MUDRY, G. Z. P.; TEMPESTI, G. **Hybrid genetic algorithm for constrained hardware-software partitioning.** In: 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006, p 1-6.

[39] NEUMAN, A. **Market Outlook and Trends.** Disponível em: http://www.edac.org. Acessado em: 28/mar/2008.

[40] PIZZOL, G. D. **SimMan: Simulation Manager. Definição e Implementação de um Ambiente de Simulação de Arquiteturas Superescalares para a Ferramenta SimpleScalar.** Porto Alegre: Universidade Federal do Rio Grande do Sul, Maio, 2002. Monografia.

[41] REGO, R. S. L. S. **Projeto e Implementação de uma Plataforma MP-SoC usando SystemC .** Natal: Universidade Federal do Rio Grande do Norte, Janeiro, 2006. Dissertação.

[42] SHIVAKUMAR, P. e JOUPPI, N. P. **CACTI 3.0: An Integrated Cache Timing, Power, and Area Model**. Technical report, Compaq Computer Corporation August, 2001.

[43] GITE, V. G. **Shell Scripting Tutorial**. 2002. Disponível em: http://freeos.com/guides/lsst. Acessado em: 11/nov/2008.

[44] **SimpleScalar.** Disponível em: http://www.simplescalar.com. Acessado em: 28/mar/2008.

[45] SIMUNIC, T., *et al*. **Dynamic Voltage Scaling and Power Management for Portable Systems.** In: Annual *ACM* IEEE Design Automation Conference. **Procedings...**, June, 2001, Las Vegas.

[46] STEVENS, W. R. **UNIX Network Programming, Volume 2: Interprocess Communications**. 2nd Edition. Prentice Hall, 1999.

[47] SWAN, R. J. *et al*. **The implementation of the Cm* multi-microprocessor.** In: *AFIPS NCC*, 1977. **Proceedings...**, p. 645-654.

[48] **SUN Java Tutorials**. Disponível em: http://java.sun.com/docs/books/tutorial/essential/ concurrency. Acessado em: 11/nov/2008.

[49] **OpenMP.** Disponível em: http://openmp.org/wp. Acessado em: 22/jul/2008.

[50] OYAMADA. M. S. **Estudo de ambientes para co-simulação**. Trabalho Individual. Porto Alegre: UFRGS, 1999.

[51] **SystemC.** Disponível em: http://www.systemc.org/home. Acessado em: 28/mar/2008.

[52] TALLY, S. **'Not so fast, supercomputers,' say software programmers.** Purdue University, May 22, 2007. Disponível em: http://news.uns.purdue.edu/x/2007a/ 070522SaiedParallel. html

[53] TARJAN, D. S., THOZIYOOR, N. P., JOUPPI, N. **CACTI 4.0.** HP Laboratories . Palo Alto, HPL-2006-86, June 2, 2006.

[54] THOZIYOOR, S., MURALIMANOHAR, N., AHN, J. and JOUPPI, N. P. **CACTI 5.1**. HPL-2008-20. HP Laboratories, Palo Alto, April 2, 2008.

[55] WAGNER, F. CARRO, L. **Sistemas Computacionais Embarcados.** XXII Jornadas de Atualização em Informática. Campinas: UNICAMP, 2003, v. 1, p. 45-94.

[56] WOLF, W. **Computers as Components - Principles of Embedded Computing.** System Design. 1st Edition. San Francisco: Morgan Kaufmann Publishers, 2001.

[57] ZHANG, C., VAHID, F., LYSECKY, R.L. **A Self-Tuning Cache Architecture for Embedded Systems.** In: DESIGN AUTOMATION AND TEST IN EUROPE, 2004, Paris, France. **Proceedings...** IEEE Computer Society Press, 2004. p. 142-147.

[58] ZHANG, Y., PARIKH, D., SANKARANARAYANAN, K., SKADRON, K. and STAN, M. **HotLeakage: A temperatureaware model of subthreshold and gate leakage for architects.** TR- CS-2003-05, University of Virginia, Dept of Computer Science, March 2003.

[59] ZIVOJNOVIC, V. **Symchronous Debugging of Multicore Systems**. VP ESL Tools ARM Ltda. MPSoC'05. July, 2007.