UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JOÃO CLÁUDIO SOARES OTERO

Javarray: uma Arquitetura Reconfigurável para o Aumento de Performance e Economia de Energia de Aplicações Embarcadas Baseadas em Java.

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Flávio Rech Wagner
Orientador

Prof. Dr. Luigi Carro
Co-orientador

Porto Alegre, abril de 2006.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Otero, João Cláudio Soares

Javarray: uma Arquitetura Reconfigurável para o Aumento de Performance e Economia de Energia de Aplicações Embarcadas Baseadas em Java / João Cláudio Soares Otero – Porto Alegre: Programa de Pós-Graduação em Computação, 2006.

111 f. + 1 CD-ROM : il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2006. Orientador: Flávio Rech Wagner; Co-orientador: Luigi Carro.

1.Sistemas Embarcados. 2.Java. 3.Arquiteturas Reconfiguráveis. 4.Redução de Consumo de Energia I. Wagner, Flávio Rech. II. Carro, Luigi. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profa. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Flávio Rech Wagner
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 5

LISTA DE FIGURAS 6

LISTA DE TABELAS 8

RESUMO 9

ABSTRACT 10

1 INTRODUÇÃO 11

1.1 Motivação 11
1.2 Objetivos 12
1.3 Contribuições Originais e Resultados Obtidos 13
1.4 Estrutura do Texto 15

2 ANÁLISE DO ESTADO-DA-ARTE 16

2.1 Computação Reconfigurável 16
2.2 Desafios e Tendências 20
2.3 Arquiteturas Reconfiguráveis 23
2.3.1 PipeRench 23
2.3.2 MorphoSys 25
2.3.3 Rapid 28
2.3.4 RawMachine 29
2.3.5 KressArray 31
2.3.6 Xtreme 33
2.3.6 Análise das Arquiteturas Reconfiguráveis 35
2.4 Trabalhos Relacionados 40

3 PROJETO JAVARRAY 42

3.1 Proposta 42
3.2 Fluxo de Projeto 43
3.3 Análises de Aplicações Embarcadas 46
3.3.1 Análise de Representatividade dos Blocos Básicos nos Profiles 46
3.3.2 Análise dos Grafos de Dependência 51

4 ARQUITETURA JAVARRAY 63

4.1 Módulos 63
4.2 Restrições 65

4.3 Classificação 66
4.4 Processamento no Javarray 67
4.4.1 Fluxo de Configuração 67
4.4.2 Fluxo de Execução 70
4.5 Acoplamento ao FemtoJava 72
4.6 Organizações Arquiteturais do Javarray 74
4.6.1 Organização Javarray Sequencial 74
4.6.2 Organização Javarray Combinacional 76
5 RESULTADOS 80
5.1 Experimentos 81
5.2 Performance 92
5.3 Energia 93
5.4 Potência 95
5.5 Estimativas gerais de Área 96
5.6 Execução baseada em Stream 97
6 CONCLUSÕES 103
REFERÊNCIAS 105
CD ANEXO 111
Experimentos CACO-OS 111
Java Reference 111
Javarray Analysers 111
Modelo Quartus – Javarray Seqüencial 111
Módulos CACO-PS Criados 111
CACO-PS.zip 111
Manual_CACO-PS.pdf 111

LISTA DE ABREVIATURAS E SIGLAS

ASIC	Application Specific Integrated Circuits
ASIP	Application Specific Instruction-set Processors
CMP	Chip Multi-processor
DAG	Directed Acyclic Graph
DSP	Digital Signal Processor
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
GPP	General Pourpose Processor
I/O	Input/Output
ILP	Instruction Level Parallelism
IPC	Instruções por Ciclo
JVM	Java Virtual Machine
LSE	Laboratório de Sistemas Embarcados
LUT	Look-up-table
PAL	Programmable Array Logic
PDA	Personal Digital Assistent
PE	Processing Element
PLA	Programmable Logic Array
PML	Programmable Multilevel Logic Array
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SEEP	Sistemas Eletrônicos Embarcados baseados em Plataforma
SIMD	Single-Instruction Multiple-Data
SMTI	Simultaneous Multi-Trace Issue
ULA	Unidade Lógica e Aritmética
VLIW	Very Large Instruction Word

LISTA DE FIGURAS

Figura 1.1: Fluxo de projeto do SEEP ... 14
Figura 2.1: Possíveis localizações para um substrato reconfigurável ... 19
Figura 2.2: Pipeline temporal ... 24
Figura 2.3: Arquitetura do Piperench ... 25
Figura 2.4:Estrutura geral do MorphoSys ... 25
Figura 2.5: (a) Nível de conectividade *local* e *linha e colunas* ... 26
Figura 2.6: Célula reconfigurável do MorphoSys ... 27
Figura 2.7: Célula básica do RaPiD-1 ... 29
Figura 2.8: *Tiles* da *Raw Machine* ... 30
Figura 2.9: KressArray - I/O para as rDPU's ... 32
Figura 2.10: Roteamento das rDPU's ... 32
Figura 2.11: Arquitetura geral do SOC configurável Xtreme/Leon. ... 34
Figura 2.12: Uma das estruturas comuns das PAE's do Xtreme. ... 35
Figura 2.13: Acoplamento X Reconfigurabilidade ... 36
Figura 2.14: Configurabilidade X Profundidade de Configuração ... 36
Figura 3.1: Fluxo de desenvolvimento deste trabalho. ... 45
Figura 3.2: Representatividade dos blocos básicos selecionados no IMDCT. ... 48
Figura 3.3: Representatividade dos blocos básicos selecionados no SortBubble. ... 49
Figura 3.4: Representatividade dos blocos básicos selecionados no Crane 16bits. ... 50
Figura 3.5: Grafo de dependências do bloco básico 12 do IMDCT. ... 54
Figura 3.6: Níveis de profundidade da árvore de *iastore*. ... 55
Figura 3.7: Distribuição de instruções do bloco básico 12 do IMDCT. ... 55
Figura 3.8: Árvore de dependências do bloco básico 8 do Crane 16bits. ... 58
Figura 3.9: Instruções *iload-istore* que podem ser suprimidas. ... 59
Figura 3.10: Outras instruções iload-istore que podem ser suprimidas. ... 59
Figura 3.11: bloco básico 62 do Crane 16bits. ... 60
Figura 3.12: Ocorrem reduções de até 80% ... 61
Figura 3.13: Ocorrem reduções de até 42% ... 62
Figura 4.1: Topologia geral do Javarray. ... 64
Figura 4.2: Simplificações de um bloco de operandos. ... 68
Figura 4.3: Mapeamento de um bloco de operandos no Javarray. ... 71
Figura 4.4: Sequência de figuras demonstrando o fluxo de execução. ... 72
Figura 4.5: Integração Femtojava-Javarray como unidades funcionais replicadas. ... 73
Figura 4.6: PE do Javarray Sequencial. ... 74
Figura 4.7: PE do Javarray Combinacional Restrito. ... 77
Figura 4.8: PE do Javarray Combinacional Geral. ... 79
Figura 5.1: Árvores de dependência dos blocos básicos 6 e 7 ... 82
Figura 5.2: Comparação entre consumo de energia ... 85

Figura 5.3: Redução de consumo quase linear com a profundidade do array. 86
Figura 5.4: Comparação gráfica da potência consumida.. 96
Figura 5.5: a) Bloco de operandos 12.a do IMDCT. 98
Figura 5.6: Execução dos blocos de operando 12.a e 12.b em paralelo. 99
Figura 5.7: Utilização de *forward* em *iastore* para execução antecipada de *iaload*..... 100
Figura 5.8: Diagrama temporal do pipeline.. 101

LISTA DE TABELAS

Tabela 3.1: Estatísticas do *profile* da aplicação IMDCT. 47
Tabela 3.2: Necessidade de recursos do bloco básico 12 do IMDCT. 56
Tabela 3.3: Análise de recursos por níveis do bloco básico 12 do IMDCT 56
Tabela 3.4: Distribuição de recursos nos grafos 56
Tabela 5.1: Estatísticas Javarray Sequencial. 83
Tabela 5.2: Estatísticas Javarray Combinacional Restrito. 84
Tabela 5.3: Estatísticas da simulação do FemtoJava Multiciclo. 87
Tabela 5.4: Estatísticas da simulação do FemtoJava Low-Power. 87
Tabela 5.5: Comparações entre Javarray e FemtoJava Multiciclo. 88
Tabela 5.6: Comparações entre Javarray e FemtoJava Low-Power 89
Tabela 5.7: Economia de energia com o FemtoJava Multiciclo 90
Tabela 5.8: Economia de energia com o FemtoJava Low-Power 90
Tabela 5.9: Estimativas de otimização de desempenho com o Javarray. 91
Tabela 5.10: Potência consumida com Javarray acoplado. 92
Tabela 5.11: Economia de Ciclos produzida pelo Javarray 93
Tabela 5.12: Economia de energia produzida pelo Javarray. 94
Tabela 5.13: Potência consumida com introdução do Javarray 95
Tabela 5.14: Execução paralela das instruções do bloco de operados 12.a 99
Tabela 5.15: Execução paralela das instruções do bloco de operandos 12.b. 99

RESUMO

A popularidade da linguagem Java no mercado de sistemas embarcados está aumentando como uma alternativa à necessidade de compatibilidade de software e ao crescimento da complexidade das aplicações, notadamente em eletrônica de consumo e automação industrial, mercado que também está se expandindo.

Apesar de um melhor gerenciamento da complexidade do software permitido pela linguagem Java, as restrições de necessidade de economia de energia, baixo consumo de potência e necessidade de desempenho impostas aos sistemas embarcados, com especial ênfase aos sistemas portáteis, são potencializadas.

Entretanto, as características da Java Virtual Machine, baseada em uma máquina de pilha, abrem possibilidades de otimização do processamento de aplicações embarcadas inerentes às máquinas de pilha e ainda não devidamente exploradas pelos processadores Java atuais. Com a aplicação de tradução binária ao código Java e utilização de técnicas de reconfiguração, consegue-se obter aumento de performance com simultânea economia de energia, permitindo-se uma melhor adequação da execução das aplicações Java para o domínio dos sistemas embarcados.

Este trabalho apresenta uma unidade reconfigurável de granularidade grossa, o Javarray, a ser acoplada a um processador de execução Java nativa, destinada à execução otimizada dos blocos básicos mais representativos das aplicações embarcadas Java. Dessa forma, conseguimos explorar ILP de uma maneira simples e com a reconfiguração de poucos blocos básicos obtivemos uma redução no número de instruções executadas em até 42%, aumentamos o desempenho das aplicações em até 2,6 vezes e obtivemos economias de energia de até 64%, ao mesmo tempo em que mantivemos compatibilidade de software com as aplicações Java, e em muitos casos obtivemos simultânea redução na potência consumida. Esses dados referem-se a um conjunto de 3 aplicações específicas utilizadas por nosso grupo.

A topologia básica do Javarray é desenvolvida a partir da análise de profiles de aplicações embarcadas, a partir da qual algumas variações organizacionais são exploradas. Em especial, desenvolveu-se uma arquitetura seqüencial, que habilita a utilização de técnicas de pipeline no Javarray, permitindo a exploração de paralelismo de mais alto nível. Como produto secundário dos esforços pela busca de economia de energia através do aumento de desempenho – foco deste trabalho – apresenta-se então os primeiros estudos acerca da possibilidade de execução de processamento do tipo *stream* em um pipeline de instruções reconfiguráveis no Javarray, aumentando dessa forma o IPC e reduzindo o impacto do consumo estático de energia.

Palavras-Chave: sistemas embarcados, Java, arquiteturas reconfiguráveis, redução de consumo de energia.

Javarray: a Reconfigurable Architecture for Performance Speedup and Energy Saving of Embedded Java Applications

ABSTRACT

Althought with a better management of the softwares' complexity, allowed by the Java language, the restrictions of energy saving, low power consumption and the need of performance imposed to the embedded systems, with special emphasis to the mobile systems, are potentialized

The popularity of the Java language in the embedded systems market is increasing as an alternative to the software compatibility necessity and the applications' complexity growth, notably at consumption electronic and industrial automation, market which is also expanding.

However, the characteristics of Java Virtual Machine, based upon a stack machine, open new possibilities to the optimization of embedded systems processing inherent to the stack machines and not yet properly explored by the actual Java processors. With the exploitation of binary translation to the Java code and the use of reconfiguration techniques, we can improve the performance with simultaneous energy savings, achieving achieving a better fit of Java applications execution to the embedded systems domain.

This work presents a coarse grain reconfigurable unit, the Javarray, to be coupled to a native execution Java microcontroller, designed to the optimized execution of the embedded systems applications more representative basic blocks. With this, we can explore ILP in a simple way and reduce the number of the executed instructions up to 42%, improving the performance up to 2.6 times and saving energy up to 64%, at the same time in which allowing for Java compatibility and, in many cases, still having less power consumption. This data refer to a set of 3 specific applications used by our research group.

The basic Javarray topology is developed from the analysis of the embedded application profiles, form which some organizational variations are explored. In special, it was designed a sequential architecture, which enables the use of pipeline techniques on the Javarray, allowing for the exploitation of coarser grains parallelism. As a secondary product of the search for the energy savings through the performance speedup – focus of this work – it is presented the first studies about the possibility of stream-based processing execution in a pipeline of reconfigurable instructions on the Javarray, this way increasing the IPC and reducing the static energy consumption impact.

Keywords: Embedded Systems, Java, Reconfigurable Architectures, Reduction of Energy Consumption.

1 INTRODUÇÃO

1.1 Motivação

Avanços tecnológicos e redução nos custos permitiram a explosão do mercado de Sistemas Embarcados – notadamente em eletrônica de consumo portátil e automação industrial. As perspectivas são de uma maior expansão do mercado e de uma crescente necessidade funcional em tais dispositivos (LEWIS, 1998), (CHRISTOFOROS, 1998).

Em consonância com o crescimento do mercado dos sistemas embarcados, as aplicações Java vêm obtendo uma popularidade cada vez maior, notadamente em telefones celulares, PDAs e pagers (VDC-CORP, internet),(MULCHANDANI, 1998). Existe uma enorme base de bibliotecas e aplicações embarcadas desenvolvidas para Java, e a tendência é de uma consolidação ainda maior da linguagem nesse domínio. Por exemplo, estima-se que mais de 74% dos telefones móveis executarão Java já em 2006 (McATEER, internet),(LAWTON, 2002).

Essa nova perspectiva no cenário da computação embarcada dá margem a um novo foco nas pesquisas da área, que devem preocupar-se ainda mais com as questões de complexidade de projeto, *time-to-market* e compatibilidade de software, além da árdua tarefa de economia de energia com simultânea necessidade de desempenho impostas pelos sistemas portáteis atuais, para citar algumas.

Nesse contexto, o Laboratório de Sistemas Embarcados - LSE (www.inf.ufrgs.br/~lse), do Instituto de Informática da Universidade Federal do Rio Grande do Sul, dentro do contexto do projeto SEEP – Sistemas Eletrônicos Embarcados Baseados em Plataforma - tem desenvolvido alguns projetos relativos ao FemtoJava (ITO, 2000) – um microcontrolador destinado a executar nativamente *bytecodes* Java para aplicações embarcadas.

Embora as tradicionais arquiteturas RISC tenham possibilitado um ganho em performance para sistemas *desktop* com técnicas avançadas como execução superescalar, especulação agressiva e escalonamento dinâmico, tais técnicas consomem muita potência (WILCOX, 1999) e não são apropriadas ao domínio dos sistemas embarcados.

As arquiteturas reconfiguráveis mostraram-se uma alternativa atrativa para a implementação de sistemas de propósito dedicado de largo espectro – ou seja, sistemas capazes de executar com desempenho próximo ao dos ASICs (Application Specific Integrated Circuits) uma grande quantidade de algoritmos próprios a algumas aplicações específicas (SILVA, 2001). Essas arquiteturas situam-se a meio-termo entre a generalidade dos processadores de propósitos gerais e o alto desempenho dos ASICs (HAUCK, 1998).

Obteve-se bastante êxito no tratamento de muitas aplicações com a utilização de arquiteturas reconfiguráveis de granularidade fina. Entretanto, os custos em área da implementação de um hardware em FPGA chegam facilmente a ser multiplicados por um fator 20 na implementação de uma lógica aleatória, podendo chegar até 100 em lógica estruturada como ALUs (EBELING, 1996). Isso torna essas arquiteturas ineficientes ao se tratar de aplicações com intensas computações aritméticas e de propósitos mais gerais.

Além disso, limitações tecnológicas como os tempos de reconfiguração do chip, complexidade da estrutura de interconexões e mapeamento de aplicações em tempo real, entre outras, e também questões relativas ao desempenho das aplicações, impuseram a pesquisa de novos métodos e deram margem à criação de novas arquiteturas em detrimento dos FPGAs.

A evolução natural das pesquisas levou ao desenvolvimento de arquiteturas menos genéricas, mas mais eficientes para as aplicações-chave em análise: arquiteturas reconfiguráveis de granularidade grossa, onde os grãos deixam de ser LUTs, e passam a se constituir de unidades funcionais mais especializadas, com ALUs ou até mesmo processadores completos; e com interconexões não mais de 1 bit, mas sim da largura de palavras inteiras, tornando mais eficiente a utilização dos recursos de roteamento (HARTENSTEIN, 2001).

As características da JVM, baseada em uma máquina de pilha, abrem possibilidades de otimização do processamento dessas aplicações embarcadas, mas que ainda não têm sido devidamente exploradas pelos processadores Java existentes – como PicoJava I (O'CONNOR, 1997), PicoJava II (SUN, 1999) e Komodo (KREUZINGER, 1999), por exemplo. Sendo a economia de energia um dos principais objetivos dos projetos de sistemas embarcados, apresentamos o Javarray – uma unidade reconfigurável de granularidade grossa a ser acoplada a um processador de execução Java nativa –, destinada à execução otimizada, por tradução binária, de *blocos de operandos* dos *blocos básicos* mais representativos de aplicações embarcadas.

Com a aplicação de tradução binária ao código Java e utilização de reconfiguração, conseguimos explorar ILP de uma maneira simples e obtivemos uma redução no número de instruções executadas, aumentando o desempenho com simultânea economia de energia, ao mesmo tempo em que se mantém compatibilidade de software com as aplicações Java e se permite uma melhor adequação dessas aplicações para o domínio dos sistemas embarcados.

1.2 Objetivos

Os principais objetivos deste trabalho são:

- obter economia no consumo de energia, em aplicações embarcadas, através do aumento de desempenho das mesmas;
- realizar a otimização de desempenho através de uma arquitetura reconfigurável voltada à execução de aplicações Java nativas;

- obter uma arquitetura reconfigurável genérica e de baixa complexidade para otimizações de aplicações embarcadas Java, com a maior simplicidade possível para o mapeamento, reconfiguração e controle de fluxo das mesmas;
- demonstrar a aplicabilidade da abordagem reconfigurável a aplicações de propósitos gerais, ampliando o escopo dos reconfiguráveis em relação às aplicações SIMD, onde estas arquiteturas já se provaram eficientes;
- demonstrar que as características da JVM, baseada em máquina de pilha, permitem uma melhor exploração da execução dos bytecodes Java do que os atuais processadores Java têm realizado.
- contribuir com uma nova arquitetura para a exploração do espaço de projeto no contexto do projeto SEEP.

1.3 Contribuições Originais e Resultados Obtidos

Este trabalho faz parte do fluxo de projeto do SEEP, proporcionando biblioteca de elementos funcionais para construção de modelos e também modelos completos de unidades funcionais a serem acopladas aos processadores já existentes pelo grupo, situando-se, portanto, na etapa de *Platform Library*, que propicia explorações arquiteturais, conforme exposto na figura 1.1 a seguir. Uma descrição detalhada do projeto e de suas etapas está disponível na página do LSE (www.inf.ufrgs.br/~lse).

Dentro do escopo do projeto SEEP existe também o projeto SASHIMI, que se traduz num ambiente de exploração de espaço de projeto de sistemas embarcados onde se determina a melhor combinação *hardware-software* para uma aplicação com restrições específicas a partir da descrição em alto nível dessa aplicação.

Esse ambiente faz uso de *plataformas* de hardware para determinar as possibilidades de exploração do projeto na parte de hardware, onde o FemtoJava constitui-se como um microcontrolador de execução nativa Java existente em diferentes versões e destinado a esse fim. Neste trabalho, incrementa-se as possibilidades de exploração do FemtoJava (ADÁRIO, 1997) com a introdução de uma Unidade Reconfigurável de Granularidade Grossa, destinada ao aumento de desempenho e redução do consumo de energia de aplicações embarcadas Java – o Javarray.

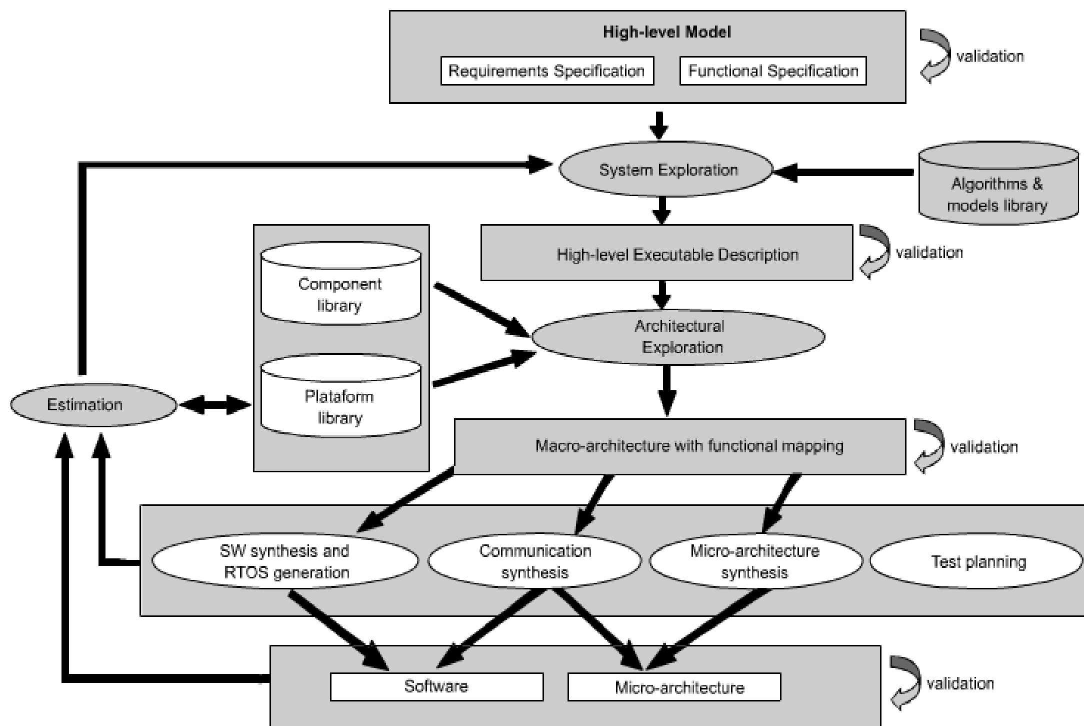


Figura 1.1: Fluxo de projeto do SEEP

Além das contribuições deste trabalho ao fluxo de projeto do SEEP, demonstra-se aqui a aplicabilidade das técnicas de reconfiguração para o domínio dos sistemas embarcados na busca por otimização de performance e economia de energia, sendo que se produz até mesmo redução simultânea da potência consumida em alguns casos.

Mostra-se que a exploração das características da Java Virtual Machine, baseada em pilha, pode economizar até 42% no número de instruções a serem executadas com a abordagem arquitetural adotada, contribuindo para o aumento de desempenho e economia de energia citados. Além de se conseguir aumentos de desempenho de 2.6 vezes e redução de consumo de energia em até 64%, este trabalho também avalia o impacto de duas abordagens distintas na criação do array reconfigurável: versão combinacional e versão seqüencial. Esses dados referem-se a um conjunto de 3 apliações analisadas, utilizadas pelo grupo do LSE.

Embora espere-se que a versão seqüencial do Javarray consuma em excesso e degrade o desempenho em relação à sua versão combinacional, nossos dados demonstram que o impacto dessa degradação não é assim tão significativo, possibilitando a utilização de técnicas de pipeline para aumento de performance. Assim, demonstra-se as possibilidades de exploração de processamento do tipo stream no Javarray, através de técnicas de pipeline de instruções reconfiguráveis, aumentando a performance com aumentos de IPC de mais de 100% e reduzindo o impacto do consumo estático de energia, no bloco reconfigurado utilizado como demonstração da técnica.

Demonstra-se também a possibilidade de otimização, através de reconfiguração, de aplicações de propósitos gerais, e não apenas aplicações SIMD para as quais os sistemas reconfiguráveis têm sido utilizados, corroborada pelas otimizações produzidas

na aplicação Crane 16bits, notadamente mais voltada para *controle* do que para *processamento* de dados. Isso se deve ao fato de o Javarray otimizar os blocos básicos mais representativos, explorando o ILP intrínseco existente nos mesmos – fato que depende muito mais da exploração das características da JVM do que da natureza da aplicação.

O projeto arquitetural do Javarray, em ambas as suas versões, se mostrou altamente escalável e de fácil gerenciamento de sua complexidade, com alta localidade de recursos de hardware.

Como resultado deste trabalho foram desenvolvidos ainda um aplicativo para análise de profiles de aplicações Java, um aplicativo para análise de dependências de blocos básicos Java, diversos componentes para a biblioteca CACO-PS (BECK, 2003), um modelo Javarray VHDL altamente parametrizável e 2 modelos arquiteturais Javarray para o CACO-PS, contribuindo para os trabalhos do LSE no âmbito do projeto SEEP.

1.4 Estrutura do Texto

Este trabalho inicia com uma análise do Estado-da-Arte, no Capítulo 2, começando por uma introdução à área da Computação Reconfigurável e seguindo com os Desafios e Tendências da computação reconfigurável e dos Sistemas Embarcados atuais. Em seguida, analisa-se algumas arquiteturas reconfiguráveis que de alguma forma serviram de inspiração para o presente trabalho e então, analisa-se algumas arquiteturas reconfiguráveis focadas nos mesmos princípios de busca de economia de energia através do aumento de desempenho. Então, no capitulo 3, detalha-se o fluxo de projeto deste trabalho, apresentando-se a proposta do mesmo e a análise de aplicações embarcadas que motivou as definições arquiteturais do Javarray, apresentadas no capítulo 4, e desenvolvida em 2 versões organizacionais. O capítulo 5 demonstra os experimentos realizados com as organizações criadas e detalha os resultados obtidos em área, consumo de energia, consumo de potência e performance, bem como apresenta uma análise sobre a possibilidade de execução *stream* no Javarray. Então, no capítulo 6, apresenta-se as conclusões deste trabalho.

Um CD foi juntado a este trabalho e contém todos os códigos, gráficos e demais anexos citados no decorrer do texto.

2 ANÁLISE DO ESTADO-DA-ARTE

2.1 Computação Reconfigurável

O mapeamento de problemas reais numa solução computacional pode ser pensado numa estrutura hardware-software onde a arquitetura de hardware é fixa, com estruturas e recursos genéricos – ou seja, um hardware de propósitos gerais (GPP) – e o software é responsável por toda a adaptabilidade necessária para mapear em hardware um conjunto de instruções que represente quaisquer funcionalidades não suportadas diretamente pelo hardware. Neste modelo, a facilidade de programação apresentada pelo hardware, ou *programabilidade*, é um requisito fundamental. Essa programabilidade pode englobar um conjunto fixo de instruções, a possibilidade de definição de pesos e parâmetros arquiteturais, a capacidade de estabelecer prioridades de execução e permite também o uso de microprogramação, por exemplo (SILVA, 2001).

A adaptação dos sistemas computacionais através de modificações do hardware, tornando-o menos genérico, portanto, está associada à execução de aplicações de propósitos mais dedicados – como os ASIC's (Application Specific Integrated Circuits) e os ASIP's (Application Specific Instruction-set Processors) – permitindo uma execução mais eficiente dessas aplicações.

A *configurabilidade* dos sistemas computacionais refere-se à capacidade de adaptabilidade da organização do sistema para se adequar à aplicação a ser executada. A configurabilidade em software, através de recursos para configuração de diferentes serviços em sistemas operacionais, é há mais tempo conhecida. A configurabilidade em hardware surgiu mais tarde, com os PAL's (Programmable Array Logic), PLA's (Programmable Logic Array) e PML's (Programmable Multilevel Logic Array), e popularizando-se com o surgimento dos FPGA's (Field Programmable Gate Arrays).

No conceito de configurabilidade de hardware está embutida a idéia de que os dispositivos não possuem, a princípio, um conjunto de instruções e nem mesmo uma funcionalidade definida. Seus recursos internos estão "incompletos" até que uma organização lhes seja atribuída.

Os dispositivos configuráveis iniciais permitiam o mapeamento de funções de alto nível através da configuração de portas lógicas (ou seja, granularidade fina). Posteriormente, começaram a surgir arquiteturas onde os elementos de processamento disponíveis passaram a ter uma granularidade maior, constituindo-se de unidades lógico-aritméticas, por exemplo. Ao mesmo tempo, começou-se a procurar soluções que permitissem a *reconfiguração* desses dispositivos em tempo de execução.

A configurabilidade do hardware permite aos sistemas computacionais um maior grau de flexibilidade que a programabilidade. Assim, as arquiteturas reconfiguráveis

situam-se a meio termo entre a generalidade das aplicações permitidas pelos processadores de propósitos gerais e a especificidade e otimização de recursos proporcionada pelos processadores de propósitos específicos, e consegue aliar a possibilidade de execução de um grande espectro de aplicações com o desempenho próximo ao obtido com os ASIC's.

Inúmeras foram as tentativas de se classificar as arquiteturas reconfiguráveis. Em cada uma delas, os autores procuraram identificar aspectos relevantes que permitam a definição de critérios de comparação entre vários sistemas reconfiguráveis já descritos na literatura científica. Os aspectos considerados nas várias propostas de classificação são:

- **Objetivo** (RADUNOVIC, 1998): funcionalidade, tolerância a falhas ou desempenho. Refere-se ao objetivo principal de desenvolvimento de uma arquitetura reconfigurável. A aceleração de aplicações, a funcionalidade e a tolerância a falhas são os principais fatores a se levar em conta.

- **Granularidade**: fina, grossa ou muito grossa. É o mais popular dos conceitos de classificação. Está relacionado ao tamanho das unidades lógicas – em quantidade de lógica (GUCCIONE, 1997), pelo tamanho da palavra de computação (HARTENSTEIN, 2001), ou pela complexidade das operações de computação que se pode realizar (GUCCIONE, 1997), (RADUNOVIC, 1998). É uma questão de grande importância, pois determina a eficiência e a adequação da arquitetura para certas classes de problemas. Quanto mais grossa a granularidade, mais específica e eficiente é a arquitetura para um dado domínio; quanto mais fina a granularidade, mais genérica é a arquitetura. Além da questão da adequação, a granularidade de uma arquitetura tem grande relação com fatores de roteamento (HARTENSTEIN, 2001). Para se conseguir uma boa taxa de *utilização* num substrato reconfigurável, uma rica rede de interconexões é necessária (DEHON, 1996). De fato, a maioria das arquiteturas usa mais de 50% da área para esse fim (LU, 1999). Em arquitetura de granularidade fina, os elementos de configuração são portas lógicas e flip-flops, operam em nível de bit (WAINGOLD, 1997), implementam funções booleanas ou máquinas de estado (Splash (BUELL, 1996)), e destinam grande parte da área disponível na configuração de roteamentos de bits individuais. Arquiteturas de granularidade grossa possuem elementos de configuração que se constituem em ULA's completas e operam em nível de palavras. A vantagem da granularidade grossa, nesse ponto, pode tornar-se mais clara ao pensarmos no exemplo de uma interconexão do tipo *crossbar*: como se está conectando *N* pe's de *B* bits, tem-se uma *crossbar* de *NxN – B bits*, e não (*NxB*)x(*NxB*), como seria com uma granularidade de 1 bit! Sistemas de granularidade muito grossa estão surgindo como uma idéia de pesquisa e referem-se a grãos compostos de processadores ou até mesmo de arrays de granularidade fina, para a explorações arquiteturais da ordem de 1 bilhão de transistores disponíveis numa pastilha de silício (WAINGOLD, 1997), (BARROSO, 2000), (SANKARALINGAM, 2003). Nesses sistemas, a flexibilidade tende a ficar ainda menor que a existente nos sistemas de granularidade grossa e se começa a confundir o conceito de reconfiguráveis com o de *multi-processadores em um único chip (CMP's)* e *networks on chip (NoC's)*.

- **Profundidade de Configuração** (WAINGOLD, 1997): refere-se à quantidade de contextos de configuração no sistema. Em sistemas com um único contexto, apenas uma configuração está residente no sistema, sendo necessária uma reconfiguração para alterar sua funcionalidade; em sistemas com múltiplos contextos, várias configurações podem estar residentes no sistema, bastando-se a alteração do contexto para acessar-se a funcionalidade desejada.

- **Acoplamento ou Integração** (RADUNOVIC, 1998): dinâmico, estático fracamente acoplado ou estático fortemente acoplado. Verifica a existência de um sistema hospedeiro. Sistema controlados por hospedeiros classificam-se em *fracamente* ou *fortemente* acoplados. Quanto maior o grau de acoplamento do sistema, menor é o overhead de comunicação, possibilitando seu uso mais freqüente pelo processador – nesse caso o sistema reconfigurável é usado como um *co-processador*. Quanto menor o acoplamento, mais independente o sistema reconfigurável se torna do hospedeiro, possuindo geralmente uma maior capacidade computacional; embora aumentando o *overhead* de comunicação, as intervenções do hospedeiro se tornam menos freqüentes, e um melhor paralelismo pode ser explorado – nesse caso, o sistema reconfigurável é utilizado mais como um *processador* (BONDALAPATI, 2002), (COMPTON, 2002). Um sistema *dinâmico* é independente do controle de um hospedeiro. A utilidade e aplicabilidade de um substrato reconfigurável é influenciada pela maneira na qual este está integrado ao *datapath.* Em (LU, 1999), eles são classificados em três classes: Como um *attached processor* (PAM (BERTIN, 1994)**,** Splash (BUELL, 1996) e DISC (WAWRSYNEK, 1996)), não se tem acesso direto ao processador – o substrato é controlado através de um barramento. Esses sistemas são fáceis de se adicionar a sistemas computacionais já existentes. Contudo, pelas restrições de largura de banda e latência impostas por esse barramento, são úteis apenas para computações com uma alta taxa de comunicação com a memória. Ou seja, são mais adequados a *stream-based functions* que requerem pouca ou nenhuma comunicação com o processador. Como um *co-processador* (Garp (HAUSER, 1997), Napa-1000 (RUPP, 1998)), existe uma baixa latência e uma grande largura de banda entre o processador e o reconfigurável – o que aumenta o número de *stream-based functions* que podem fazer uso do reconfigurável. Já como uma *unidade funcional* dentro do processador principal (P-RISC (RAZDAN, 1994), Chimaera (HAUCK, 1997), OneChip (WITTIG, 1996)), permite-se que instruções customizadas sejam executadas. A unidade reconfigurável faz parte do *datapath* do processador e tem acesso aos registradores. Entretanto, essas implementações restringem a aplicabilidade da unidade reconfigurável desabilitando, em alguns casos, o acesso direto à memória, essencialmente eliminando sua utilidade para processamento de *stream-based functions.* A figura 2.1 exemplifica esses conceitos, mostrando as possíveis localizações para um substrato reconfigurável na hierarquia de memória.

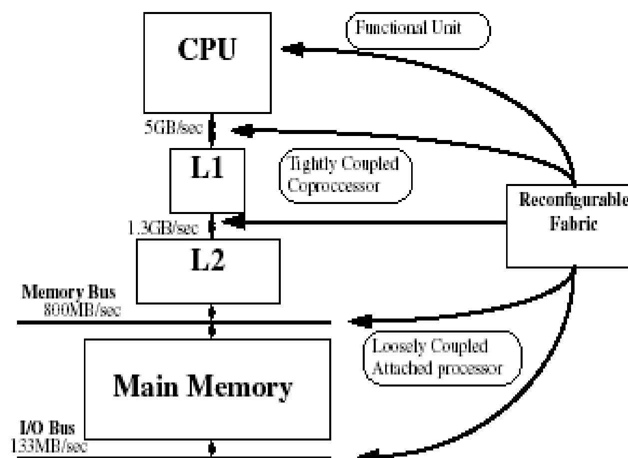


Figura 2.1: Possíveis localizações para um substrato reconfigurável (ACHUTHARAMAN, 2003).

- **Modelo de Comunicação com Hospedeiro** (PAGE, 1996): co-processamento, cliente-servidor, chamada de procedimento ou execução paralela. O critério de comunicação com o hospedeiro leva em conta a forma como um algoritmo referencia e executa as operações implementadas nas unidades reconfiguráveis da arquitetura.

- **Interconexões** (HARTENSTEIN, 2001), (RADUNOVIC, 1998): reconfigurável, fixa, mesh, array linear, crossbar. É um dos conceitos fundamentais e, como já dito, está intimamente relacionado ao conceito de *granularidade*. Geralmente, a estrutura de interconexões impõe uma organização aos grãos da arquitetura – num vetor linear, numa estrutura de array ou qualquer outra. Além disso, contribui significativamente para a área total do componente, no sentido de que para granularidades finas a estrutura de interconexões tende a ser mais complexa.

- **Modelo de Execução** (PAGE, 1996): hardware puro, microprocessador de aplicação específica, uso seqüencial, uso múltiplo simultâneo ou uso sob demanda. Nesses modelos, se estuda os modos de implementação e execução dos algoritmos. No *hardware puro*, o algoritmo é convertido numa configuração única sem que haja a necessidade de elementos externos. Já na classificação de *uso sob demanda*, fica evidenciada a necessidade de um gerenciador de instruções configuradas externo – que no caso seria um sistema hospedeiro.

- **Configurabilidade** (ADÁRIO, 1997): projeto estático, reconfiguração estática ou reconfiguração dinâmica. Nesta proposta se analisa apenas o número de configurações possíveis, o momento da reconfiguração e o modo de programação. O projeto estático considera o uso de dispositivos programáveis para realizar uma configuração única que não poderá ser alterada. As

classificações de *reconfiguração dinâmica* ou *estática* consideram a existência de várias configurações possíveis que podem ser executadas em tempo de execução ou ao término de um processamento.

- **Reconfigurabilidade** (WAINGOLD, 1997): refere-se à capacidade de um sistema de sobrepor uma reconfiguração com a execução de um processamento. Está relacionado ao conceito de *profundidade da* configuração. Em sistemas estáticos, deve-se esperar a conclusão de uma computação para a carga de um novo contexto de configuração; em sistemas dinâmicos, o processo de configuração é concorrente à execução de uma outra configuração.

A despeito dos inúmero critérios de classificação que já foram propostos, o domínio das *arquiteturas reconfiguráveis* é ainda recente demais para ser classificado de forma mais segura (SILVA, 2001). A ortogonalidade de muitas das propostas reflete-se no fato de ainda não haver emergido uma plataforma padrão, amplamente aceita, com paradigmas de programação e exploração adequados e genéricos para um domínio amplo o suficiente de aplicações. As soluções existentes referem-se a aplicações especificas, com algoritmos, técnicas e métodos muito particulares ao contexto de cada arquitetura. O “Santo Graal” da Computação Reconfigurável ainda não foi encontrado... São justamente as dificuldades impostas a esses novos sistemas computacionais os focos atuais de pesquisa que se constituem nos *Desafios e Tendências* expostos na seção a seguir.

2.2 Desafios e Tendências

Para que a computação reconfigurável possa ser utilizada de uma forma mais sistemática para domínios de aplicações mais genéricos, diversas questões devem ser resolvidas ou melhoradas.

Ao se situar tais arquiteturas numa projeção futura de sua utilização é necessário que estudemos o contexto desse uso futuro, sendo importante, então, que possamos avaliar adequadamente as tendências que se delineiam.

Como se tem a pretensão de utilização da computação reconfigurável no desenvolvimento de sistemas embarcados, os desafios e tendências desse domínio também devem ser abordados.

Nesta seção, alguns dos principais conceitos em estudo e tendências de desenvolvimento serão citados, tanto na área da *computação reconfigurável* quanto na área dos *sistemas embarcados.*

Aplicações

Como o previsto, as aplicações desta década evoluíram do domínio das aplicações científicas e de escritório para o domínio das aplicações multimídia e processamento de sinais (CONTE, 1997). O surgimento de processamento vetorial em processadores de propósitos gerais já é prova desse fato.

A adequação dos reconfiguráveis a esse tipo de aplicação já se mostrou interessante, mas há uma contínua existência da necessidade de processamento

seqüencial e de aplicações orientadas ao controle, para os quais os *processadores de propósitos gerais* continuam melhores. Eis uma necessidade para o uso efetivo dos reconfiguráveis: a obtenção de um desempenho razoável no processamento de tarefas seqüenciais.

Complexidade

A crescente complexidade das etapas de projeto, verificação e teste, e a evolução de questões como compatibilidade *para trás*, *time-to-market* e gerenciamento de equipes de engenheiros (projetos atuais levam cerca de quatro anos e algumas poucas centenas de engenheiros (CHRISTOFOROS, 1998)) é uma questão preocupante, ainda mais quando se estima uma disponibilidade de bilhão de transistores por chip dentro de poucos anos.

Técnicas simples e projetos mais repetitivos e modulares – como os reconfiguráveis – se mostram mais promissores, nesse sentido, do que outras abordagens, como processadores multithreading e processadores superescalares bem largos (CHRISTOFOROS, 1998).

Outra tendência é que "questões de hardware" sejam cada vez mais resolvidas "por software" – como no processador Crusoe da Transmeta há alguns anos atrás, com técnicas de *binary translation* – permitindo a correção de problemas de projeto e a própria evolução dos sistemas e seus padrões após a fabricação dos chips. A tendência da evolução dos sistemas após a fabricação é perceptível nas já comuns atualizações de software pela internet, e é bem possível que com os reconfiguráveis avancem para o domínio do hardware. Nesse sentido, a tendência é de que "o hardware vire software".

Uma possibilidade existente com essa tendência é de que os *softwares* passem a ser mais complexos, o que poderia sugerir apenas uma mudança no domínio da complexidade, mas não necessariamente na redução dela com o uso de reconfiguráveis.

Quanto aos sistemas embarcados, é crescente a utilização de SOC's (*systems on chip*), que realizam a reutilização de módulos através da integração de componentes num sistema único, buscando a redução da complexidade e diminuição do *time-to-market*. Esses sistemas têm sido cada vez mais *configuráveis* (parametrizáveis), e muitos começam também a incluir hardware *reconfigurável* em seu projeto.

Código

Para qualquer nova arquitetura receber uma grande aceitação, é necessário que ela possa executar um amplo espectro de software. Arquiteturas que rodem os softwares existentes atualmente possuem uma grande vantagem nesse ponto (CHRISTOFOROS, 1998).

Muitas das arquiteturas reconfiguráveis propostas atualmente sofrem da problemática do software, onde necessitam de novos compiladores e bibliotecas, além de técnicas de roteamento, mapeamento e escalonamento de tempo real para torná-las arquiteturas usáveis.

O crescimento de Java como linguagem e da Internet como plataforma de comunicação – à qual Java está muito bem integrada – sugere uma boa oportunidade às arquiteturas integradas a essa linguagem.

Mapeamento da Aplicação (MORENO, 2003)

O mapeamento de aplicações para os substratos reconfiguráveis tem sido um problema relevante. Muitos avanços no campo dos compiladores e das linguagens de programação são ainda necessários.

Nesse aspecto, tem-se tentado explorar a geração de hardware a partir de linguagens de alto nível, combinadas com técnicas de síntese de alto nível.

Outro ponto importante é a identificação da possibilidade de particionamento hardware/software, onde apenas parte da aplicação – a mais freqüentemente utilizada – é mapeada para o substrato.

Compiladores

As dificuldades encontradas para os compiladores em torno de tecnologias de processamento vetorial, como a tecnologia MMX, com *subword execution*, também são válidas para os reconfiguráveis. Os compiladores idealmente deveriam ser capazes de identificar paralelismo nos dados (CONTE, 1997) e livrar o programador da tarefa de explorar propositadamente esse paralelismo.

Ainda em relação aos compiladores, tem-se atenção crescente às técnicas de *binary translation* e otimização dinâmica de código (CIFUENTES, 2002).

Memory Wall (MORENO, 2003)

As arquiteturas reconfiguráveis reduzem a necessidade de acesso à memória para a busca de instruções; porém, necessitam de uma maior largura de banda para o acesso aos dados. Assim, as limitações de comunicação entre a memória e o substrato reconfigurável impõem limites na capacidade de processamento do mesmo.

Reconfiguração Dinâmica (MORENO, 2003)

O tempo de execução de uma aplicação engloba também o tempo de configuração da mesma no substrato reconfigurável. Para tanto, em sistemas de reconfiguração dinâmica, necessita-se que o processo seja rápido o suficiente para não cancelar os ganhos obtidos com a especialização do hardware.

Nesse sentido, tem-se estudado técnicas de multi-contextos (DEHON, 1996), reconfiguração parcial (CADAMBI, 1998), (SHIRAZI, 1997), busca antecipada de configuração (HAUCK, 1998), compressão de configurações (HAUCK, 1998b), memórias cache de reconfiguração (LI, 2000) e técnicas que limitem a quantidade de informação que precisa ser modificada durante as reconfigurações (SHIRAZI, 1998), (ADÁRIO, 2001).

Sistemas Embarcados e Computação Móvel Pessoal

As aplicações que têm dominado o modelo de computação na última década (uniprocessador em *desktops* para processamento técnico e científico, e multiprocessador em servidores para processamento de transações e sistemas de arquivos) serão suplantadas em quantidade por aplicações de reconhecimento de voz e comunicação, portáteis e dependentes de baterias, e que dêem suporte a multimídia e processamento de sinais (CHRISTOFOROS, 1998), decorrente do crescimento vertiginoso da *computação móvel pessoal.*

Com essa mudança de paradigmas, as prioridades de tais sistemas embarcados, em particular economia de energia, baixo consumo de potência, restrições quanto à capacidade de processamento, necessidade de uso eficiente das memórias e compatibilidade de software se potencializam ainda mais.

As aplicações que os processadores de sistemas embarcados deverão executar possuirão as características de resposta em tempo-real, processamento de streams, paralelismo de granularidade fina (como processamento de imagem), paralelismo de granularidade grossa, alta localidade de instruções, grande largura de banda com a memória, e grande largura de banda para comunicação em rede e I/O.

O tempo e o custo dos projetos e desenvolvimento de software também representam fatores importantes. A crescente popularidade da linguagem Java nesse domínio difunde ainda mais o paradigma da orientação a objetos aos sistemas embarcados, ajudando a gerenciar esse custo e a aumentar a compatibilidade dos sistemas, trazendo porém novos desafios a serem tratados.

2.3 Arquiteturas Reconfiguráveis

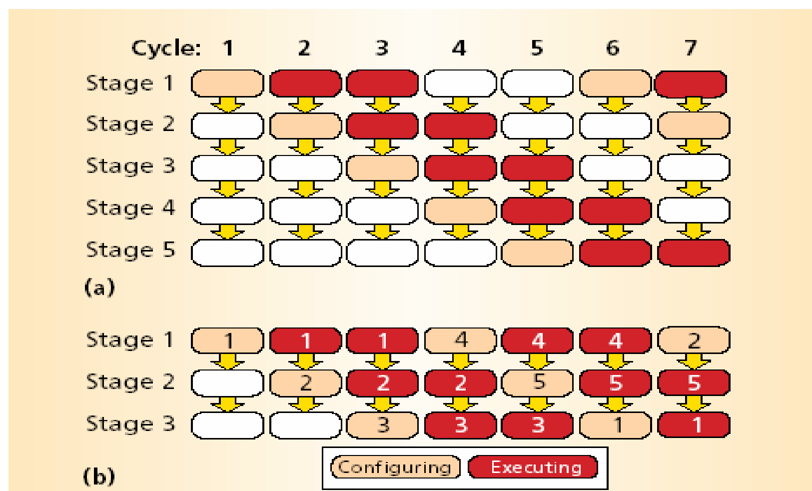
Esta seção descreve algumas características interessantes de uma gama diversificada de arquiteturas reconfiguráveis, que englobam diferentes aspectos das classificações apresentadas na seção anterior.

Após, num resumo das análises, as idéias mais originais dessas arquiteturas são resumidas, alguns resultados interessantes obtidos por elas são mostrados e é buscado um paralelo com sua aplicabilidade em Sistemas Embarcados que influenciam o projeto do Javarray em seções adiante.

2.3.1 PipeRench

O PipeRench é fundamentado no princípio de que *"future computing workloads will emphasize an architecture's ability to perform relatively simple calculations on massive quantities of mixed-width data."* (GOLDSTEIN, 1999). Logo, claramente, propõe-se a aplicações do tipo SIMD e baseada em *streams.*

Como o próprio nome já sugere, a abordagem adotada nessa arquitetura volta-se à exploração de técnicas de pipeline. Quando não há recursos suficientes para se configurar uma função completa no substrato, utiliza-se a técnica de reconfiguração *pipeline* (GOLDSTEIN, 1999)**,** (GOLDSTEIN, 2003) onde se virtualiza o pipeline através da utilização do mesmo substrato com uma configuração diferente no tempo, obtendo-se um pipeline temporal (figura 2.2):



Nota: Os números em (b) são referentes aos estágios reais na virtualização

Figura 2.2: Pipeline temporal: (a) corresponde ao pipeline real; (b) pipeline virtual (GOLDSTEIN, 2003.

Conforme a figura 2.3, a arquitetura do PipeRench apresenta as seguintes características:

- conjunto de estágios de pipeline físico, chamados *stripes*;
- cada *stripe* é composto de interconexões e *pe's*, e cada *pe* possui registradores e ula;
- a ula é uma *look-up table* com circuitos extras para *carry-chains* e *zero-detection*, entre outros.
- pe's têm acesso a um barramento global de I/O;
- pe pode acessar operandos de registradores de saída das pe's da *stripe* anterior e saídas *registradas* ou não da própria *stripe*;
- não existe barramento que retorne para um *stripe* anterior, tornando *feedback loops* impossíveis (esta característica está de acordo com os objetivos da arquitetura);
- qualquer *feedback* deve situar-se no máximo em 1 stripe;
- entradas e saídas no substrato utilizam o barramento global para alcançar os seus destinos;
- os *carrys* podem ser concatenados para se construir ulas maiores;
- interconexão é *full-crossbar*; caro em área, mas simples para o compilador;
- pe's possuem um *registrador de passagem*, utilizado para realizar uma comunicação entre a mesma pe em configurações diferentes, em tempos diferentes de computação;
- 4 barramentos: 2 para *store* e *restore* de *stripe states* durante a virtualização e 2 para entradas e saídas;

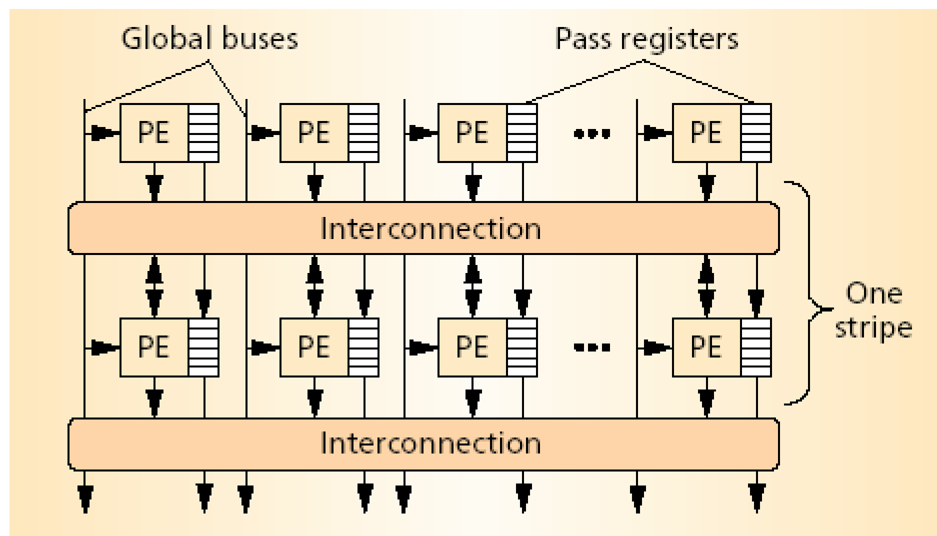


Figura 2.3: Arquitetura do Piperench (GOLDSTEIN, 2003)

2.3.2 MorphoSys

O MorphoSys (LU, 1999) visa aplicações com alto paralelismo, alta regularidade, granularidade no nível de palavras e computação intensiva, do tipo: compressão de vídeo, processamento de imagens, multimídia e segurança de dados. Contudo, é flexível o suficiente para dar suporte a aplicações irregulares e no nível de bits.

O MorphoSys compreende 5 componentes vistos na figura 2.4: um *core processor* semelhante ao MIPS; um controlador DMA, um array reconfigurável (RC Array), um Frame Buffer para acesso a dados, e uma Context Memory que armazena configurações do RC Array.

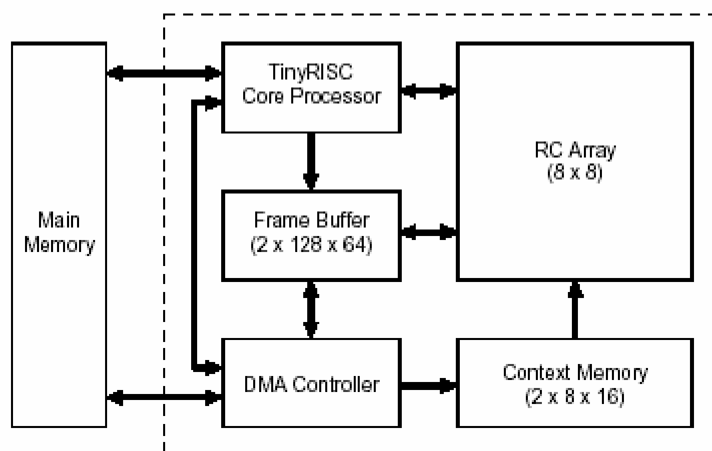


Figura 2.4:Estrutura geral do MorphoSys (LU, 1999).

O processador TinyRISC é composto de pipeline de 4 estágios, registradores de 32 bits e 3 unidades funcionais que compreendem, cada qual: uma ula 32 bits, uma *shift unit* 32 bits e uma unidade de memória, além de uma cache de dados *on-chip*. Esse processador possui um conjunto de instruções aumentado para dar suporte a instruções específicas de controle dos outros componentes do sistema MorphoSys: instruções de controle do DMA e instruções de controle do RC Array.

Quanto ao modelo de execução do MorphoSys, se baseia no particionamento de aplicações entre tarefas paralelas e seqüenciais. As tarefas seqüenciais são processadas pelo TinyRISC, e as paralelas são configuradas no RC Array. O TinyRISC inicia todas as transferências de dados envolvendo aplicação e dados de configuração. Enquanto o RC Array realiza a execução sobre os dados de um conjunto do Frame Buffer, novos dados podem ser salvos no outro conjunto ou a Context Memory pode receber novos contextos. O TinyRISC controla também o modo de operação do *broadcast* de contextos, provê sinais e endereços para a Context Memory, controla o Frame Buffer e também o DMA.

Como resumo de algumas características, o MorphoSys é um sistema dinamicamente reconfigurável, de granularidade grossa, fortemente acoplado e com grande profundidade de programabilidade, já que possui 32 contextos e 2 modos de *broadcast* dos mesmos aos pe's do RC Array.

As aplicações utilizadas pelos autores, e onde o MorphoSys já se provou interessante, foram processamento de imagem (DCT/IDCT) e encriptação de dados (IDEA).

Quanto ao RC Array, trata-se de um array 8x8 de células reconfiguráveis, com 3 níveis de interconexão, conforme a figura 2.5.

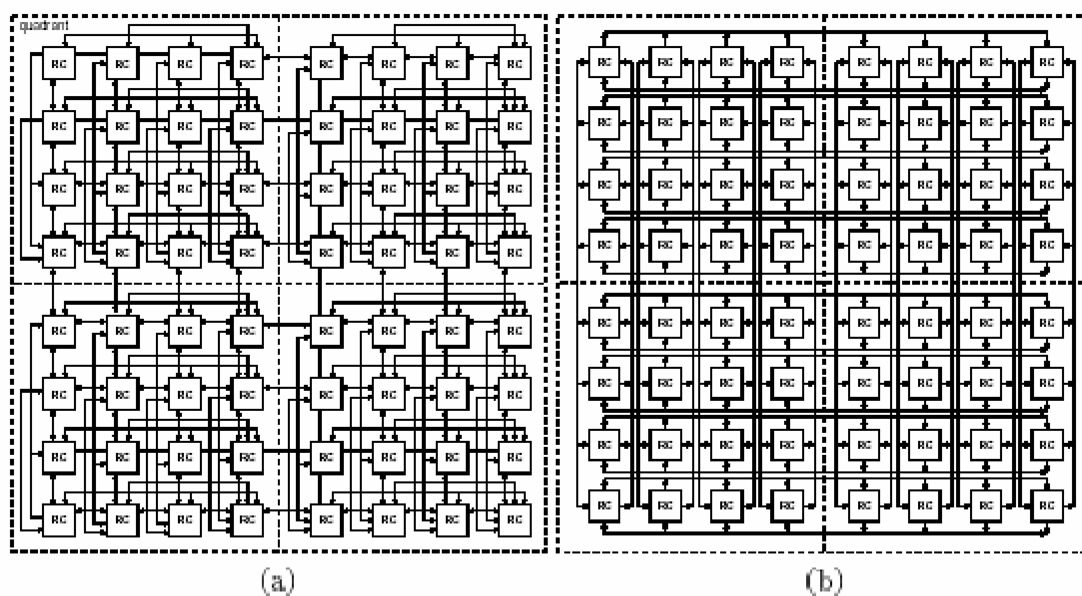


Figura 2.5: (a) Nível de conectividade *local* e *linha e colunas*; (b) nível de conectividade de *entre quadrantes* (LU, 1999).

Cada célula reconfigurável é composta de (figura 2.6):

- alu-multiplier com 4 entradas, capaz de realizar multiplica-acumula em um único ciclo;
- shift unit;
- multiplexadores de entrada;
- banco de 4 registradores de 16 bits cada;
- um registrador de contexto.

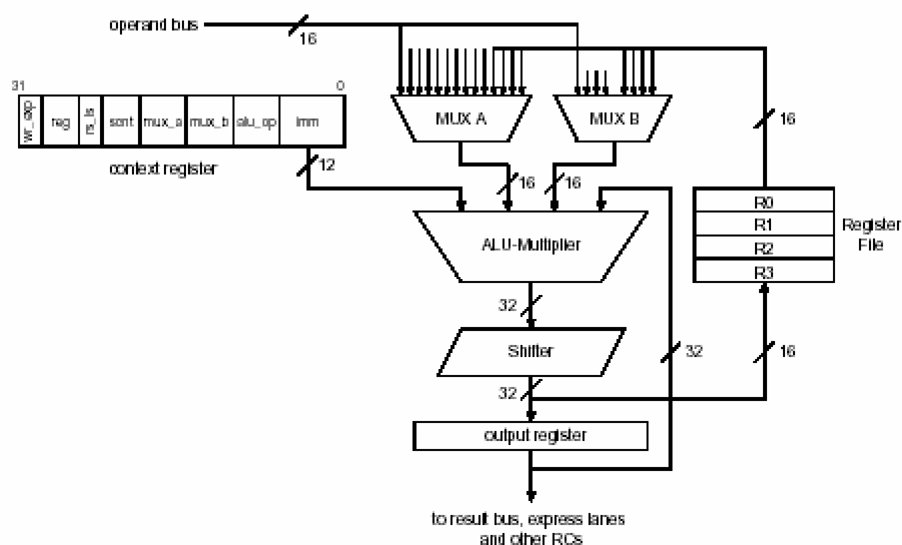


Figura 2.6: Célula reconfigurável do MorphoSys (LU, 1999).

A Context Memory é composta de 2 *context blocks* que contêm cada um 8 *context sets*, cada qual com 16 *context words.*

Como o foco do RC array é em aplicações com alto paralelismo e regularidade, as *context words* são propagadas por colunas e linhas. Cada *context block* possui 6 conjuntos de contexto e cada conjunto está associado a uma coluna (ou linha) específica do RC.

O Frame Buffer é uma memória interna de dados logicamente organizada em dois conjuntos, chamados Set 0 e Set 1. Cada conjunto é subdividido em dois bancos, A e B. Cada banco possui 64 colunas de 8 bytes (logo, o Frame Buffer inteiro possui 128 x 16 bytes). Um barramento de 128 bits é utilizado para transferir dados do Frame Buffer para o RC. As células numa coluna compartilha o mesmo segmento de 16 bits do barramento. No protótipo analisado em (LU, 1999), o barramento opera em modo entrelaçado. Outro barramento de 128 bits é utilizado para transferir dados no sentido RC-Frame Buffer. Em versões futuras ao trabalho analisado, os autores pretendem fazer o modo de operação do barramento ser reconfigurável, para que possa melhor se adequar a mais classes de aplicações.

O Controlador DMA é o responsável pela transferência de dados entre o Frame Buffer e a Memória Principal. Ele também é o encarregado da carga de contextos na

Context Memory. O TinyRISC utiliza instruções DMA para especificar os parâmetros necessários de transferência de dados e contextos para o Controlador DMA.

2.3.3 Rapid

O nome RaPiD (EBELING, 1996) provém de Reconfigurable Pipelined Datapaths – no mesmo conceito que o PipeRench. Seu foco está em aplicações altamente regulares e de computação intensiva, como DSP's e algoritmos sistólicos.

Os *datapaths* construídos no RaPiD são arrays lineares de unidades funcionais (módulos de memória, registradores, ula's e multiplicadores) que se comunicam principalmente através de conexões locais (*nearest-neighbor).* Esses *datapaths* podem ser configurados para aplicações sistólicas ou também como pipelines profundos, com diferentes computações em seus diferentes estágios, para diferentes tempos de computação.

RaPiD limita o paralelismo das aplicações ao máximo de duas leituras e uma escrita por ciclo – o que mostrou-se ser necessário e também suficiente para as aplicações que foram testadas (EBELING, 1996). Mesmo para essas restrições, uma arquitetura com memória de alto desempenho é necessária.

O exemplo de avaliação proposto pelos autores em (EBELING, 1996), e onde a arquitetura RaPiD se mostrou eficiente, trata do problema da *multiplicação de matrizes*.

Implementações da arquitetura RaPiD podem sofrer variações decorrentes de diversos parâmetros, incluindo tamanho e formato dos dados, número e tipo de unidades funcionais, e número e configuração dos barramentos. A arquitetura da versão RaPiD-1, tratada pelos autores, é basicamente um array linear que pode conter entre 8 e 32 células reconfiguráveis.

Cada célula (figura 2.7) possui:

- um multiplicador inteiro;
- três ula's inteiras;
- seis registradores de propósitos gerais;
- três pequenas memórias locais;
- um conjunto de barramentos segmentados;
- multiplexadores e tri-states para tratar os barramentos;
- e, opcionalmente, um registrador de pipeline.

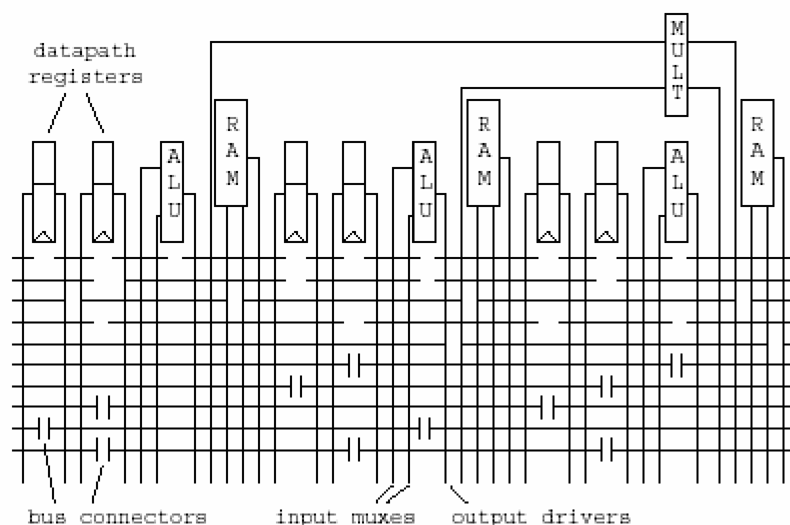


Figura 2.7: Célula básica do RaPiD-1 (EBELING, 1996).

RaPiD opera dados de 16 bit em ponto-fixo, com ou sem sinal, que são mantidos por *shifters* nos multiplicadores – diferentes representações de ponto-fixo podem ser usadas na mesma aplicação através da apropriada configuração desses *shifters*.

O *datapath* não gera nenhuma exceção durante a computação, mas as incorpora nos dados gerados, tendo-se um *tag bit* que indica *overflow* junto aos dados produzidos.

ULA's adjacentes podem ser combinadas para processar dados de 32 bits.

I/O é processado através de *streams* nas extremidades do array linear. Cada *stream* é composto de FIFO operando assincronamente (o datapath sofre um *stall* automático quando as FIFOs estão vazias ou cheias, até que a FIFO esteja pronta novamente), está associado à blocos predeterminados de memória, e é responsável pela geração dos endereços necessários.

2.3.4 RawMachine

A Raw Machine (WAINGOLD, 1997) foi pensada com visão no cenário de um futuro próximo onde se tenha disponível 1 bilhão de transistores num chip. Seus esforços concentram-se na busca por três soluções fundamentais: a necessidade em manter as interconexões internas curtas, para que o *clock* seja escalável com o aumento de área do chip; a necessidade de verificação do projeto de um chip de tal tamanho; e a mudança dos *workloads* de aplicações, que irão enfatizar aplicações multimídia baseadas em *streams*.

Para tratar desses problemas, a idéia proposta é a de se utilizar um hardware simples e replicado (um conjunto de *tiles*), que exponha seus detalhes arquiteturais para o compilador, de forma que tendo-se apenas um pequeno conjunto de mecanismos implementados em hardware, o software seja o principal responsável pela melhor alocação de recursos para cada aplicação.

Dado seu alto grau de paralelismo e grande rede (estática) de interconexões, a Raw Machine também é particularmente eficiente para aplicações científicas e outras

aplicações que requeiram grande largura de banda para comunicações de granularidade fina.

Além do fato de se constituir num processador completo, independente de hospedeiro, as principais características da Raw Machine são a importância do compilador em seu projeto e a natureza estática dessa compilação. Enquanto que no passado era mais complicado compilar-se estaticamente para uma arquitetura como a Raw, os *workloads* baseados em *streams* podem se beneficiar bastante dessa característica.

Quanto à compilação, cada *tile* possui seu próprio fluxo de instruções, e uma *thread* ou mais podem ser mapeadas diretamente para um *tile*. Diferentemente das arquiteturas VLIW que extraem o paralelismo de um fluxo de instruções seqüenciais, a Raw Machine enxerga o conjunto de *tiles* como uma coleção de unidades funcionais para explorar ILP.

Além da compilação diretamente a partir de uma linguagem de alto nível, fazem-se necessárias as etapas estáticas de particionamento, posicionamento, roteamento e escalonamento. Ainda surge a questão da seleção dinâmica de configurações para carregar nos *tiles*, e principalmente a questão dos *eventos dinâmicos* – eventos que não podem ser determinados estaticamente. Para esses eventos, os autores propõe um conjunto de alternativas a se explorar (WAINGOLD, 1997).

Essas questões não estão todas resolvidas. A Raw Machine é uma arquitetura em desenvolvimento, pensando-se num futuro onde de fato se tenha a capacidade de transistores por chip para a qual ela foi projetada.

Um processador Raw é constituído de um conjunto de *tiles* interconectados por uma rede de interconexões programável (figura 2.8), cada qual possuindo o seu próprio *stream* de instruções, e contendo:

- memória de instruções;
- memória de dados;
- ula;
- registradores;
- lógica configurável;
- um switch programável, que comporta tanto roteamento estático definido pelo compilador, quanto um roteamento dinâmico.

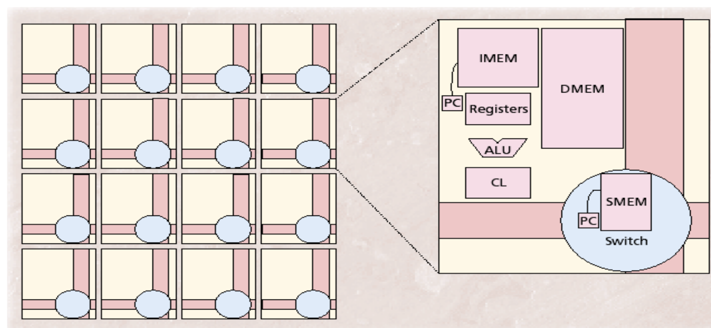


Figura 2.8: *Tiles* da *Raw Machine* organizados em uma rede de interconexões (WAINGOLD, 1997).

A comunicação inter-tile possui latência de acesso a registradores e um escalonamento estático garante que os operadores estarão disponíveis quando necessários, eliminando a necessidade de sincronização explícita.

Além disso, cada tile suporta operações multigranulares (bit, byte e word-level), e a lógica configurável pode ser usada para criar instruções únicas e específicas para cada aplicação.

Diferentemente dos superescalares modernos, a Raw não implementa estrutura especializadas em hardware como renomeação de registradores e despacho dinâmico de instruções, mantendo sempre o foco em se manter um tamanho pequeno para o *tile* e maximizar o número de *tiles* por chip.

Estima-se que um chip convencional de 1 bilhão de transistores seria capaz de dar suporte a 128 *tiles*, onde cada um teria uma distribuição da seguinte forma, considerando que interconexões consumam 30% da área do chip:

- 5 milhões de transistores para memória (uma Memória de Instruções de 16 Kb, uma Memória de Instruções para o Switch de 16 Kb, e uma Memória de Dados de 32 Kb);
- 2 milhões para controle do *datapath* do tipo *pipeline*;

2.3.5 KressArray

Orientado a aplicações multimídia, e sabendo-se da crescente demanda de tais aplicações em sistemas embarcados como *handhelds* e também da necessidade da preocupação com o consumo de energia de tais sistemas, o KressArray sugere como ineficientes os sistemas baseados em granularidade fina – mais explicitamente os FPGA's – e aponta-se como uma solução capaz de prover a capacidade necessária de processamento, com economia de energia e área não possíveis de serem obtidos através dos FPGA's (HARTENSTEIN, 1998).

O KressArray é na verdade uma família de processadores reconfiguráveis, com algumas variações arquiteturais específicas, mas que mantêm a mesma essência. No presente caso estamos tratando do KressArray-III.

O KressArray necessita de um processador hospedeiro para gerar as configurações, e é projetado para ser escalável, de forma que se possa integrar diversos chips idênticos num mesmo sistema reconfigurável.

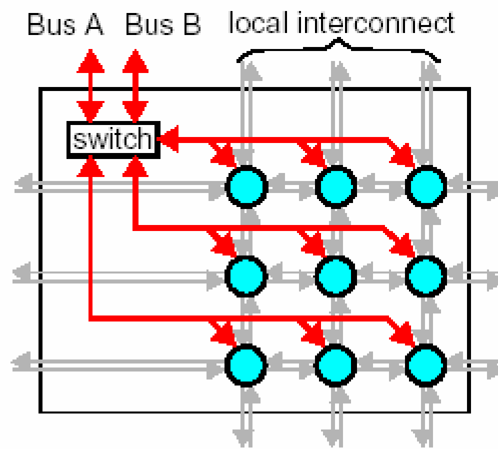


Figura 2.9: KressArray - I/O para as rDPU's (HARTENSTEIN, 1998).

O KressArray é composto por uma rede do tipo *mesh* de elementos de processamento chamados rDPU's – *reconfigurable DataPath Unit*. Normalmente, aplicações necessitam de dados de entrada e saída que não estejam necessariamente nas rDPU's das bordas do chip. Por esse motivo, uma rede hierárquica – para evitar gargalos no barramento – de roteamento global foi criada para prover esse mecanismo de I/O para qualquer rDPU onde fosse necessário (figura 2.9).

Não existem facilidades de roteamento, há apenas comunicação entre os 4 vizinhos mais próximos, economizando-se na área que seria gasta numa rede de interconexões. Pelo mesmo motivo, muito menos dados de configuração são necessários para se estabelecer um datapath do que seria o caso com a existência de uma rede de interconexões, e isso também ajuda a reduzir o próprio tempo de reconfiguração.

Dadas as restrições de uma comunicação global, o problema do mapeamento de uma aplicação é reduzido a uma questão de posicionamento, a menos de necessidades eventuais de uma comunicação mais remota, onde os PE's podem ser configurados para realizar roteamento (figura 2.10).

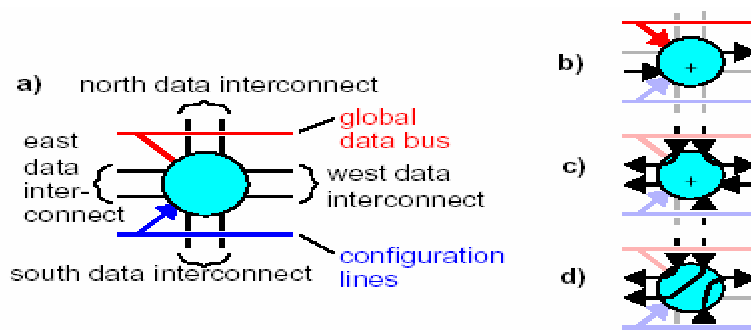


Figura 2.10: Roteamento das rDPU's: (a) barramentos de I/O e configuração; (b) rDPU configurada como operador aritmético; (c) configurada como operador aritmético e como roteador; (d) rDPU configurada exclusivamente como roteador (HARTENSTEIN, 1998).

A configuração dos roteamentos e operações aritméticas de uma rDPU é armazenada em uma memória de configuração de 4 níveis, capaz de armazenar 4 contextos, através dos quais se realiza uma reconfiguração rápida através da simples troca de contexto. Enquanto um dos quatro contextos está ativo, a configuração dos demais é independente, e pode ser realizada em paralelo, tanto com a computação quanto com as demais configurações dos contextos.

As memórias de configuração são escritas por um processador hospedeiro através de um barramento de configuração. No processo de reconfiguração de um contexto, todas as rDPU's estão conectadas ao barramento de configuração, que identifica únicamente uma rDPU por um endereço de 32 bits e logo em seguida grava uma palavra de 32 bits de configuração.

2.3.6 Xtreme

Devido ao crescimento exponencial dos custos das máscaras CMOS, um aspecto essencial para a indústria é a adaptabilidade dos SoC's. O Xtreme (BECKER, 2003) é uma arquitetura reconfigurável proposta para ser integrada a uma plataforma de SoC configurável, e também possui ele próprio características configuráveis. O objetivo global é de se utilizar hardware reconfigurável reusável em diferentes granularidades dentro dos SoC's, de forma que esses possam ser melhor configurados para aplicações específicas.

Trata-se de uma arquitetura que introduz o conceito de *sequenciamento de configurações*, ao invés do *sequenciamento de instruções*, visando aplicações de alta performance desde *processamento embarcado de sinais* até o *co-processamento* em diferentes ambientes de aplicações do tipo DSP.

Outro foco dessa arquitetura está na adaptabilidade às necessidades futuras de padrões industriais e modos de operação variáveis, como 3G, UMTS, QoS, e outros (BECKER, 2003).

A arquitetura proposta dá suporte à execução de múltiplos fluxos de dados em paralelo.

Ela ainda foi projetada para ser escalável, já que é pensada para ser usada no contexto de SoC's configuráveis, e distribui a tarefa de configuração das aplicações em uma hierarquia na forma de árvore de PAC's – que são um conjunto entre um *gerenciador de configurações* e um *array de elementos reconfiguráveis*.

O número de PAC's pode ser configurado por chip, especificamente de acordo com as necessidades do SoC ao qual o Xtreme será integrado. A PAC raiz dessa hierarquia é chamada de *supervisora*, e geralmente conectada a uma RAM externa ou global.

O conceito básico consiste em substituir um *stream* de instruções do tipo Von-Neumann por um sequenciamento automático de configurações, processando-se *streams de dados* ao invés de *palavras* únicas.

Os autores argumentam que a regularidade do Xtreme facilita a extração de *ILP* e *pipeline* implicitamente contidos nos algoritmos das aplicações. Dizem, ainda, ter o Xtreme muitas similaridades com o KressArray e a Raw Machine, projetados todos para o tratamento de aplicações baseadas em *streams* (BECKER, 2003).

A arquitetura do SoC no qual o Xtreme está inserido é composta de (figura 2.11):

- um core Xtreme reconfigurável;
- um microcontrolador Leon, RISC com os 5 estágios tradicionais;
- vários tipos de memória RAM, usados para armazenar programas LEON, dados para o Xtreme e configurações para o Xtreme;
- um barramento AHB, da ARM.

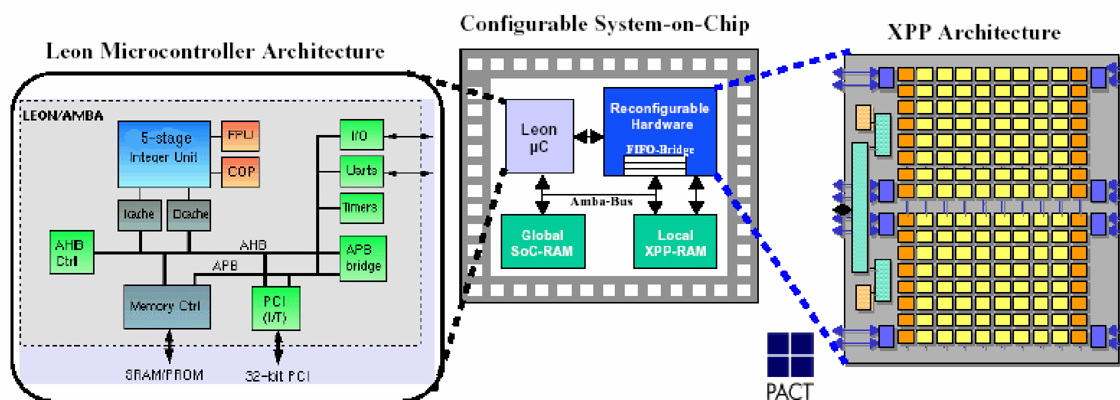


Figura 2.11: Arquitetura geral do SOC configurável Xtreme/Leon (BECKER, 2003).

Para um acoplamento eficiente entre o Xtreme e o AHB, projetou-se uma *AHB-bridge* que conecta a interface de I/O do lado do Xtreme ao AHB via um único módulo.

A arquitetura do Xtreme propriamente dita é baseada num array hierárquico de elementos reconfiguráveis de granularidade grossa, chamados PAE's – *Processing Array Elements* – e em uma rede de interconexões com comunicação orientada a *pacotes*.

As principais peculiaridades do Xtreme encontram-se no seu mecanismo de reconfiguração em *run-time* e no tratamento automatizado de pacotes de comunicação.

Como a configuração está distribuída em diversos Configuration Managers dentro do array, PAE's podem ser rapidamente configurados em paralelo enquanto PAE's vizinhos processam dados. Diversas aplicações podem ser configuradas e rodar independentemente em diversas partes do array.

Além da configuração externa, controlada pelo LEON, *pacotes de eventos* especiais permitem a implementação de projetos auto-reconfiguráveis, e também a implementação de computação condicional.

Um chip Xtreme é composto de um ou mais PAC (*Processing Array Cluster*) – que é uma coleção de PAE's (figura 2.12), as células básicas da arquitetura – cada qual associado a um CM (*Configuration Manager*), responsável pela escrita de dados de configuração nos objetos do PAC. Dispositivos com múltiplos PAC's formam uma hierarquia de CM's, em forma de árvore. Os PAC's podem ser compostos por um número parametrizado de PAE's, de acordo com as aplicações do SoC, mas as configurações comuns são de 4x4 ou 8x8.

A arquitetura do Xtreme foi ainda projetada para que se possa cascatear múltiplos dispositivos nessa mesma estrutura, formando uma arquitetura multi-chip.

Cada CM consiste de uma máquina de estados e de uma memória RAM. Cada PAC possui um barramento de configuração que conecta o CM com os PAE's do array.

A arquitetura de cada PAE é configurável, podendo ser adicionada qualquer funcionalidade que se deseje, e portanto sua descrição aqui é bem genérica. Entretanto, uma arquitetura típica inclui *back registers* e *forward registers*, utilizados para roteamento vertical, e uma ALU, capaz de executar operações em ponto-fixo, operações lógicas e operações de três entradas, como multiplica-acumula, sort e contadores. Outra configuração comum aos PAE's é como uma memória, capaz de operar em modo FIFO ou como uma RAM para *look-up tables* ou para armazenar resultados intermediários.

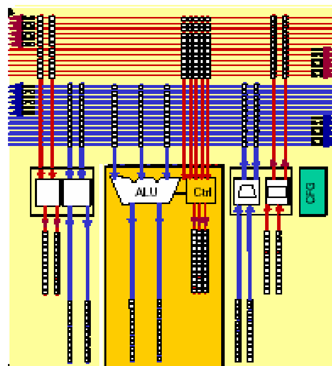


Figura 2.12: Uma das estruturas comuns das PAE's do Xtreme (BECKER, 2003).

Os PAE's são auto-sincronizados, não sendo necessária uma sincronização global: a computação é realizada tão logo os pacotes de entrada necessário estejam disponíveis e os resultados são propagados tão logo eles estejam prontos. Dessa forma, é trivial o mapeamento de um grafo de fluxo de sinais para os PAE's. O sistema de comunicação é projetado para transmitir um pacote por ciclo, e protocolos de hardware garantem que nenhum pacote é perdido, mesmo no caso de *stalls* ou durante o processo de reconfiguração. O conjunto desse mecanismo de tratamento automatizado de pacotes simplifica o desenvolvimento de aplicações, tornando desnecessária a preocupação com um escalonamento de tarefas.

2.3.6 Análise das Arquiteturas Reconfiguráveis

As arquiteturas estudadas, embora todas de granularidade grossa, diferem entre si em muitos pontos, levantando questões bastante interessantes e que podem ser exploradas adequadamente no domínio dos sistemas embarcados.

Para melhor resumir cada uma das arquiteturas num quadro geral de classificação, para utilizar-se apenas os conceitos mais relevantes, tem-se os gráficos de posicionamento, nas figuras 2.13 e 2.14, logo a seguir.

Observe-se que as arquiteturas propostas exploram o domínio dos reconfiguráveis de formas diversas, e que não convergem num modelo específico – o que evidencia a não existência de um *padrão* na área, tendo-se ainda alguns nichos a serem explorados. Ademais, dada a ortogonalidade das classificações, já expostas anteriormente, ainda que no mesmo espectro já pesquisado, a combinação de diversos

desses pontos de formas alternativas pode levar a uma arquitetura completamente diferente das já propostas, e mais ou menos apta a um dado propósito.

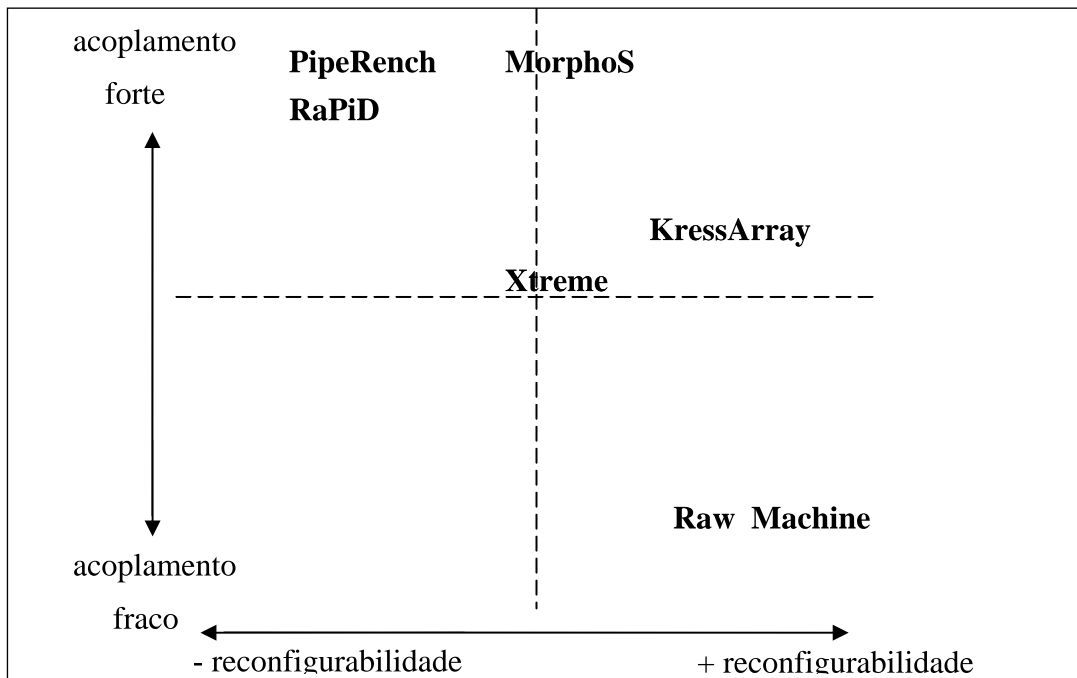


Figura 2.13: Acoplamento X Reconfigurabilidade

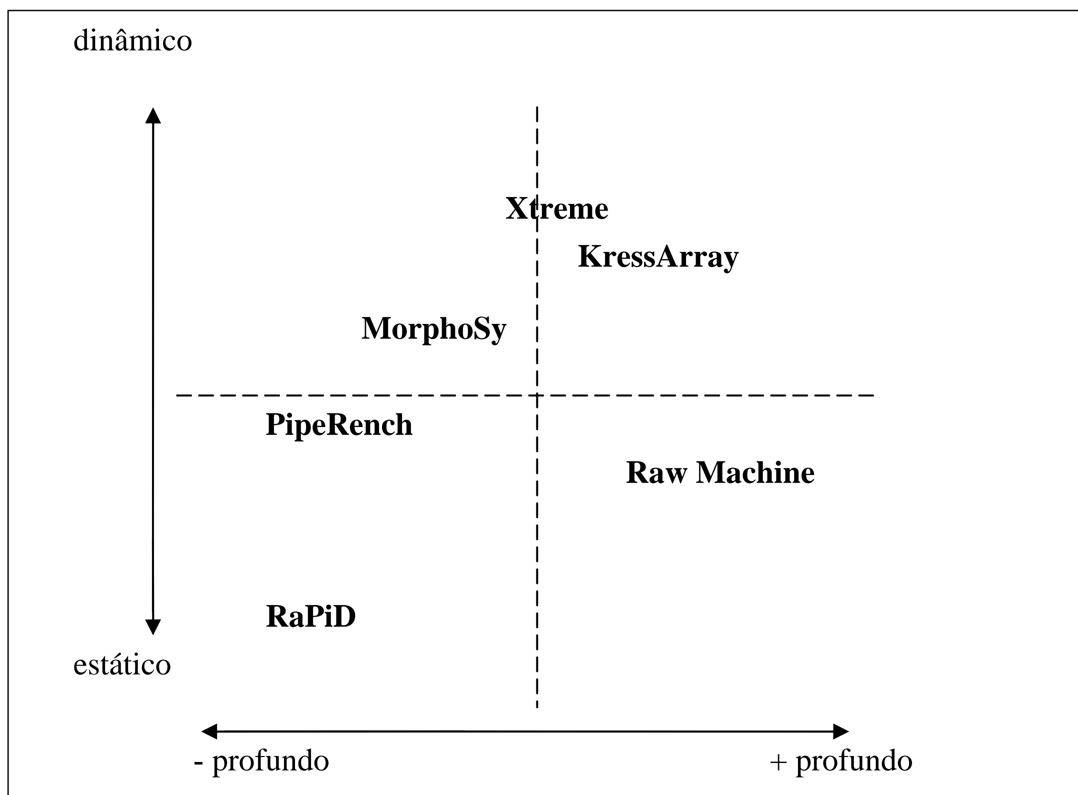


Figura 2.14: Configurabilidade X Profundidade de Configuração

O que se deseja neste trabalho é explorar as arquiteturas reconfiguráveis para o domínio dos Sistemas Embarcados. Sendo assim, alguns pontos relevantes das arquiteturas estudadas podem ser resumidos.

Quanto ao Xtreme, se percebe a possibilidade da configuração do array, e das próprias funcionalidades das células desse array. Essa pode ser uma alternativa interessante aos embarcados, já que para aplicações conhecidas pode-se estaticamente determinar alguns parâmetros arquiteturais apropriados. Entretanto, é de se esperar que o mapeamento da aplicação seja menos genérico e mais complexo.

Outro ponto interessante é a busca da simplificação da escrita de aplicações, quando se tenta eliminar a necessidade de sincronização fazendo as células orientadas ao fluxo de dados e quando se tenta automatizar a comunicação das células. Quanto mais automatizado e simples o processo de escrita de uma aplicação, maiores as chances de uma arquitetura prosperar. Ainda quanto à comunicação, parece ser bastante interessante a utilização de comutação de pacotes ao invés de circuitos, pois o compartilhamento das interconexões aumenta, e pode-se dessa forma tentar reduzir-se o gasto em área das interconexões, que corresponde no geral a uma parcela significativa dessas arquiteturas.

Por fim, um outro ponto interessante é a capacidade de execução de computação condicional e da auto-reconfiguração, que aumenta a usabilidade do array e a sua independência do hospedeiro, tornando-o apto a executar um espectro maior de aplicações e reduzindo o gargalo de comunicação com o hospedeiro e a memória.

Considerando-se a Raw Machine, a idéia de tornar os mecanismos de hardware simples e deixar o seu gerenciamento e alocação de recursos para o software pode ajudar enormemente no gerenciamento da complexidade dos projetos, e também de seu teste e redução do *time-to-market*. Ainda que se aumente a complexidade do software (principalmente do compilador), tem-se agora dois fluxos de desenvolvimento que podem ocorrer em paralelo e razoavelmente independentes, e não mais sequencialmente caso tudo fosse feito em hardware.

A Raw Machine é outra arquitetura que substitui o uso de barramentos de comunicação por switches, que assim como os pacotes do Xtreme indicam um caminho de melhor utilização da rede de interconexões. Entretanto, os switches da Raw são programados estaticamente.

Mas a característica mais marcante da Raw Machine é o fato de ela ser a única dentre as arquiteturas estudadas que se constitui num processador completo e independente de hospedeiro. Seria muito interessante a simplificação de hardware obtida para os *sistemas embarcados* se uma alternativa dessas se mostrasse viável, já que a diferenciação das aplicações se daria apenas por software e se poderia eliminar várias etapas do fluxo de projeto tradicional de SoC's; a massificação da produção dos chips poderia reduzir enormemente o seu custo e ampliar o uso dos *sistemas embarcados* ainda mais.

O KressArray também chama a atenção para alguns pontos importantes. A existência de contextos múltiplos é mais interessante caso se possa configurá-los independente e paralelamente, pois isso pode reduzir o gargalo do tempo de configuração – uma questão importante na tecnologia atual.

Outro ponto ressaltado é a preocupação com a entrada e saída de dados (I/O) em qualquer célula do chip, que não apenas as células das bordas. Isso facilita a implementação das aplicações nas suas etapas de mapeamento, posicionamento e particionamento.

Ainda, assim como o Xtreme, o KressArray foi projetado para ser escalável em nível de chip, o que pode ser interessante se pudermos considerar o aumento da capacidade de processamento assim como fazemos com a capacidade de memória atualmente: apenas acrescenta-se um chip a mais. Essas abordagens sugerem que a escalabilidade dos chips será um tema de ampla pesquisa no futuro, tão logo as pendências atuais sejam resolvidas.

O RaPiD não oferece nenhuma idéia mais inovadora, mas é relevante por ser uma arquitetura precursora das demais. Uma característica interessante, e também explorada pelo KressArray, é a utilização de comunicação *nearest-neighbor*, que pode aumentar a eficiência da computação e economizar em área.

O MorphoSys utiliza DMA para a configuração de contextos, tendo também a maior profundidade de contextos dentre as arquiteturas estudadas. Além disso, apresenta 3 níveis de interconexões, o que aumenta a flexibilidade, mas por outro lado aumenta também a complexidade da criação de software e principalmente compromete uma área importante do chip, que talvez fosse melhor aproveitada com a replicação de mais células, como sugere a Raw Machine.

PipeRench é uma arquitetura notadamente voltada para o processamento de pipelines. É a única dentre as arquiteturas explicitamente pensada para explorar pipelines virtuais (ou temporais). Outra idéia interessante é a utilização de restrições na comunicação das suas *stripes*, que reduzem a área gasta em interconexões e simplificam o mapeamento de aplicações. Além disso, há ainda a idéia dos registradores utilizados para realizar uma comunicação entre contextos diferentes, em tempos diferentes, mas na mesma célula – que evita a propagação de comunicação com outras células e também com a memória (comunicações essas que são recursos valiosos e gargalos importantes para as arquiteturas reconfiguráveis no geral, além de consumir muita energia, que os sistemas embarcados tentam evitar).

Por fim, em muitas dessas arquiteturas há a preocupação com o processamento de pipelines e *streams* de dados – que é uma questão que deve estar sempre presente visto as tendências já expostas em seção anterior. Não é novidade que as aplicações com processamento potencialmente vetorial (SIMD) são as mais beneficiadas com a exploração de uma computação reconfigurável. Aplicações com grande grau de paralelismo e concorrência também são bastante favorecidas. Portanto, não é de se espantar que aplicações de processamento de sinais (como mp3, imdct e outras) tenham obtido tanto êxito. Nesse sentido, muito trabalho já tem sido realizado (MARSHALL, 1999), (RABAEY, 1997), (LEITE, 1994), (BECKER, 2000), (MITSUSYAMA, 2001), mostrando ganhos muito significativos das aplicações reconfiguradas em relação às mesmas aplicações rodando em processadores de propósitos gerais.

Com relação aos sistemas embarcados, essas aplicações poderiam, potencialmente, trazer um ganho de economia de energia – já que se poderia reduzir o tempo de computação. Entretanto, continuariam perdendo parar os ASIC's tanto em área quanto na própria economia de energia.

Para o domínio de sistemas embarcados, a computação reconfigurável passa a fazer sentido na busca dessa economia de energia com a simultânea economia de área – ou seja: havendo no sistema embarcado várias aplicação que, concorrendo pelos recursos reconfiguráveis, pudessem, no contexto geral, aproveitar ambos os benefícios.

Não é difícil encontrar tais aplicações e, ainda que uma idéia bastante proveitosa, essa é uma face restrita do domínio da computação reconfigurável. Reduzir os benefícios da computação reconfigurável a essa única possibilidade seria desprezar grande parte das possibilidades que ela pode oferecer.

Os benéficos da computação reconfigurável poderiam facilmente ser explorados por sistemas embarcados para tolerância a falhas: poder-se-ia aumentar a confiabilidade de um sistema mimetizando-se parte do mesmo num substrato reconfigurável. Em paralelo, decide-se pela sua validade, podendo forçar uma recomputação com uma substituição temporária da parte afetada pela parte reconfigurada.

Outro benefício potencial é a possibilidade de adaptação a diferentes padrões de comunicação e diferentes serviços de camadas de rede, conforme os mesmos vão evoluindo e se consolidando (BECKER, 2003).

Ainda, outra idéia refere-se à possibilidade de se ter diversos *drivers* e subsistemas configurados s*ob demanda*, que é aventada pelo grupo do IMEC, num *PDA* seu em desenvolvimento (MEI, 2002).

E existe também a possibilidade de que num futuro não muito distante, a computação reconfigurável possa vir a competir com os processadores de propósitos gerais. Para tanto resta descobrir-se *como* a computação reconfigurável poderia ser usada de forma eficiente para tratar aplicações de propósitos gerais. Até agora as tentativas têm fracassado.

Entretanto, pelas características da linguagem Java, fundamentada numa máquina virtual baseada em pilha, surgem algumas possibilidades de exploração da mesma, que podem propiciar vantagens também para aplicações de propósitos gerais, como será visto no capítulo 3.

Nessa linha, um arranjo reconfigurável que entenda bytecodes Java, através de *binary translation*, serve como mais uma alternativa de exploração do espaço de projeto para o Sashimi. Poder-se-ia perguntar sobre se o hardware de *binary translation* não faria com que se consumisse todo o ganho de energia pretendido. Ainda que essa seja uma pergunta a ser respondida, deve-se lembrar que os processadores mais recentes (que cada vez mais se preocupam com a economia de energia, dada a ascendência do mercado de pda's e notebook's), a partir do Pentium IV (da Intel), e como o Crusoe (da Transmeta), utilizam *trace cache* e também *binary translation* em sua organização.

Sendo amplas as potencialidades de aplicação das arquiteturas reconfiguráveis aos sistemas embarcados, bem como crescente o domínio das aplicações embarcadas, implicando numa também crescente gama de restrições de projeto, este trabalho concentra-se na busca por economia de energia e aumento de desempenho, mantendo compatibilidade com Java e buscando, tanto quanto o possível, propiciar ganhos também em aplicações de propósitos gerais.

Aliando as possibilidades oferecidas pelas arquiteturas reconfiguráveis com as necessidades dos sistemas embarcados, muitos das inspirações obtidos com o estudo das arquiteturas recém vistas são utilizados na definição da proposta da arquitetura

reconfigurável Javarray e apresentada neste trabalho, como a utilização de *stripes* com interconexões localizadas, como feito no Piperench, por exemplo. Outras inspirações são aproveitadas como sugestões para trabalhos futuros, como a utilização de multicontextos e reconfiguração de pipeline.

2.4 Trabalhos Relacionados

Existem vários projetos de arquiteturas reconfiguráveis destinados a diversos objetivos distintos. Através da implementação de alguns trechos de software diretamente em hardware reconfigurável, grandes aumentos de performance (GUPTA, 1993) bem como economia de energia foram alcançados (SITT, 2002). Processadores como o Chimaera (HAUCK, 1997) e o ConCISe (KASTRUP, 1999) possuem um array reconfigurável fortemente acoplado ao núcleo do processador, limitados a lógica combinacional, o que torna o controle mais simples, reduzindo a sobrecarga necessária de comunicação entre o array reconfigurável e o resto do sistema.

Outros processadores possuem um objetivo mais geral, tentando explorar distintos níveis de paralelismo em aplicação *desktop*, tendo grãos muito maiores como seus elementos de processamento. Piranha (BARROSO, 2000) é um multiprocessador em chip (CMP) de granularidade grossa destinado a explorar paralelismo em nível de threads para servidores comerciais, e TRIPS (SANKARALINGAM, 2003) é uma arquitetura polimórfica em forma de grid com grãos ultra-grandes, destinado a se adaptar a concorrência de pequena e grande granularidades.Tratando com o mercado embarcado, a arquitetura Javarray toma uma outra abordagem e utiliza grãos menores que o TRIPS, propondo o uso das arquiteturas reconfiguráveis para economia de energia através da otimização de desempenho de trechos críticos de software. Idéias semelhantes têm sido exploradas com resultados bastante promissores em trabalhos afins.

Em (ACHUTHARAMAN, 2003) é detalhada uma arquitetura SMTI (*simultaneous multi-trace instruction issue*) onde fica claro a potencialidade da exploração de ILP em arquiteturas baseadas em Java, dadas as características da linguagem, que é baseada em pilha. A abordagem utilizada parte da identificação de *bytecode traces* – seqüências de bytecodes que não possuem dependência de operandos na pilha com quaisquer outros bytecodes em um método. Isso possibilita busca de instruções fora-de-ordem e execução especulativa. Assim, ocorre aumento de desempenho de 13.5% sobre um bloco básico com a exploração do ILP existente, sendo que com o uso de *folding* pode-se chegar a até 54% de aumento de desempenho sobre a execução em ordem tradicional. Embora voltado para a busca de aumento de desempenho, exclusivamente, essa arquitetura SMTI guarda muitas semelhanças com as idéias adotadas para o Javarray, na medida em que tenta alcançar a exploração de ILP em Java através da identificação e reorganização de trechos de bytecodes independentes e da percepção de que a arquitetura da JVM, baseada em pilha, potencializa tais condições. Além da preocupação com energia, a diferença para o Javarray é que esses trechos independentes são reconfigurados numa árvore de dependências, onde algumas instruções são adicionalmente eliminadas, enquanto que no trabalho em questão os trechos identificados rodam em uma arquitetura SMTI – que no fundo segue o modelo de execução tradicional RISC.

O processador XiRisc (GUPTA, 1993) é um processador VLIW baseado numa arquitetura clássica de 5 estágios de pipeline RISC que integra também um datapath reconfigurável em tempo de execução e uma unidade DSP. A unidade reconfigurável do XiRisc - PiCoGA – utiliza células compostas de duas LUTs de granularidade de 2 bits, sendo capazes de executar funções lógicas de 6:1, 5:2 ou 4:4. Essa células são organizadas em colunas onde cada coluna é destinada a implementar um estágio de um pipeline reconfigurável. Dessa forma, e sendo a unidade reconfigurável fortemente integrada ao processador, um conjunto de instruções para aplicações específicas pode ser utilizado. Assim, as porções de algoritmos com computação mais intensiva são mapeadas para a unidade reconfigurável, fazendo com que se obtenha aumentos de performance entre 4.3x até 13.5x, ao mesmo tempo que se reduz o consumo de energia em até 92%.

Já o trabalho com o WCLA (SITT, 2002) propõe o desenvolvimento de uma arquitetura de lógica configurável especificamente destinada ao particionamento dinâmico de software/hardware, num processo conhecido como *warp processing*. Através dessa arquitetura, e projetando-se a FPGA para tratar especificamente o aumento de desempenho de kernels de software, ao invés da aplicação total, os autores atestam ter alcançado aumentos de desempenho entre 2x e 4x, com economia de energia na média de 33%, chegando a 74%. Além disso, o WCLA é voltado para o domínio dos sistemas embarcados, e incorpora, adicionalmente, características de DSPs.

A arquitetura do ADRES (LODI, 2003) é especificamente projetada para o domínio dos sistemas embarcados e utiliza uma abordagem similar ao Javarray, e possui um processador VLIW associado a um array reconfigurável de granularidade grossa, destinado a executar eficientemente apenas os kernels de computação intensiva das aplicações.

O Piperench (GOLDSTEIN, 1999) é uma famosa arquitetura reconfigurável especificamente destinada à execução de processamento *stream* através de técnicas de pipeline, com expressivos resultados de aumento de performance para essa classe de aplicações.

O trabalho desenvolvido no Javarray considera a mesma idéia de otimização de trechos de software mapeados para uma arquitetura reconfigurável, mas difere das abordagens do XiRisc e do WCLA pelo uso de uma granularidade mais grossa nos elementos de processamento e também pela estratégia de otimização, que é escolhida através da análise dos *profiles* das aplicações. Diferentemente da arquitetura ADRES, o Javarray mantém compatibilidade de software com Java, não necessitando compiladores especiais. Ainda, a versão seqüencial da arquitetura Javarray abre a possibilidade de exploração de técnicas semelhantes às utilizadas pelo Piperench, explorando além de ILP outros níveis mais grossos de paralelismo, como processamento baseado em *stream*.

3 PROJETO JAVARRAY

3.1 Proposta

As atuais pesquisas do LSE têm-se desenvolvido grande parte entorno do microcontrolador FemtoJava, já brevemente apresentado na seção 1.3, voltado especificamente para o mercado de sistemas embarcados baseados em Java.

Conforme as restrições impostas a esses sistemas, como necessidade de baixo consumo de energia e potência, e limitações na capacidade de processamento, uma das principais necessidades que se possui é a de executar os códigos Java com eficiência, visando esses objetivos.

Para tanto, pensou-se em aproveitar-se o desenvolvimento das arquiteturas reconfiguráveis aplicando conceitos relevantes ao domínio dos sistemas embarcados e, ao mesmo tempo em que explorando a máquina de pilha do Java, buscar a economia de energia através do aumento de desempenho das aplicações Java, em relação às mesmas aplicações executando no FemtoJava.

Tendo-se então já possuído um microcontrolador de execução nativa Java, optou-se simplesmente por criar uma unidade reconfigurável que permita otimizações apenas em trechos interessantes, ao invés de criar-se um processador reconfigurável completo, deixando-se a cargo do próprio FemtoJava tarefas menos interessantes.

Dessa forma, um requisito do projeto é o de que a arquitetura reconfigurável a ser criada seja transparente ao usuário do microcontrolador. Ou seja: o código da aplicação Java não deve ser alterado, indicando a necessidade do uso de tradução binária.

A unidade reconfigurável foi pensada para ser o mais simples e eficiente possível. Partiu-se da idéia original genérica de utilizar-se um conjunto de registradores a ser interconectado por reconfiguração a um conjunto de ula's. A partir dessa idéia, algumas aplicações embarcadas Java foram analisadas, e acabou-se por determinar uma estrutura de um array com 3x8 *processing elements (pe)*, com conexões localizadas e unidirecionais (ou seja, nos 8 níveis existentes, cada *pe* de um nível apenas pode se comunicar com um *pe* do nível seguinte).

Conforme será melhor explicado nas seções deste capítulo, as características da máquina de pilha Java permitem algumas otimizações na execução de seus *bytecodes* quando reconfigurados. Dessa forma, conseguindo-se otimizar o processamento dos bytecodes Java, espera-se obter uma sensível redução no consumo de energia.

A partir de análises dos *profiles* das aplicações, que foram obtidos através de *profile* das mesmas, identificou-se os trechos com possibilidades de otimização por reconfiguração.

Para tanto, os *profiles* obtidos foram divididos em seus respectivos blocos básicos. Diferentemente da definição clássica de “blocos básicos”, define-se para este trabalho Blocos Básicos como sendo trechos de código limitados por instruções de controle ou escrita em memória; ou seja: são trechos contínuos de código sem saltos internos.

Identificou-se que poucos blocos básicos das aplicações representavam a grande maioria das instruções executadas no *profile*. Esses blocos básicos formam um grafo de dependência de instruções, potencialmente com mais de uma árvore interna: ou seja, um DAG – Dependence Acyclic Graph. Assim, se esses blocos possuíssem um tamanho adequado em quantidade de instruções, eles poderiam ser bons candidatos para otimização através de reconfiguração.

Cada bloco básico pode ser internamente dividido em blocos de operandos. Cada instrução-raíz da árvore de dependências do bloco básico inicia um bloco de operandos, que formam sub-árvores de instruções independentes entre si. Como ficará claro em seções adiante, cada um desses blocos de operandos poderá ser configurado como uma instrução Javarray.

Para o presente trabalho, considera-se que a identificação de trechos de otimização e seu mapeamento reconfigurado para o Javarray ocorrem estaticamente, antes da execução do programa.

Assim, o FemtoJava iniciará a execução de uma dada aplicação, parando e disparando o Javarray quando identificar o endereço de um bloco reconfigurado, e voltando a assumir o controle do programa quando o Javarray terminar, podendo eventualmente chamar o Javarray novamente, quando outro bloco reconfigurado for identificado.

Para realizar as análises das aplicações e selecionar-se os blocos básicos mais relevantes, bem como para identificar a árvore de dependências e os blocos de operandos a serem mapeados para o Javarray, criou-se dois pequenos aplicativos, citados mais adiante.

3.2 Fluxo de Projeto

Para a validação da idéia do trabalho, e também como limitação de escopo deste estudo, foram selecionadas algumas aplicações Java embarcadas, já desenvolvidas pelo LSE, para que seus profiles pudessem ser analisados, através de um *profile* das mesmas. Para a etapa de análise dessas aplicações, foram desenvolvidos dois pequenos programas: um deles, um analisador de blocos básicos, divide o profile da aplicação em blocos básicos e levanta estatísticas sobre eles, como o número de repetições de cada bloco, por exemplo; o outro programa, um gerador de grafos de dependências, toma um bloco básico como entrada e gera um DAG (directed acyclic graph) representando dependências de instruções, onde podem ocorrer internamente árvores de dependência, sendo que cada uma dessas árvores representa um bloco de operandos independente dos

demais. Nesse processo, as instruções são montadas no DAG na medida em que aparecem no bloco básico, sendo que o DAG não é otimizado, já que instruções comuns a diferentes árvores podem não ser reaproveitadas. O gerador de grafos de dependência não chegou a ser desenvolvido em toda sua completude, sendo que alguns *bytecodes* não são suportados. Os programas, bem como seu código fonte, podem ser encontrados no cd de anexos.

Numa segunda etapa, a partir das análises realizadas sobre as aplicações, as noções arquiteturais gerais do Javarray são desenvolvidas, levando em conta também *as inspirações* obtidas com o estudo de outras arquiteturas embarcadas, apresentadas no capítulo anterior.

Sobre essa definição arquitetural uma série de requisitos e problemas acabaram sendo levantados, como questões de acesso à memória e sincronização da computação, por exemplo. Um modelo VHDL totalmente parametrizável de uma primeira versão seqüencial do Javarray foi criado, de forma a se perceber outros eventuais problemas e se realizar o teste de mapeamentos dos blocos básicos – o objetivo foi o de se ter uma primeira versão funcional da arquitetura, que ainda seria evoluída.

O CACO-PS (BECK, 2003) é um simulador de potência criado e utilizado pelo grupo LSE que utilizad descrições em nível RTL para realizar estimativas de consumo de potência em alto nível, informando os resultados em *capacitâncias de gates* de acordo com os bits chaveados.

Após as duas primeiras etapas – de análise dos profiles e de geração da arquitetura base, pôde-se iniciar a exploração do espaço de projeto. Neste momento, fez-se a modelagem de 2 organizações do Javarray para CACO-PS, e outras alternativas e possibilidades foram percebidas e anotadas. Implícito na criação desses diferentes modelos, criou-se uma gama de componentes adicionais à biblioteca CACO-PS. Tanto os modelos CACO-PS quanto o modelo VHDL e os componentes criados estão disponíveis no cd de anexos.

Tendo-se realizado a análise das aplicações, foram selecionados os blocos básicos mais interessantes para mapeamento no Javarray, sobre os quais os blocos de operandos foram identificados e alguns deles selecionados para mapeamento. O mapeamento dos blocos nas diferentes organizações do Javarray foi realizado através dos modelos CACO-PS.

Após a calibragem dos modelos de potência dos componentes CACO-PS, dados de desempenho, energia e potência consumida puderam ser extraídos. Estimativas de área também foram feitas com relação ao modelo VHDL. Nesta fase, além de simulações com os modelos do Javarray, foram realizadas simulações com os modelos já existentes do FemtoJava em suas versões Multiciclo e Low-Power Pipeline.

A Figura 3.1 detalha todas as etapas seguidas nesta dissertação.

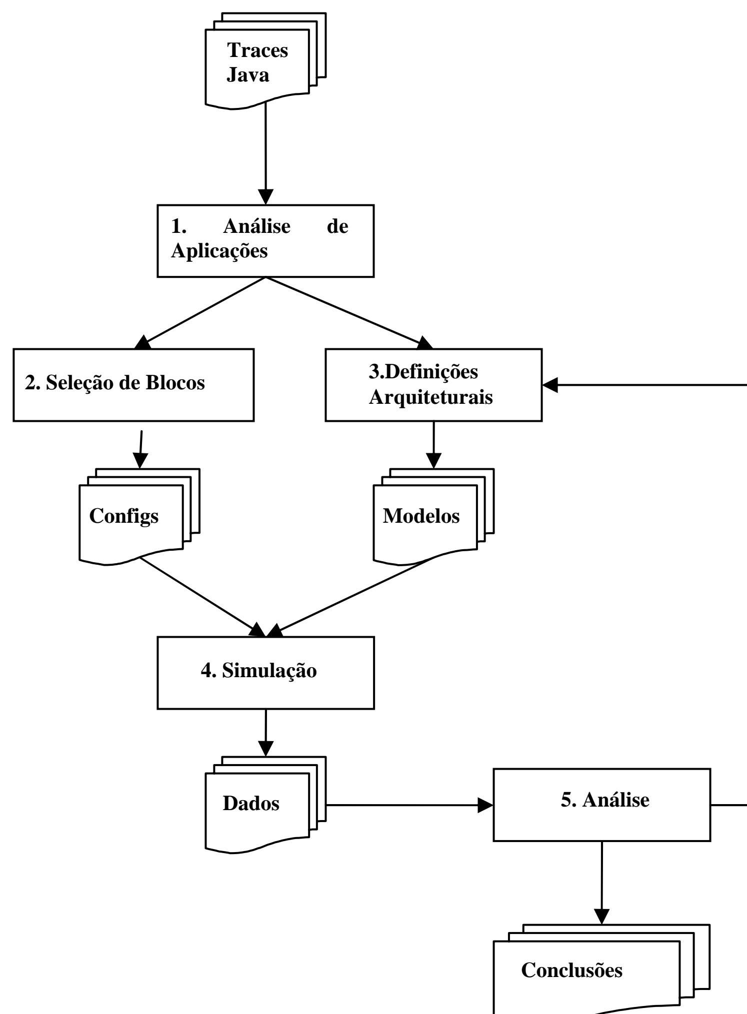


Figura 3.1: Fluxo de desenvolvimento deste trabalho.

O fluxo de desenvolvimento inicia com a Análise de Aplicações (1) embarcadas Java, através de seus *profiles*, de onde blocos básicos interessantes para reconfiguração foram extraídos (2) e necessidades arquiteturais são levantadas (3). A partir das necessidades arquiteturais – classificou-se os bytecodes em instruções de entradas de dados e instruções de execução, para determinar a necessiade de registradores e de elementos de processamento – modelos do Javarray foram implementados e o mapeamento dos blocos selecionados foi realizado para essa arquitetura, permitindo a sua Simulação em CACO-PS (4) e a obtenção de resultados para Análise (5). Após a etapa de Análise, eventualmente foram realizadas melhorias arquiteturais, realimentando o fluxo de projeto. Por fim, os dados de análise geraram as conclusões deste trabalho.

Na etapa de Análise das Aplicações, foram utilizadas ferramentas especificamente desenvolvidas para tal, neste trabalho - o Basic Block Analyser e o Dataflow Graph Generator, encontradas no cd de anexos. Na etapa de Simulação, foram utilizados o simulador CACO-PS, do LSE, e o ambiente Quartus, da Altera. Fez-se

também uso de modelos já existentes do FemtoJava, para posterior comparação de resultados.

3.3 Análises de Aplicações Embarcadas

Conforme as etapas do fluxo de projeto detalhadas na seção anterior e decorrente dos objetivos propostos de redução de consumo de energia através do aumento de performance, a abordagem adotada foi selecionar algumas aplicações embarcadas Java, já previamente escritas para as diferentes versões do FemtoJava, buscando inspiração na análise de seus blocos básicos, com a expectativa de que idéias proveitosas pudessem emergir e de que se identificasse possibilidades de exploração das arquiteturas reconfiguráveis na execução de tais aplicações. A partir disso, blocos básicos *interessantes* para reconfiguração foram selecionados e então a topologia geral da arquitetura reconfigurável Javarray pôde ser proposta.

As aplicações selecionadas para análise constituem-se em aplicações Java utilizadas em outros projetos do LSE, previamente descritas para o FemtoJava, e que representam 3 classes distintas de computação: uma delas, o IMDCT, é um *kernel* do algorítmo de descompressão do MP3, representando aplicações de computação intensiva; outra, o SortBubble é um algorítmo de ordenamento amplamente difundido; e por fim, a terceira aplicação, o Crane 16 bits, representa uma aplicação de controle de um guindaste, de forma a movimentar-se as cargas mantendo-as equilibradas, sem oscilações.

A ênfase em selecionar-se classes distintas de computação está de acordo com o objetivo de procurar por aumento de performance e economia de energia em aplicaçòes de propósitos gerais, e não apenas em aplicações SIMD.

Todas as inspirações obtidas e considerações realizadas acerca das análises de aplicações são expostas a seguir.

3.3.1 Análise de Representatividade dos Blocos Básicos nos Profiles

As aplicações consideradas em análise são, a saber, o IMDCT, o SortBubble e o Crane 16bits (os arquivos de dados e instruções podem ser encontrados no cd de anexo). Tais aplicações já haviam sido previamente desenvolvidas pelo grupo LSE e foram reaproveitadas neste trabalho. Inicialmente, rodou-se tais aplicativos no simulador CACO-PS para a obtenção dos *profiles* das aplicações. Logo após, esses *profiles* foram analisados pelo Basic Block Analyser, desenvolvido especificamente para o presente trabalho. As estatísticas coletadas através desse processo estão resumidas adiante.

3.3.1.1 IMDCT

A tabela 3.1 seguinte resume os dados obtidos com o *profiles* da aplicação IMDCT, e explicita na primeira coluna um número de identificação para o bloco básico,

tendo logo em seguida o número de *repetições* desse bloco básico no *profiles* e o percentual de repetições sobre todas as repetições de blocos básicos ocorridas no *profiles*; logo em seguida, há o número de instruções existentes em cada bloco e o percentual das instruções desse bloco sobre o total de instruções quando somadas todas as instruções dos blocos básicos; há ainda o número de instruções executadas por cada um dos blocos – uma simples multiplicação das *repetições* pelo *número de instruções* – e, da mesma forma, o percentual sobre o total; por fim, o percentual de execução de cada bloco básico sobre o total de instruções executadas é dividido pelo número de instruções de cada bloco para gerar a coluna *ganho*, e representa o custo-benefício médio de cada instrução do bloco básico em questão.

Tabela 3.1: Estatísticas do *profile* da aplicação IMDCT.

Bb	Repetições	%	instruções	%	inst exec	%	ganho/inst
0	1	0,11%	2	1,32%	2	0,01%	0,00%
1	1	0,11%	1	0,66%	1	0,00%	0,00%
2	1	0,11%	7	4,64%	7	0,03%	0,00%
3	1	0,11%	3	1,99%	3	0,01%	0,00%
4	1	0,11%	6	3,97%	6	0,02%	0,00%
5	32	3,51%	3	1,99%	96	0,38%	0,13%
6	51	5,60%	3	1,99%	153	0,60%	0,20%
7	36	3,95%	17	11,26%	612	2,40%	0,14%
8	1	0,11%	13	8,61%	13	0,05%	0,00%
9	1	0,11%	5	3,31%	5	0,02%	0,00%
10	18	1,98%	13	8,61%	234	0,92%	0,07%
11	18	1,98%	3	1,99%	54	0,21%	0,07%
12	558	61,25%	37	24,50%	20.646	81,07%	2,19%
13	31	3,40%	7	4,64%	217	0,85%	0,12%
14	144	15,81%	23	15,23%	3.312	13,01%	0,57%
15	15	1,65%	7	4,64%	105	0,41%	0,06%
16	1	0,11%	1	0,66%	1	0,00%	0,00%
	execuções de bb's:	911	instruções totais:	151	inst exec totais:	25.467	

Na inspeção da tabela 3.1 anterior, pode-se perceber que ocorreram no profile do aplicativo IMDCT 17 blocos básicos. Em sua maioria, são blocos que ocorrem com uma freqüência de repetição muito baixa e com um pequeno número de instruções também.

Entretanto, dois desses blocos chamam especial atenção: o 12 e o 14. Esses dois blocos básicos somados correspondem a aproximadamente 40% das instruções totais, mas 94% de todas as instruções executadas, como pode ser visto na figura 3.2.

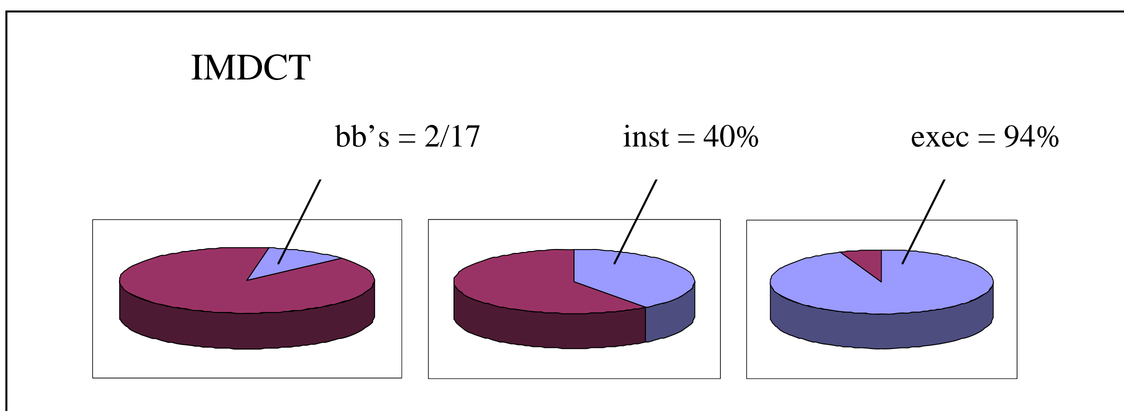


Figura 3.2: Representatividade dos blocos básicos selecionados no IMDCT.

Além de corresponderem a uma parcela enorme do *profile* total, esses dois blocos básicos apresentam características interessantes para sua aplicabilidade em um substrato reconfigurável: elevado número de repetições e tamanho razoável em número de instruções.

Da mesma forma que para o IMDCT, foram coletadas as mesmas estatísticas para o SortBubble e para o Crane 16bits (as tabelas se encontram no cd de anexo). Os dados, já resumidos, dessas aplicações são expostos nas figuras 3.3 e 3.4.

3.3.1.2 SortBubble

Quanto ao SortBubble, percebeu-se a existência de 12 blocos básicos na composição total da aplicação. Da mesma forma que no IMDCT, dois blocos básicos chamam especial atenção. Estes, por sua vez, correspondem a 35% das instruções e 80% da execução desse *profile*, como resumido graficamente pela figura 3.3. As mesmas características de repetição e tamanho, necessárias para a exploração no substrato reconfigurável, foram encontradas.

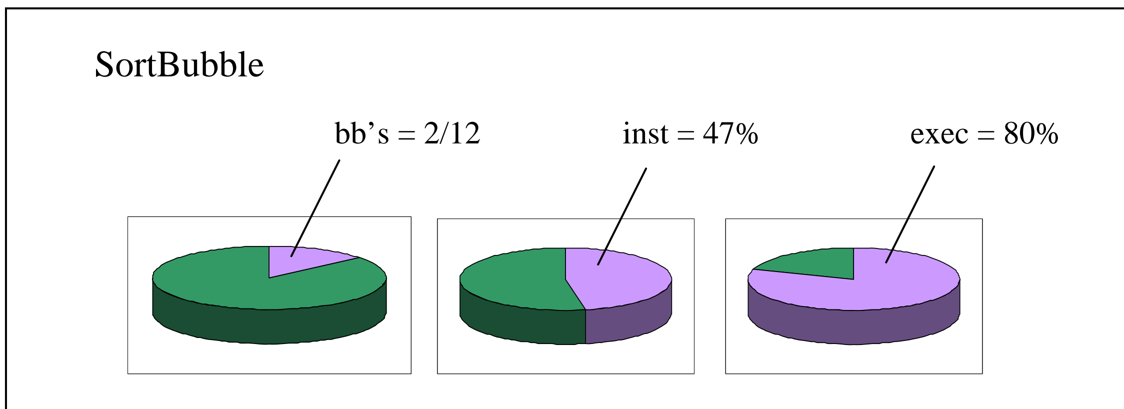


Figura 3.3: Representatividade dos blocos básicos selecionados no SortBubble.

Dois blocos básicos estariam em segundo lugar em um nível de interesse. Entretanto, ambos os blocos apresentam um tamanho pequeno em número de instruções – o que, a princípio, os faz menos interessantes que os primeiros.

Mas prestando-se atenção à coluna de *ganho*, se percebe que as instruções do bloco 8 são mais "econômicas" que as instruções do bloco 7 – que foi um dos selecionados (a tabela do SortBubble se encontra no cd em anexo). Deve-se estar atento para essa situação. Havendo restrições que determinem a opção do projetista do sistema embarcado pela configuração por um ou outro bloco básico, a informação de *ganho* pode ser representativa para sua decisão. Blocos básicos com *ganho* maior pode trazer melhor custo-benefício no uso da memória necessária para armazenar instruções reconfiguradas, por exemplo.

3.3.1.3 Crane 16bits

Diferentemente das duas aplicações anteriores – que são *kernels* bastante específicos usados em aplicações maiores – o Crane16bits é uma aplicação real e completa. Ainda diferentemente das duas anteriores, fortemente voltadas ao *processamento*, o Crane é uma aplicação voltada para o *controle*. Devido a isso, seu *profile* é muito maior que os anteriores, ocorrem muitos desvios e, por conseqüência, um número maior de blocos básicos e de menor tamanho.

Pode-se facilmente perceber que os padrões verificados nas duas primeiras aplicações deixam de ocorrer. Este *profile* possui 118 *blocos básicos* – um número bem mais elevado. Ainda, como consideração deste caso, os *blocos básicos* deste *profile* são pequenos principalmente pelo tipo de aplicação, que é orientada ao *controle*.

Mas ainda que não haja como nos casos anteriores apenas dois *blocos básicos* que representem o *profile*, se formos agrupar os blocos básicos responsáveis pela maior parcela da execução, teremos que todos os blocos básicos que correspondem a 2% ou mais das execuções das instruções no *profile* – 14 blocos básicos – quando somados, representam 60% da execução total do *profile*. Em relação ao número total de blocos básicos existentes, esses representam 12% , e suas instruções somam 23% do código total da aplicação.

Entretanto, nem todos esses blocos básicos apresentam as características de repetição e tamanho adequados – muitos desses selecionados são constituídos de poucas instruções ou com um número muito reduzido de repetições, o que faz com que o seu processo de análise e configuração possa eventualmente não ser interessante.

Sendo assim, selecione-se apenas os *blocos básicos* com mais de 5 instruções – mais adiante faz-se algumas considerações sobre a adequação dos blocos básicos para mapeamento – e temos que 7 blocos básicos representam 34% das instruções executadas; e são 6% dos blocos básicos existentes, e que representam 20% do código total do profile. Esses dados estão representados graficamente na figura 3.4.

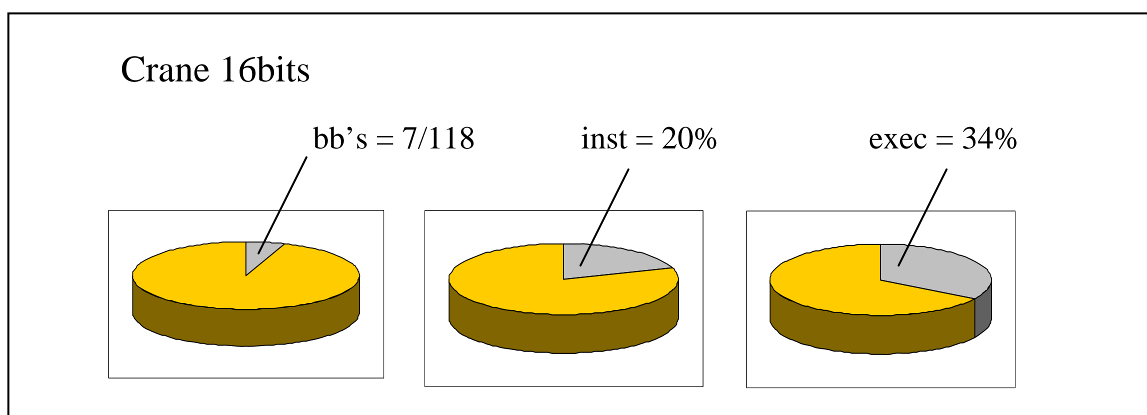


Figura 3.4: Representatividade dos blocos básicos selecionados no Crane 16bits.

Ainda vale a observação de que não se sabe se 6 é mesmo um número mínimo de instruções adequado para a exploração num reconfigurável; apenas se está chamando a atenção para uma propriedade que parece comum à todos os aplicativos: a de que pouco código é responsável por grande parcela da execução dos *profiles*, e que se dentro desses poucos blocos puder-se encontrar blocos adequados para reconfigurar, aí existe um ganho potencial a ser explorado.

Além disso, a resposta à validade ou não de um bloco de poucas instruções não deve ser respondida de uma forma tão simplista. Precisa-se levar em conta quais e de que forma ocorrem as interdependências entre as instruções desse bloco – essa questão é melhor compreendida mais adiante, com a obtenção dos grafos de dependência dos blocos básicos selecionados nos *profiles* anteriores. Entretanto, quanto mais instruções existentes num bloco básico, maior a probabilidade de exploração de paralelismo.

A análise dos tipos e orientação de mais aplicações, e a elaboração e análise de mais *profiles* de aplicativos reais é uma tarefa importante, e que pode trazer alguma inspiração maior. Entretanto, pelo tempo escasso isso não foi feito.

Mesmo que um espectro tão amplo de aplicações quanto o desejado não tenha sido explorado, acredita-se que os *profiles* analisados nesta seção sejam suficientes para demonstrar os princípios que levaram à proposta da arquitetura reconfigurável Javarray, e de como pretende-se aumentar o desempenho de aplicações embarcadas baseadas em Java para a redução do consumo de energia. Ainda, espera-se demonstrar que uma abordagem reconfigurável pode ser útil na exploração de aplicações não-vetoriais ou de processamento de sinais, domínios onde as arquiteturas reconfiguráveis já se

comprovaram eficientes, mas também é possível sua aplicação para aplicações de propósitos gerais e até mesmo para aplicações voltadas ao controle.

Não há um critério predefinido para saber quais blocos básicos são interessantes para reconfiguração ou não. Adiantando um pouco idéias que serão expostas nos capítulos a seguir, observa-se que a adequação do bloco para mapeamento para o Javarray depende dentre outras coisas do processo de mapeamento ser estático ou dinâmico, e também da quantidade de memória disponível e desejável para utilização com configurações de blocos básicos. Assim, se o processo de mapeamento for dinâmico – em tempo de execução da aplicação – dispende-se um certo gasto de energia e tempo de processamento para se produzir o mapeamento de um bloco básico para o Javarray; se esse bloco se repete poucas vezes na execução da aplicação, o processo de mapeamento do bloco pode consumir mais energia do que se pode eventualmente economizar com a sua execução reconfigurada para o Javarray. Ainda, vale lembrar que quanto mais blocos básicos forem mapeados para o Javarray, mais memória será necessária para armazená-los, o que também acarreta um consumo de energia adicional. Portanto, a definição de quais blocos básicos valem a pena serem reconfigurados para o Javarray depende muito de uma análise dos projetista do sistema embarcado e das aplicações em questão. Com a utilização dos analisadores de código criados, essa analise é rápida, entretanto.

3.3.2 Análise dos Grafos de Dependência

Tomando-se por base os blocos básicos selecionados nos *profiles* das aplicações IMDCT, SortBubble e Crane 16bits, e utilizando um outro pequeno programa, o Dataflow Graph Generator, desenvolvido durante este trabalho (o aplicativo, bem como seus fontes, pode ser encontrado no cd de anexos), gerou-se o grafo de dependências das instruções dos mesmos.

O grafo do Crane16bits não foi gerado porque a implementação atual do DataflowGraph ainda não suporta completamente todos os *bytecodes* Java – entre eles alguns necessários nos blocos básicos do Crane16bits. Assim, alguns grafos foram gerados manualmente, através da análise dos bytecodes dos blocos básicos.

3.3.1.4 Necessidade de Recursos dos Blocos Básicos

Os grafos que se seguem apresentam as interdependências ocorridas nas instruções dos blocos básicos adequados, sendo que a instrução da raiz tem como sub-árvore as instruções da qual depende para poder executar.

Um DAG, que representa um bloco básico, pode conter mais de uma árvore. Cada uma das árvores derivadas desses DAG representa um bloco de operandos – que são independentes entre si. O DAG inicial não se encontra em sua forma otimizada: as instruções são montadas em ordem de dependência na medida em que são verificadas no código do bloco básico, em diferentes blocos de operandos. O método utilizado para a identificar os blocos de operandos é o mesmo utilizado em (BECK, 2005).

Numa tentativa despreocupada de classificação das instruções, identifica-se com o desenho de um cilindro as operações que acessam a memória; com o desenho de uma

ula, as instruções que executam operações; com o desenho de uma pilha, as instruções que acessam a pilha ou variáveis locais; com uma interrogação, as instruções de comparação; e por fim, as instruções *getstatic* e *putstatic* são referenciadas por uma exclamação.

Perceba-se nos grafos seguintes a possível existência de mais de um bloco de operandos. Isso ocorre quando existem instruções como *iastore*, que gravam conteúdo na memória, não se podendo garantir qual o endereço sem que se compute de fato sua sub-árvore. Sendo assim, também não há forma de se identificar onde tal endereço está sendo utilizado, fazendo com que a árvore cuja instrução *iastore* é raiz seja, no grafo, independente das demais árvores.

Outro exemplo são as instruções condicionais de controle de fluxo, como os if's. Não se pode garantir o resultado do *if*, fazendo com que essa árvore também seja independente das demais.

Percebe-se, dessa forma, que não basta apenas identificar *blocos básicos*, pois existem, dentro desses, outros blocos mais fundamentais – os blocos de operandos, que serão de fato os blocos a serem reconfigurados.

Entretanto, a identificação dos blocos básicos mais representativos é importante, pois determina os pontos de maior possibilidade de otimização. Após um bloco básico selecionado, deve-se verificar se existe algum, ou mais, bloco de operandos com um número razoável de instruções que valha a pena reconfigurar.

A seguir, analisa-se os blocos básicos selecionados para as aplicações já discutidas anteriormente.

Como observação, o trechos de código existente a seguir representa o bloco básico 12 da aplicação IMDCT, e significa: o primeiro número indica a linha de execução no *profile*; o segundo campo indica o código do bytecode executado; a seguir há uma descrição textual do mesmo bytecode e então o endereço correspondente dessa instrução na rom. Esse formato é gerado pelo CACO-PS, habilitando-se a opção de impressão do *profile*.

O código do bloco básico 12 do IMDCT traduz-se conforme abaixo, onde o primeiro número indica o número da instrução executada no *profile*, seguido pelo código do *opcode*, cujo mnemônico por sua vez está comentado e também indicando o endereço da ROM onde esse *bytecode* se localiza:

```
0388  :  b2; -- getstatic = e5
0389  :  b2; -- getstatic = e8
038a  :  10; -- bipush = ea
038b  :  b2; -- getstatic = ed
038c  :  68; -- imul = ee
038d  :  60; -- iadd = ef
038e  :  b2; -- getstatic = f2
038f  :  b2; -- getstatic = f5
0390  :  10; -- bipush = f7
0391  :  b2; -- getstatic = fa
0392  :  04; -- iconst_1 = fb
0393  :  64; -- isub = fc
```

```
0394  :  68; -- imul = fd
0395  :  60; -- iadd = fe
0396  :  10; -- bipush = 100
0397  :  60; -- iadd = 101
0398  :  2e; -- iaload = 102
0399  :  b2; -- getstatic = 105
039a  :  b2; -- getstatic = 108
039b  :  10; -- bipush = 10a
039c  :  b2; -- getstatic = 10d
039d  :  04; -- iconst_1 = 10e
039e  :  64; -- isub = 10f
039f  :  68; -- imul = 110
03a0  :  60; -- iadd = 111
03a1  :  10; -- bipush = 113
03a2  :  60; -- iadd = 114
03a3  :  2e; -- iaload = 115
03a4  :  60; -- iadd = 116
03a5  :  4f; -- iastore = 117
03a6  :  b2; -- getstatic = 11a
03a7  :  04; -- iconst_1 = 11b
03a8  :  60; -- iadd = 11c
03a9  :  b3; -- putstatic = 11f
03aa  :  b2; -- getstatic = 122
03ab  :  10; -- bipush = 124
03ac  :  a4; -- if_icmple = e2
```

O grafo a seguir refere-se ao bloco básico 12 do *profile* do IMDCT, e foi obtido com o uso da aplicação DataflowGraph, já citada – figura 3.5.

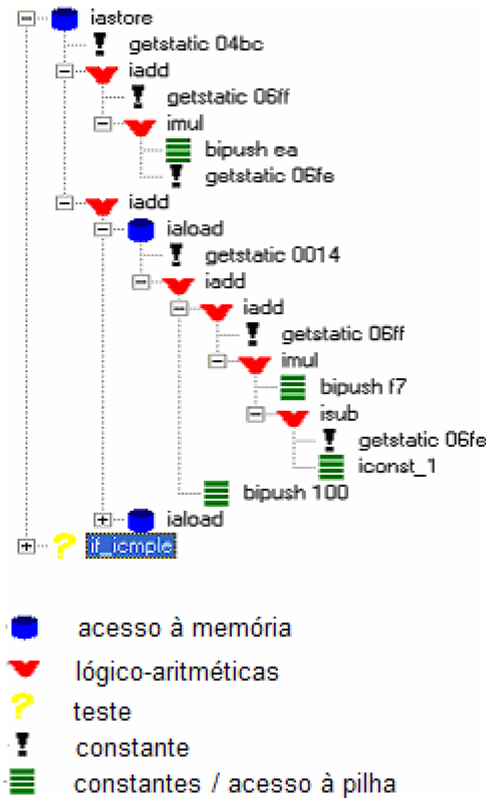


Figura 3.5: Grafo de dependências do bloco básico 12 do IMDCT.

No grafo acima, a instrução *if_icmple* não está expandida, mas contém uma sub-árvore com algumas poucas instruções, a princípio não tão interessantes para mapeamento num substrato reconfigurável. Da mesma forma, existe um *iaload* que não está expandido; mas é porque sua sub-árvore é idêntica à do outro *iaload* que aparece expandido em seu mesmo nível.

Os ícones utilizados ao lado do *lable* de cada bytecode indica a natureza do mesmo. Os ícones de “ulas” representam bytecodes que necessitam execução em um *pe*. Os pontos de exclamação representam *getstatics* e *putstatics*. Os ícones em forma ciclíndrica representam acessos à memória. O ponto de interrogação representa instruções de controle de fluxo. Os ícones em forma de pilha representam constantes ou variáveis em pilha.

Tomando-se a árvore de *iastore* como estudo, percebe-se uma *profundidade* de valor 7, conforme a figura 3.6.

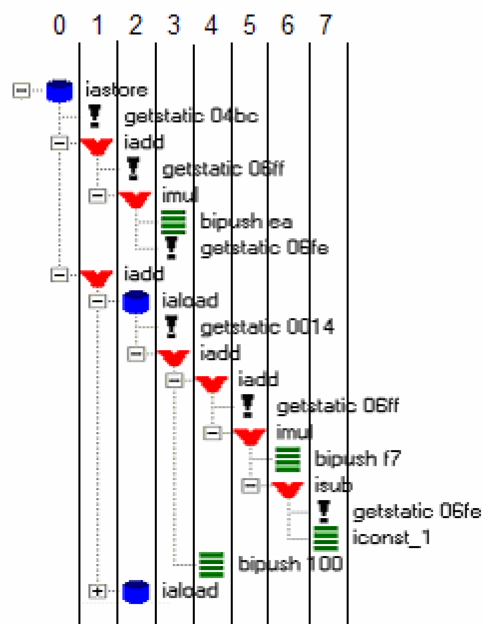


Figura 3.6: Níveis de profundidade da árvore de *iastore*.

A distribuição das instruções se dá conforme o gráfico (figura 3.7):

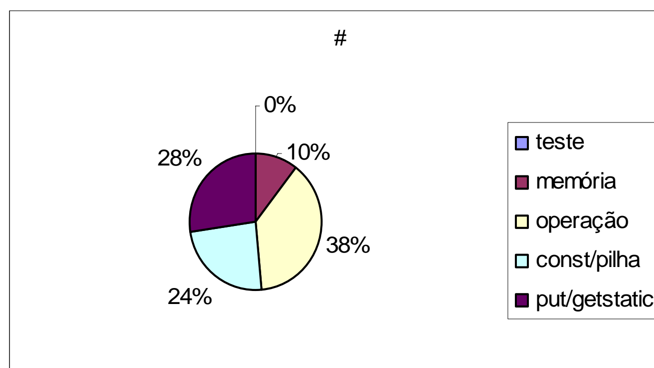


Figura 3.7: Distribuição de instruções do bloco básico 12 do IMDCT.

Perceba-se a predominância de instruções do tipo *operação*, seguidas por *putstatic* e *getstatic* e em seguida instruções de *acesso à pilha*.

Se considerarmos instruções *getstatic*, *putstatic* e de acesso à pilha numa mesma categoria (instruções de entrada de dados, que necessitam ser carregadas em algum registrador) e as demais instruções numa outra (instruções de processamento, que necessitam de algum *processor element* – de agora em diante referenciado apenas por *pe* – para executar), tem-se as seguintes necessidades, sem reuso (tabela 3.2):

Tabela 3.2: Necessidade de recursos do bloco básico 12 do IMDCT.

Tipo	#
pe's	14
Registradores	16

Perceba-se uma equivalência no número de recursos utilizados por essas duas categorias.

Prosseguindo com a análise, utiliza-se essas mesmas categorias para determinar o número de recursos utilizados a cada nível de profundidade da árvore, onde a instrução-raíz corresponde ao primeiro nível. A tabela encontrada é interessante, onde pode-se notar que o número máximo de células utilizadas é de 3 e o número máximo de registradores é 4. Além disso, se tomarmos os níveis adjacentes de dois em dois, o número máximo de registradores utilizados é 6 – tabela 3.3.

Tabela 3.3: Análise de recursos por níveis do bloco básico 12 do IMDCT

Nível	*pe's*	*Registradores*
1	1	1
2	2	1
3	3	4
4	2	2
5	2	2
6	2	2
7	2	4

A análise realizada para o bloco 12 do IMDCT considerada também para a obtenção dos dados de todos os demais blocos básicos selecionados, de todas as aplicações, e que estão resumidos na tabela 3.4 adiante. Os gráficos e árvores de dependência encontram-se no cd de anexos.

Tabela 3.4: Distribuição de recursos nos grafos de dependência do bloco básico 12 do IMDCT.

Distribuição de Recursos nos Grafos de Dependência										
Blocos de operandos *	**nível:**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**TOTAL**
bb 12 IMDCT	**pe's:**	**1**	**2**	**3**	**2**	**2**	**2**	**2**		**14**
	reg's:	**1**	**1**	**4**	**2**	**2**	**2**	**4**		**16**
bb 14 IMDCT	**pe's:**	**1**	**2**	**2**	**1**	**1**				**7**
	reg's:	**1**	**2**	**3**	**1**	**2**				**9**
bb 6 SortBubble	**pe's:**	**1**	**2**	**1**						**4**
	reg's:	**0**	**3**	**2**						**5**

bb 7 SortBubble	**pe's:**	**1**	**1**	**1**						**3**
	reg's:	**2**	**1**	**2**						**5**
bb 8 Crane 16bits	**pe's:**	**1**	**1**	**0**						**2**
	reg's:	**0**	**1**	**2**						**3**
bb 31 Crane 16bits	**pe's:**	**1**	**1**	**0**						**2**
	reg's:	**0**	**1**	**2**						**3**
bb 35 Crane 16bits	**pe's:**	**1**	**1**	**0**						**2**
	reg's:	**0**	**1**	**2**						**3**
bb 39 Crane 16bits	**pe's:**	**1**	**1**	**2**	**0**					**4**
	reg's:	**0**	**1**	**0**	**4**					**5**
bb 44 Crane 16bits	**pe's:**	**1**	**1**	**2**	**2**					**6**
	reg's:	**0**	**1**	**0**	**2**	**4**				**7**
bb 51 Crane 16bits	**pe's:**	**1**	**1**	**2**	**2**	**0**				**6**
	reg's:		**1**	**0**	**2**	**4**				**7**
bb 62 Crane 16bits	**pe's:**	**1**	**2**	**2**	**2**	**4**	**2**	**2**	**0**	**15**
	reg's:	**0**	**0**	**2**	**0**	**0**	**4**	**2**	**4**	**12**

*** nesta tabela, quando ocorre mais de um bloco de operandos por grafo, está-se considerando apenas o maior.**

Pode-se perceber, pela tabela, que o número máximo de níveis utilizados foi 8, sendo que grande parte dos blocos básicos utiliza apenas 3 níveis de profundidade. Além disso, perceba-se a relativa equivalência entre recursos de registradores e *pe's* necessários, mostrando um equilíbrio entre instruções de entrada de dados e instruções de execução.

Analisando-se cada nível individualmente, o número máximo de *pe's* necessários implica em uma largura de 3 *pe's*.

Ainda, buscando-se reaproveitamento de recursos através de compartilhamento, e também buscando uma alta localidade dos mesmos, uma análise de recursos de 2 em 2 níveis mostra que nunca foi necessário mais do que 6 registradores, e também que nunca foi necessário mais de 2 acessos à memória num mesmo nível. Os dados referentes ao acesso à memória não são mostrados aqui, mas podem ser observados através dos gráficos relativos aos grafos de dependência de instruções, no cd em anexo, onde se percebe que o número de acessos à memória é de fato bastante reduzido.

Algumas observações pertinentes podem ser efetuadas a respeito dos dados compilados acima. Primeiramente, considerou-se na tabela 3.4 apenas o maior bloco de operandos do bloco básico em questão, para simplificação da análise.

Também, como algumas instruções-raiz são mais complexas que operações de ula, como o *ireturn* e o *invokestatic*, tais instruções foram desconsideradas nas estatísticas apresentadas. Considera-se que essas instruções não serão reconfiguradas,

devendo ser executadas pelo processador hospedeiro. Como exemplo, o grafo do bloco básico 8 do Crane 16bits é apresentado na figura 3.8. A instrução de *ireturn* foi desconsiderada, sendo portanto a instrução *iand* a raiz dessa árvore contada nas estatísticas.

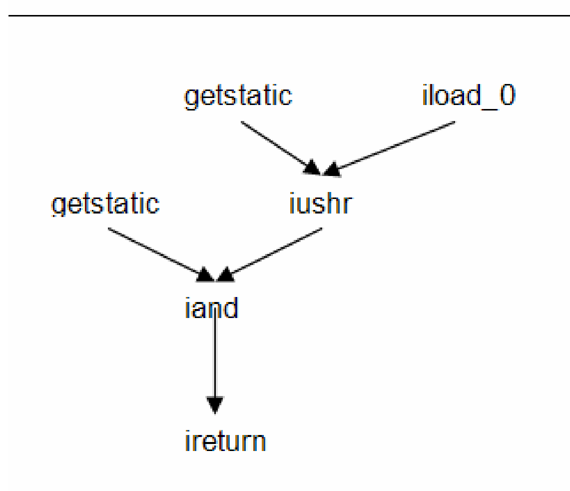


Figura 3.8: Árvore de dependências do bloco básico 8 do Crane 16bits.

Além disso, é conhecido o fato de que uma máquina de pilha lineariza um fluxo de instruções potencialmente paralelo, e que com a reconfiguração do código, podemos explorar toda a capacidade de ILP existente na aplicação. Uma vez que o código na máquina de pilha é linear, faz-se uso comum de instruções de armazenamento em variáveis para utilização de um resultado mais adiante no código. Quando esse fluxo é traduzido para um grafo de dependências, todos os pares de instrução do tipo *istore-iload* e *putstatci-getstatic* presentes internamente no grafo podem ser eliminados, reduzindo ainda mais o número de instruções necessárias para a execução do bloco básico reconfigurado. Essas instruções suprimíveis foram também desconsideradas na contagem da tabela 3.4. As figuras 3.9 e 3.10 demonstram grafos com instruções que foram suprimidas. De fato, o *istore_3* deve ser executado, por possibilidade de dependência com outros blocos básicos, mas pode ser executado em paralelo ou no final, sendo eliminado do fluxo de processamento seqüencial.

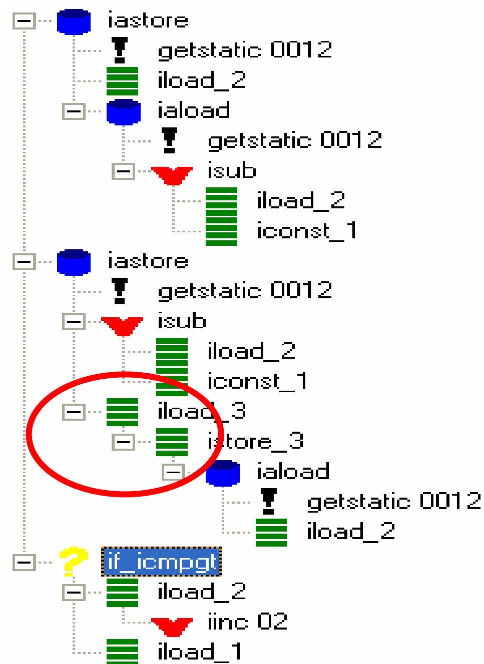


Figura 3.9: Instruções *iload-istore* que podem ser suprimidas.

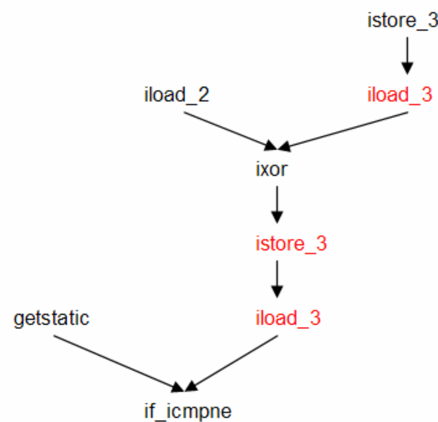


Figura 3.10: Outras instruções iload-istore que podem ser suprimidas.

Há casos, como no bloco 62 do Crane 16bits, onde ocorre muito reaproveitamento de resultados através de uso intenso de pares de instruções *istore-iload* (uso intenso de variáveis), conforme mostrado na figura 3.11. Nesta análise, está-se considerando apenas a parte do grafo que deriva da instrução-raíz que comporta a maior parte das instruções do grafo. No momento de mapeamento do mesmo, como será visto adiante no capítulo de resultados, esse grafo será particionado temporalmente para que caiba na arquitetura proposta.

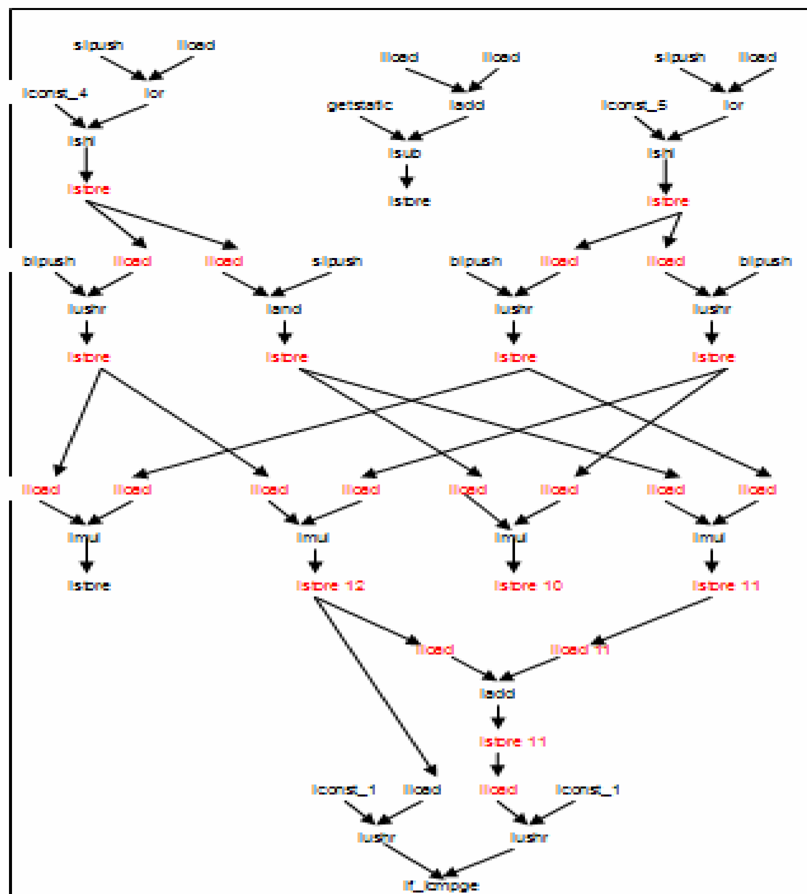


Figura 3.11: bloco básico 62 do Crane 16bits.

3.3.1.5 Repetibilidade de Instruções nos Blocos Básicos

A análise de recursos recém exposta considera um limite superior na necessidade de recursos para a execução dos blocos de operandos considerados. Deve-se perceber, porém, que existem instruções de entrada de dados (representadas por instruções-folha nas árvores de dependência), que ocorrem repetidas na árvore. Assim, o número de registradores necessários pode ser de fato inferior aos números expostos. Uma análise quanto a repetibilidade de instruções no fluxo normal dos bytecodes Java foi realizada.

Por se tratar de uma máquina de pilha, muitas instruções de entrada de dados ocorrem repetidas no fluxo normal do código Java. A reconfiguração dessas instruções para um grafo de dependências pode eliminar essa repetição, fazendo com que o resultado seja calculado apenas uma vez e usado onde necessário. Além disso, muitas das instruções de entrada de dados referem-se à constantes, significando que, após ter-se as instruções reconfiguradas, apenas uma parcela das instruções de entrada de dados deve ser de fato executada a cada repetição do bloco básico.

Considerando apenas as instruções de entrada de dados, já que muitas das instruções representam constantes e algumas instruções variáveis ocorrem replicadas no código Java original, há uma redução de até 80% nas instruções que necessitam ser

executadas a cada ocorrência dos blocos básicos (figura 3.12). Isso por si só já pode economizar energia e alguns ciclos de processamento.

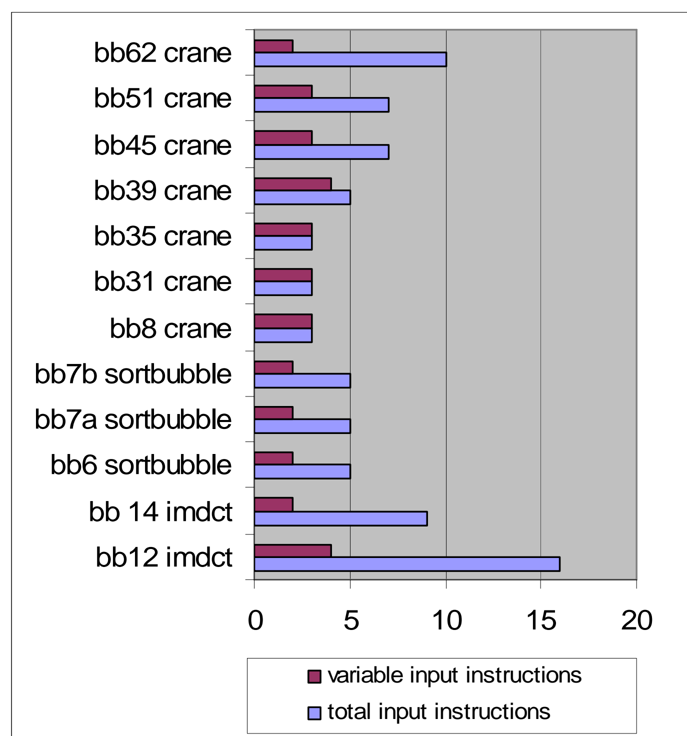


Figura 3.12: Ocorrem reduções de até 80% no número de instruções de entrada de dados necessárias com o Javarray.

Por fim, somando-se a redução no número de instruções de entrada de dados à redução obtida por supressão de instruções internas, obtém-se uma redução no número de instruções de até 42% (figura 3.13), em relação ao código completo dos blocos básicos em questão – que de outra forma deveria ser integralmente executado pelo FemtoJava.

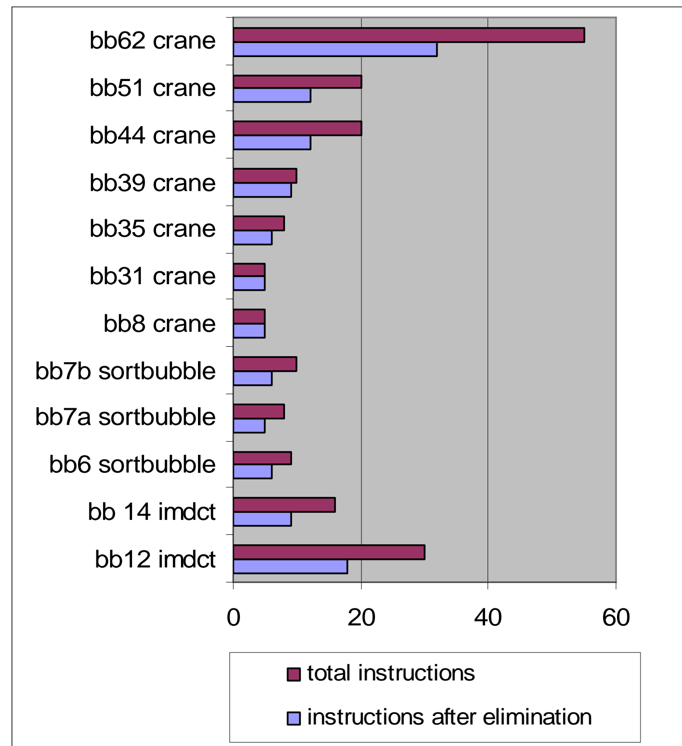


Figura 3.13: Ocorrem reduções de até 42% no número total de instruções necessárias a executar com o Javarray.

Todas as observações realizadas a partir de análise do profile das aplicações embarcadas descritas foram consideradas na definição da topologia geral da arquitetura reconfigurável Javarray apresentada no capítulo a seguir, capaz de otimizar a execução dos blocos básicos mais significativos das aplicações embarcadas e possibilitar um aumento de performance ao mesmo tempo em que reduz o consumo de energia.

4 ARQUITETURA JAVARRAY

4.1 Módulos

Da análise dos grafos de dependência gerados para as aplicações da seção anterior (desconsiderando apenas as instruções-raiz *iastore* e *invokestatic*), todos eles possuem uma estrutura de árvore binária, ou podem ser transformados para tal (como nos casos em que ocorre uso intenso de instruções istore-iload, como no bloco 62 do Crane 16bits, por exemplo). Esse fato é verificado em todas as árvores dos blocos básicos analisados, e decorre do próprio formato e modelo de programação dos bytecodes Java, que são baseados em uma máquina de pilha.

Sendo assim, a primeira possibilidade considerada foi a criação de um substrato de pe's com roteamento fixo na forma de árvore binária, aliado a um conjunto de registradores com roteamento configurável para qualquer entrada de qualquer *pe*.

Ainda que esse seja um arranjo simples, facilmente mapeável e com muitas interconexões locais (que facilitam a questão do roteamento e economizam em área), poderia ocorrer grande desperdício de recursos – visto que o número de pe's cresce exponencialmente conforme a expressão (2^n – 1), onde n é a quantidade de níveis de profundidade da árvore.

Após as análises mais aprofundadas dos grafos de dependência obtidos, percebeu-se que a profundidade das árvores tratadas nunca ultrapassou 8 níveis, e que no máximo foram utilizadas 3 instruções simultaneamente num nível que necessitassem processamento de pe's, com exceção do caso do bloco básico 62 do Crane 16bits, que possui uma largura de 4 instrução. Ainda, considerando os níveis de dois em dois, tem-se a necessidade máxima da utilização de 6 registradores – de fato, dado a repetição de instruções de entrada de dados verificada, esse número poderia até ser menor. Percebe-se também que a quantidade de operações de acesso à memória é pequena – nunca ultrapassou 3 instruções nas árvores inteiras, e nunca ultrapassou dois acessos num mesmo nível.

Sendo essas diretrizes válidas para a maioria dos blocos básicos interessantes, pode-se assumir que um array de 3x8 pe's será suficiente para executar significativamente a maior parte das configurações desejadas. Dessa forma, obtém-se um arranjo compacto, capaz de executar a grande maioria dos mapeamentos interessantes – dos blocos básicos com elevado número de repetições, capazes de otimizar o desempenho e economizar em consumo de energia.

A topologia básica do Javarray é portanto descrita conforme a figura 4.1, a seguir:

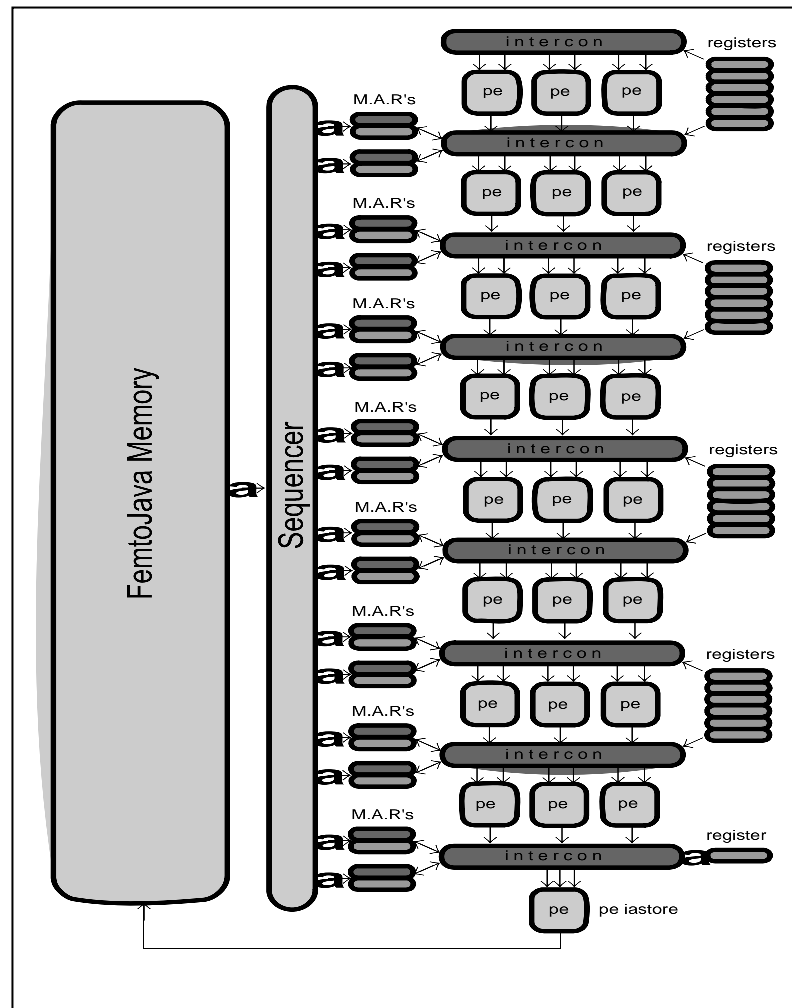


Figura 4.1: Topologia geral do Javarray.

Na figura 4.1 acima, observa-se, à esquerda, os registradores de acesso à memória (ou mar’s – memory access registers). No centro há o arranjo de 3 x 8 pe’s. Todos os pe’s possuem cada um 2 entradas e 1 saída. Há a possibilidade de conexão de cada saída de um nível com qualquer uma das entradas dos pe’s do nível imediatamente subseqüente.

A cada nível, há disponíveis dois mar’s, que aceitam uma única entrada, e que pode ser a saída de qualquer um dos pe’s desse mesmo nível. A saída desses mar’s pode ser configurada, através de mux’es, como entrada de qualquer uma das entradas dos pe’s do nível imediatamente subseqüente.

Os mar’s são estruturas pensadas para tratar da comunicação do array com a memória do FemtoJava, onde se faz a requisição do dado de algum endereço através de uma entrada, obtendo-se algum tempo depois o resultado em sua saída, transparentemente para o array.

Para efeito de organização do projeto, a cada dois níveis do arranjo, dá-se o nome de *estágio*. Cada estágio possui uma *ilha* de 6 registradores – define-se *ilha* como um conjunto de recursos disponível apenas localmente a qualquer pe do estágio em questão. Teoricamente, o conceito de *ilha* pode ser utilizado para acrescentar funcionalidades adicionais ao array, mas que sejam apenas eventualmente necessárias. Cada registrador da ilha de registradores pode ser configurado como qualquer uma das duas entradas de qualquer um dos 6 pe's do estágio.

Abaixo do arranjo de pe's, há um pe especial "iastore" que representa a execução de instruções-raíz iastore, sendo o único capaz de executar operações com 3 entradas, onde uma dessas entradas é um registrador específico. Essa restrição se deve ao fato de uma grande ocorrência da instrução *iastore* como instrução-raíz, e de que em todos esses casos uma dessas três entradas foi sempre uma instrução classificada como do tipo "registrador". Essa pe é a única capaz de escrever um resultado na memória. Idealmente, quer-se um arranjo tal que a computação nos pe's ocorra tão logo suas entradas estejam disponíveis e válidas, culminando no processamento do *pe iastore*.

Ao lado da memória, há um Seqüenciador, utilizado para sequencializar acessos à memória, potencialmente concorrida simultaneamente por mais de uma pe do arranjo. A prioridade será sempre dada à uma pe mais acima, uma vez que o fluxo de computação ocorre de cima para baixo, como será visto adiante. Durante o fluxo de processamento, são permitidas leituras à memória (pelas instruções *iaload*). A escrita em memória, como já foi dito, pode ser realizada apenas através do *pe iastore*.

4.2 Restrições

A topologia geral do Javarray não se preocupa com o ajuste fino dos parâmetros do array, como o número de registradores em cada *ilha* ou o número de mar's por nível, ou até mesmo a profundidade e largura do array. A preocupação foi de buscar um limite superior razoável nas necessidades arquiteturais, mais do que de esgotar as possibilidades de otimização. Muitas das definições da topologia geral do Javarray podem ainda ser otimizadas.

Como exemplo, muitas das instruções de pré-processamento ocorrem repetidas em diversos pontos numa configuração de um bloco básico, podendo até ser que 6 registradores em cada ilha seja um número superior ao necessário. No caso dos mar's, poderia-se, pelas análises anteriores, fazer 2 mar's para cada estágio, ao invés de 2 por linha, ou talvez até 2 mar's para todo o array. Também, limitou-se a execução da célula-raíz para a instrução de iastore, ignorando-se as instruções de comparação. Ainda, o mecanismo de roteamento também pode ser otimizado.

Essas escolhas justificam-se: em primeiro lugar, não há um interesse inicial em estressar o desempenho do array – deseja-se obter conceitos e medidas gerais, para posterior ajuste; em segundo lugar, deseja-se obter resultados rapidamente – assim, a escolha de se utilizar 2 mar's a cada linha ao invés do array antecipou a etapa dos testes dos mar's quando da descrição vhdl, economizando tempo de projeto, o mesmo ocorrendo com a limitação da funcionalidade da célula-raíz para a instrução de iastore. Esses relaxamentos de projeto não são significativos para a obtenção dos propósitos de análise da arquitetura, mas foram significativos na redução do tempo de projeto dos experimentos. É bom ter em mente, entretanto, que esses parâmetros podem ser

posteriormente melhor ajustados. Procurou-se sempre errar *para mais*, nas estimativas durante todo o fluxo de desenvolvimento do projeto, quando fosse o caso.

As organizações arquiteturais implementadas, descritas na seção 4.6 , possuem ainda algumas restrições práticas adicionais. Por exemplo, as unidades lógico-aritméticas utilizadas para a criação dos pe's não executam todas as operações necessárias para a execução dos blocos básicos apresentados. Essas restrições se devem ao fato de ter-se reaproveitado código de ula's já utilizadas em outros projetos, tanto em vhdl quanto no caco-ps. Essas restrições foram percebidas tardiamente no fluxo de desenvolvimento, mas não trazem maiores impedimentos para as análises de consumo e desempenho que se seguem. As análises, apresentadas no capítulo 5, contornam esse problema simulando a execução de tais operações não suportadas com outras suportadas. Assim, por exemplo, a instrução de shift é considerada a grosso modo equivalente a uma instrução de adição.

Adicionalmente, as instruções-raiz de comparação (*if's*) são simuladas como instruções de subtração, processadas nas pe's iastore; e as instruções *istore*, que escrevem em variáveis, são consideradas idênticas a instruções *iastore*, já que no FemtoJava Multiciclo a pilha é implementada em memória. Há possivelmente um erro *para mais* aqui, em termos de consumo, quando se considerar a implementação da pilha à parte à memória.

Também, as instrução *ireturn* e *invokestatic* são descartadas dos grafos para mapeamento no array, como já citado anteriormente. Isso se deve ao fato de o comportamento de tais instruções ser mais complexo. Assim, o Javarray considera que tais instruções devam ser executadas pelo processador hospedeiro – no caso, o FemtoJava.

4.3 Classificação

A arquitetura Javarray é uma Arquitetura Reconfigurável de Granularidade Grossa. Optou-se por essa alternativa por dois motivos principais: primeiramente, se irá realizar a otimização de bytecodes Java através da reorganização de blocos básicos, ou seja, serão manipuladas palavras no nível de instrução; em segundo lugar, sendo que os blocos básicos, por definição, não possuem *loops* para trás, a necessidade de interconexões é altamente localizada, já que não se necessita reaproveitar tais interconexões para muitos pontos do array, o que favorece a sua utilização em nível de palavra.

Na medida em que o Javarray otimizará apenas alguns trechos da execução de um aplicativo, não é uma arquitetura independente, e necessita de um hospedeiro, ao qual estará Fortemente Acoplada, pois sua comunicação com o mesmo é intensa e crítica.

Quanto ao escopo de aplicações, o Javarray destina-se a execução de Aplicações Embarcadas de Propósitos Gerais, e não apenas aplicações vetoriais, processamento de sinais ou de alto paralelismo como a maioria das arquiteturas reconfiguráveis.

O Javarray faz uso de tradução binária para tranformar trechos de bytecode Java em novas instruções, fazendo uso do conceito de Instruções Reconfiguráveis.

Por fim, as características dos blocos a serem otimizados permitem a disposição dos recursos em formato de Array Reconfigurável executado em um fluxo unidirecional, o

que reduz a complexidade de roteamento, reduz a complexidade de projeto e aumenta a localidade de recursos computacionais.

4.4 Processamento no Javarray

4.4.1 Fluxo de Configuração

A seleção dos blocos básicos mais representativos a serem otimizados depende de uma análise anterior (estática) ou em tempo de execução (dinâmica) da aplicação em análise. Esses blocos básicos são separados em seus blocos de operandos, que constituem árvores de dependência de instrução com raízes independentes entre si, onde cada uma é mapeada numa instrução reconfigurável Javarray.

No caso das aplicações embarcadas, onde se tem um conhecimento prévio sobre a aplicação, a análise estática não é um obstáculo. Entretanto, apesar de o presente trabalho tratar de reconfiguração estática, uma abordagem dinâmica foi sempre intencionada, e, justamente pela estrutura baseada em pilha dos bytecodes Java, o mapeamento dos blocos não é complicado (ACHUTHARAMAN, 2003).

No grafo de dependências, cada operação de entrada de dados que ocorre replicada é executada apenas uma vez. Algumas dessas operações possuem valores constantes durante todo o tempo de execução da aplicação, como *iconsts* e *bipushs*, por exemplo. Usualmente, uma parcela bem menor de operações representa valores de entrada variáveis a cada execução do bloco básico, como *iloads*. Essas operações variáveis são separadas e devem ser executadas pelo processador hospedeiro antes da execução do trecho reconfigurado do bloco no Javarray. Para as operações internas ao grafo de dependências, todos os pares de instruções como istore-iload e *putstatic-geststatic* que podem aparecer, sob certas circunstâncias, podem ser eliminados, e as operações restantes são mapeadas no substrato reconfigurável. Essas considerações acerca das instruções presentes num bloco de operandos pode ser melhor verificada na figura 4.2 adiante.

O mapeamento das configurações dos blocos básicos no array se dá sempre a partir da instrução-raíz em direção às folhas – ou seja, possui um fluxo “de baixo para cima”, com relação à topologia descrita pela figura 4.1 e conforme demonstrado na figura 4.3 mais adiante. Isso porque as configurações possuem tamanho variável, e geralmente não ocupam toda a extensão do array, fazendo com que os pe’s mais ao topo possam eventualmente não participar de uma dada configuração, sendo desligados para poupar energia. Eventualmente, as instruções podem realizar uma leitura à memória, mas o fluxo continua sendo unidirecional – apenas o último pe é capaz de escrever na memória.

O espaço de 3x8 pe’s pode eventualmente não ser suficiente para acomodar uma configuração completa – mesmo que as estatísticas obtidas no capítulo 3 demonstrem que a grande maioria dos blocos de interesse serão suportados. Mesmo para blocos maiores, ainda é possível fazer um particionamento temporal dos grafos de dependência, mapeando blocos de operandos menores e independentes em momentos distintos no array, e passando os resultados através dos registradores gerais, de uma configuração à outra.

Existem algumas etapas para realizar-se o mapeamento de um bloco básico para o Javarray, descritos a seguir.

4.4.1.1 Reordenamento

As folhas são sempre instruções pré-processadas na etapa de pré-processamento (como já explicado), cujo valor é então armazenado nos registradores dos estágios. As demais instruções, entre as folhas e a raiz da árvore, são instruções que executam nos pe's, podendo haver a leitura de algum valor de memória nesse intervalo. Ao se identificar pares de instruções do tipo *getstatic-putstatic* e iload-istore, por exemplo, no interior da árvore, pode ocorrer uma supressão de instruções, implicando no reordenamento da árvore antes de um mapeamento no array.

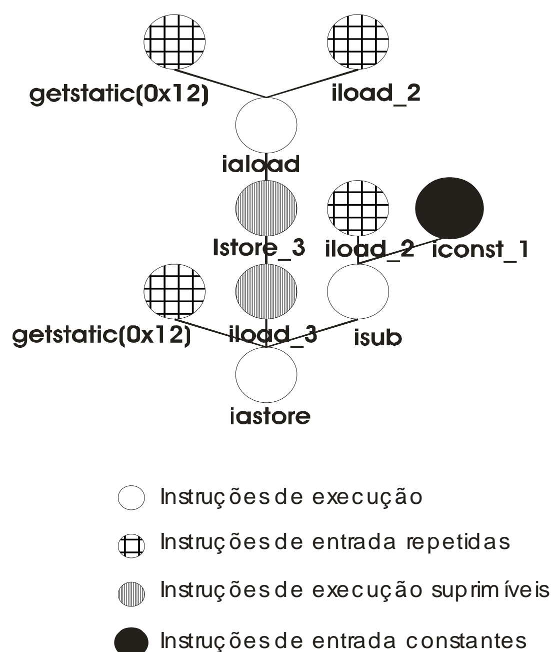


Figura 4.2: Simplificações de um bloco de operandos.

Perceba-se que, das 10 instruções originais do bloco de operandos exemplificado na figura 4.2, apenas 5 instruções deverão ser executadas a cada repetição do bloco, já que 2 instruções ocorrerm repetidas, 2 instruções são suprimidas e 1 instrução é constante.

Também pode ocorrer a necessidade de particionamento do bloco, como explicado anteriormente, no caso de o espaço de 3x8 pe's ser insuficiente.

4.4.1.2 Mapeamento

As instruções são mapeadas em dois locais distintos: instruções de entrada localizam-se sempre nos registradores. Seu valor – seja ele constante ou variável – é carregado para um registrador específico, num estágio adequado.

As instruções de processamento localizam-se nos pe's. Como já explicado anteriormente, o mapeamento ocorre a partir da instrução-raíz para as folhas. Isso

porque os blocos básicos possuem tamanhos distintos, e sempre a última pe a executar numa configuração deve ser, por definição do array, a pe especial, capaz de escrever na memória se for necessário.

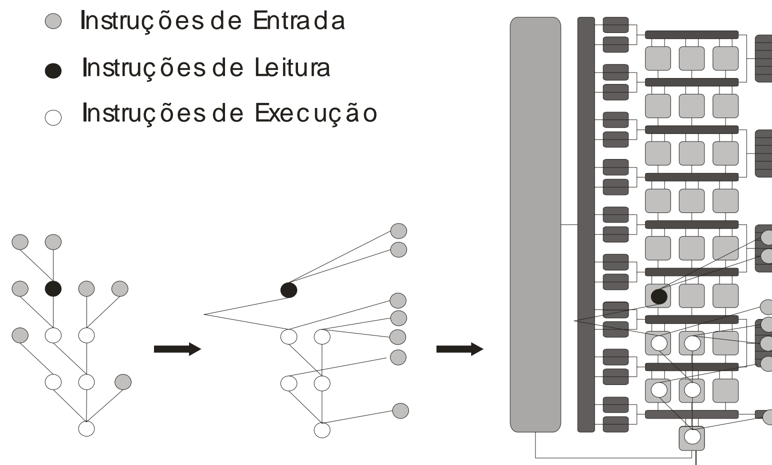


Figura 4.3: Mapeamento de um bloco de operandos no Javarray.

Perceba-se que pode haver *pe's* não utilizadas – fragmentação interna do array, para fazer uma analogia à fragmentações de memória. Esses recursos não utilizados não devem interferir no resultado e idealmente também não deve interferir no consumo do array.

Após a configuração das operações dos pe's e dos valores de entrada dos registradores, deve-se realizar a configuração dos roteamentos entre os registradores e as respectivas entradas nos pe's; bem como o roteamento das saídas dos pe's com as entradas dos pe's do nível subseqüente específicas à configuração em questão. Quando necessário um acesso à memória, deve-se realizar o roteamento de saída de um pe para a entrada em um *mar* específico, e novamente o roteamento da saída desse *mar* à entrada de um pe posterior que consuma esse resultado de acesso à memória.

Da forma como a arquitetura foi descrita (tanto em VHDL quanto em CACO-PS), as interconexões de roteamento do Javarray permitem o roteamento de um mesmo registrador para qualquer entrada de qualquer pe do estágio – inclusive simultaneamente para as duas entradas de um mesmo *pe*. Na prática, tais situações não ocorrem, e maiores restrições podem ser impostas. A generalidade permitida pelo mecanismo de roteamento do Javarray sabidamente desperdiça recurso de área. Entretanto, como área não foi o foco do projeto, e principalmente por questões de simplificação do mesmo, maiores restrições não foram impostas.

4.4.1.3 Pré-processamento

Convencionou-se chamar a etapa de preparação das instruções de entrada como *etapa de pré-processamento*, e assim será usado no decorrer de todo o restante deste trabalho. Essa preparação nada mais é do que a determinação dos valores que devem ser

configurados nos registradores de entrada de cada estágio. Alguns desses valores são constantes, e após definidos uma vez são imutáveis para todas as demais execuções da configuração. Entretanto, alguns desses valores são variáveis à cada execução, e devem ser apropriadamente calculados e/ou buscados antes de cada execução da configuração em questão. Essas instruções variáveis são separadas e devem ser executadas pelo processador hospedeiro antes que seja disparada a execução da instrução reconfigurável.

4.4.1.4 Pós-processamento

Após a execução de uma configuração pode ser necessário executar instruções pendentes que gerem dependências de dados com outros blocos da aplicação. É o caso de instruções *putstatic* e *istore*, por exemplo. Da mesma forma, essas instruções são separadas para serem executadas pelo processador hospedeiro, mas agora, *após* a execução do trecho reconfigurado. Outros casos que também são separados para execução pelo hospedeiro são instruções mais complexas não suportadas pelo array, como o ireturn e o invokestatic – raízes de árvores de dependências.

4.4.2 Fluxo de Execução

Além da redução no número de instruções pela eliminação de instruções de entrada de dados replicadas impostas pelo uso da pilha, e pela supressão de pares de instrução do tipo iload-istore, a reconfiguração dos blocos básicos também propicia economia de energia nos estágios de busca e decodificação das instruções, porque blocos básicos não possuem desvios internos.

Um bloco básico é composto por um ou mais blocos de operandos, usados como unidades de reconfiguração, e que são bem definidos a partir dos endereços de sua primeira e última instruções. Possuindo previamente o mapeamento de um bloco para o substrato reconfigurável, a busca do primeiro endereço correspondente a primeira instrução de um bloco dispara a execução do pré-processamento e, logo em seguida, dispara a execução do bloco reconfigurado no Javarray. Com a identificação do endereço de início de um bloco de operandos, a busca e a decodificação das demais instruções do bloco no processador hospedeiro não são mais necessários.

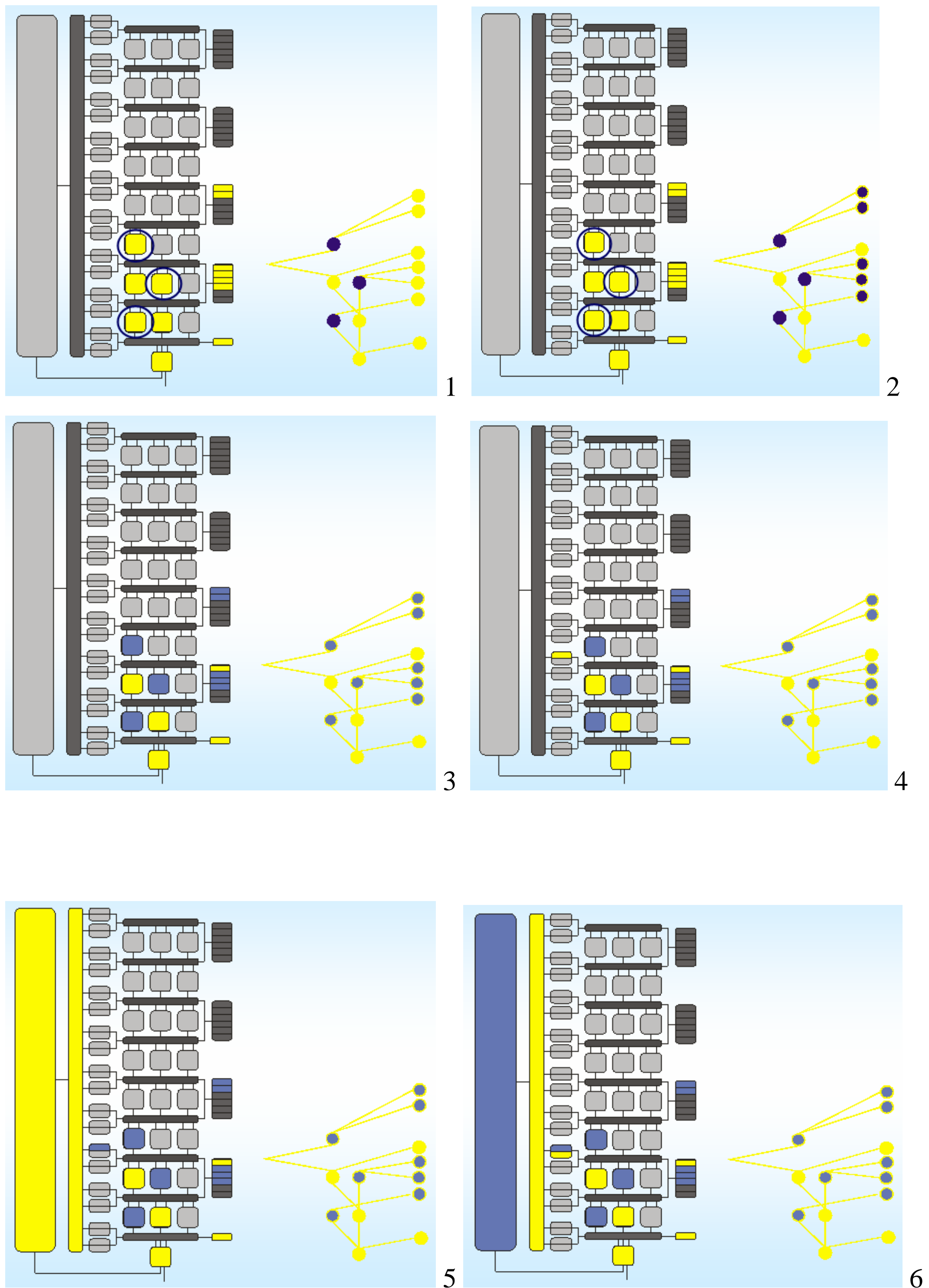
Depois da execução da configuração e do pós-processamento, o fluxo do programa deve ser ajustado para a próxima instrução depois do bloco básico em questão, voltando então ao processador hospedeiro. O código original da aplicação não é alterado.

Com a eliminação dos ciclos de busca e decodificação, salva-se energia e ao mesmo tempo aumenta-se a performance. Além disso, o Javarray captura o ILP intrínseco dos grafos de dependência, aumentando ainda mais a performance.

O fluxo de execução é oposto ao fluxo de configuração – ele ocorre do topo para baixo, em relação a figura 4.1. Os pe's configurados executam tão logo suas dependências tenham sido resolvidas. Os primeiros pe's a executar serão aqueles cujas entradas vêm dos registradores de entrada globais. Eles processam em paralelo e geram resultados de entrada para os pe's posteriores.

A inexistência de desvios (por definição) nos blocos básicos, permite a execução das configurações do array num fluxo unidirecional contínuo onde, entretanto, permite-

se acessar leituras de memória. A restrição adicional de se permitir apenas que a última pe seja capaz de escrever na memória, colabora ainda mais nesse processo. O fluxo de processamento constitui-se, portanto, na execução de um grafo no formato de árvore binária (a menos da instrução iastore, como já explicado) que parte das folhas e culmina na execução da raiz, conforme demonstra a sequência da figuras 4.4 abaixo.



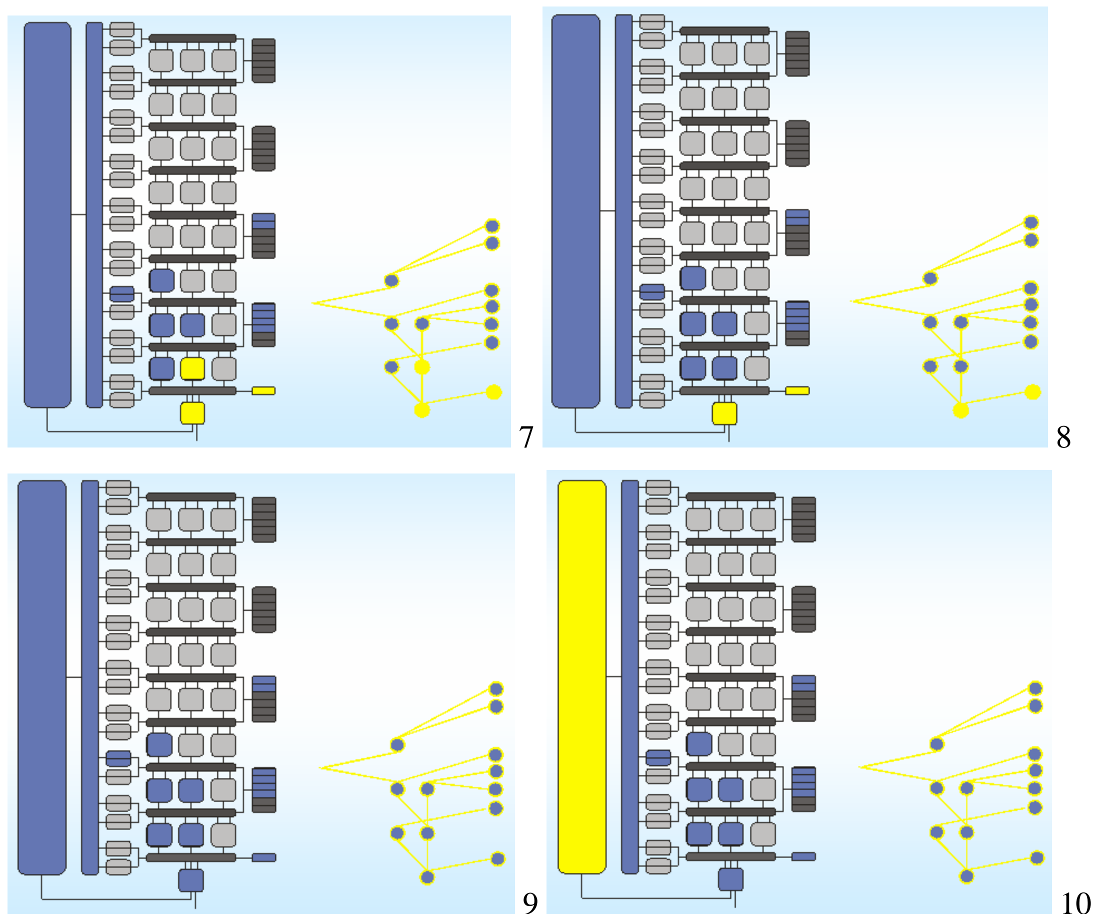


Figura 4.4: Sequência de figuras demonstrando o fluxo de execução: sequências 1 e 2 indicam os primeiros pe's prontos para executar; sequências de 3 a 10 demonstram a execução de todo o bloco reconfigurado.

Como o mapeamento de uma configuração no array pode introduzir *fragmentação* – no sentido de que algumas pe's podem ser desperdiçadas para uma dada configuração –, não se pode estabelecer um início fixo para o processamento do array: as pe's iniciadoras do processamento podem variar dependendo do bloco básico mapeado no array. Assim, as organizações da arquitetura Javarray, posteriormente descrita, devem tratar essa questão de controle.

4.5 Acoplamento ao FemtoJava

O Javarray foi pensado para ser uma unidade funcional fortemente acoplada ao FemtoJava – ou qualquer outro processador nativo Java. O mecanismo de integração, entretanto, não foi escopo deste trabalho, e não foi implementado. Cuidou-se para deixar um bit indicando a conclusão da computação do array, para futura integração com o FemtoJava.

A arquitetura Javarray servirá como substrato para a tradução binária dos códigos escritos para o FemtoJava, servindo ao propósito de aumento de desempenho das aplicações ao mesmo tempo em que economiza energia. Isso se dá pelo fato de que o Javarray é capaz de reconfigurar uma seqüência de bytecodes em uma única instrução no array. Ainda que o atraso dessa reconfiguração possa ser grande, é menor que a soma

individual das instruções, e se a instrução reconfigurada for repetida um certo número de vezes a economia de energia é significativa, já que ocorrerão menos acessos à memória e um número menor de interações no datapath do FemtoJava serão necessárias, além das economias em busca e decodificação, e a redução do número de instruções de entrada de dados e instruções suprimidas internamente aos grafos.

Duas abordagens seriam possíveis para a integração do Javarray ao FemtoJava. Utilizando-se o Javarray com uma reconfiguração dinâmica, apenas uma instância do mesmo seria necessária, mas um mecanismo de identificação, mapeamento e armazenamento das configurações não foi implementado. Neste trabalho, portanto, utiliza-se uma abordagem estática para efeito de análise dos resultados, onde para cada configuração existe um Javarray dedicado, conforme demonstrado pela figura 4.5.

Na integração do Javarray com o FemtoJava na forma estática, os profiles das aplicações a serem rodadas são analisados *off-line*, os melhores blocos básicos são selecionados e instruções reconfiguradas para esses blocos básicos são armazenadas em uma memória cache ou mesmo estaticamente em réplicas do array. Por simplificação, para não ter que tratar do consumo de energia referente às buscas dos streams de configuração na memória, considera-se que se está utilizando arrays replicados estaticamente configurados para cada bloco reconfigurado em questão. Apesar disso, os resultados obtidos demonstram haver margem para a utilização de memória de configurações e até mesmo para um mecanismo dinâmico.

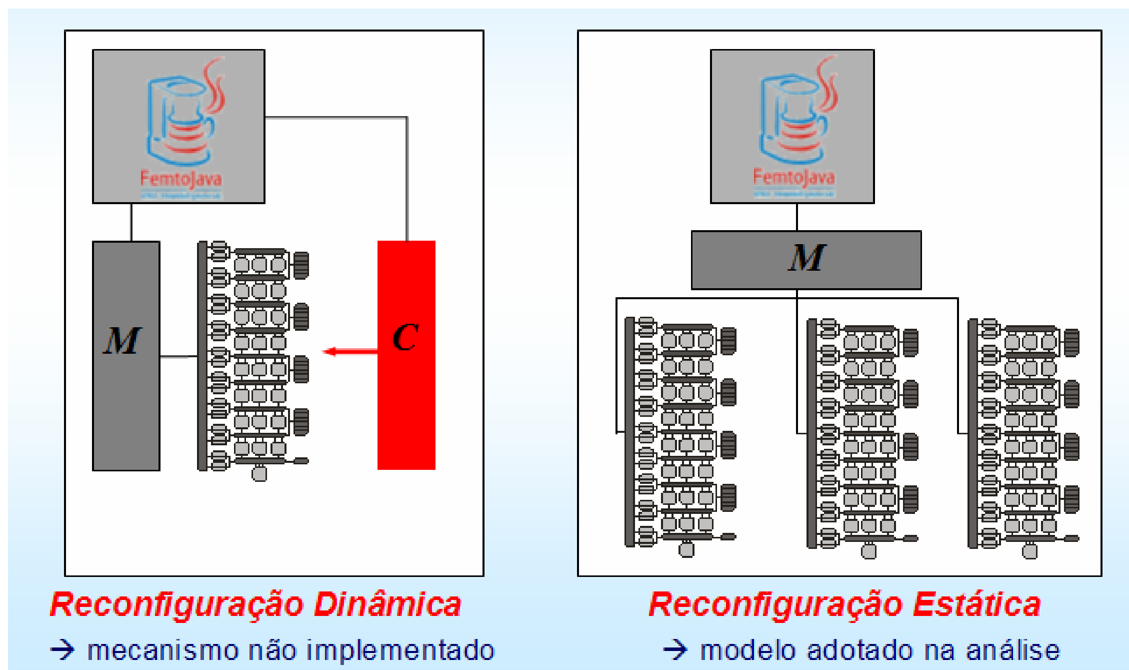


Figura 4.5: Integração Femtojava-Javarray como unidades funcionais replicadas.

Quando o FemtoJava identifica o endereço correspondente a uma dessas instruções reconfiguradas, o Javarray é chamado e o FemtoJava espera por sua conclusão.

4.6 Organizações Arquiteturais do Javarray

4.6.1 Organização Javarray Sequencial

Dado o objetivo principal da arquitetura Javarray de economia de energia através do aumento de desempenho, a idéia original almejava utilizar componentes combinacionais. Desse forma, realizaríamos a computação no seu tempo mínimo e não consumiríamos energia com registradores, então, desnecessários.

Para tanto, os pe's seriam ula's ligadas aos seus respectivos registradores de uso geral, com multiplexadores em suas entradas de dados para determinar a fonte dos mesmos – conforme a topologia básica do Javarray, já explicada no capítulo 4. Os sinais que determinam os valores de multiplexação fariam parte da configuração do bloco básico no array.

Ainda, como a arquitetura seria combinacional, faría-se necessário determinar um tempo em que garantidamente o resultado de saída se tornaria estável. Como a arquitetura Javarray define o mapeamento de uma árvore de dependências no array, esse tempo é determinado pela análise do caminho crítico dessa árvore.

Entretanto, ainda que num bloco básico, por definição, não ocorram desvios internos, e mesmo que consideremos que escritas seguidas de leituras em variáveis (que na JVM utilizam memória) possam ser suprimidas quando mapeadas para a arquitetura Javarray, deve-se perceber que ainda podem ocorrer dependências de memória (leituras) durante o processamento das instruções internas do bloco básico.

Essas leituras em memória tornam imprevisível o tempo de processamento do bloco básico mapeado, visto que podem ocorrer *cache misses*. Sendo assim, não temos como determinar o tempo de estabilização do resultado do array previamente.

Essa dificuldade levou à determinação de uma organização com componentes seqüenciais para a arquitetura Javarray. A solução desenvolvida para superar o problema do *cache miss* foi orientar a arquitetura para um fluxo de dados, incluindo-se registradores nas entradas das *ula's*, conforme mostrado na figura 4.6.

Nesses registradores e ainda nos registradores de uso geral (utilizados para alimentar o array, e configurados através do pré-processamento, como já explicado na seção anterior), incluiu-se um bit como indicador de "dado pronto".

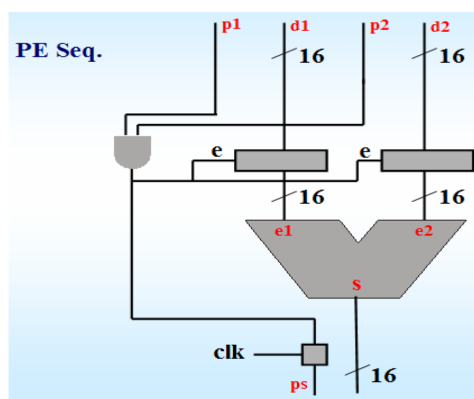


Figura 4.6: PE do Javarray Sequencial.

As pe's apenas processam se ambas as suas entradas estiverem "prontas", também ligando, então, um sinal de pronto para o seu resultado. Quando ocorre uma leitura à memória, um sinal de pronto é da mesma forma retornado juntamente com o resultado da leitura.

Através das interconexões de roteamento (utilizando multiplexadores, como já descrito anteriormente), os valores das leituras em memória e dos processamentos das pe's podem ser associados à uma entrada específica de uma pe subseqüente no fluxo de processamento do array. Esse processo prossegue até alcançar-se a computação da instrução-raíz da árvore de dependências do bloco básico, cuja saída de "pronto" indicará que o resultado da computação do bloco básico configurado no array está totalizada e estável.

Assim, não existe mais o problema da indeterminação do momento de conclusão da computação, pois há um bit indicativo par tal. O problema de acesso à memória também é superado, pois, da mesma forma, um bit de pronto acompanhará o resultado da leitura tão logo ele tenha sido buscado através da hierarquia de memória.

Orientando-se a arquitetura ao fluxo dos dados consegue-se simplificar o controle de execução das pe's, visto que esse controle é determinado pelo próprio processamento do bloco básico, sem necessidade alguma de considerações adicionais. Também, fazendo-se com que as pe's computem apenas quando ambas as suas entradas estiverem prontas, cada pe consumirá no máximo apenas uma vez durante a execução do bloco básico, sendo que as pe's não participantes da configuração do bloco básico podem até mesmo ser desligadas, poupando ainda mais energia.

A orientação do processamento ao fluxo de dados, com a utilização dos bits de pronto resolve trivialmente também o problema da determinação de quais pe's devem iniciar a computação – derivado do fato de o mapeamento dos blocos básicos ocorrer a partir da *pe-iastore* para cima, sendo que blocos básicos distintos ocupam um número também distinto de pe's no array, e as pe's que iniciam a computação podem não ser as mesmas em todos os casos, conforme visto em seção anterior.

Com esta organização sequencial, o mapeamento pode se dar normalmente e as pe's que não estiverem fazendo parte da configuração não irão interferir no processamento, pois não terão dados prontos em suas entradas. As pe's que participam da computação iniciarão tão logo suas duas entradas estejam prontas. Não precisamos determinar explicitamente quais as pe's iniciam a computação. Basta apenas preparar os dados de pré-processamento nos registradores dos estágios e roteá-los para as pe's adequadas, determinadas no fluxo de mapeamento.

Logo após as duas entradas estarem prontas, o pe irá computar o resultado, que estará disponível para os pe's subseqüentes no próximo ciclo. O mecanismo utilizado para produzir o sinal de pronto computado no pe utiliza um *and* dos sinais de prontos das duas entradas para *setar* um registrador de 1 bit que indica o sinal de pronto do pe, que também no próximo ciclo estará disponível para os níveis subseqüentes, indicando que a computação desse pe já foi realizada e o resultado está válido. Assim, beneficia-se do fato do tempo de computação determinístico (1 ciclo) do pe para evitar protocolos mais complexos de orientação da computação aos dados.

A versão sequencial aqui descrita possui o mesmo formato geral descrito na seção 4.1, com 3x8 elementos. A versão sequencial, portanto, não economiza área em

relação a versão combinacional. Consome mais, inclusive, por conta dos registradores adicionais. Ou seja, a versão sequencial aquela descrita não tenta reaproveitar temporalmente uma linha de 3 elementos do array em 8 ciclos distintos para simular o espaço 3x8. De fato, os 3x8 elementos existem fisicamente. O reaproveitamento temporal de uma linha de pe's poderia reduzir a área ocupada e levar a resultados próximos, embora inferiores, aos alcançados pela versão combinacional; entretanto, essa alternativa não foi implementada.

Além de resolver os problemas já citados, a existência de registradores nas entradas das pe's numa estrutura física 3x8 sugere o aproveitamento do array para a implantação de técnicas de pipeline para execução de computação baseada em *stream*, aumentando a utilização do array, fazendo com que o impacto do consumo estático seja reduzido ao mesmo tempo em que se aumenta a performance, como será demonstrado em no capítulo 5.

O esquemático da organização sequencial do Javarray foi validado e implementado em VHDL e em CACO-PS, sendo que os códigos encontram-se no cd de anexos. O modelo VHDL desenvolvido é estrutural em nível de transferência entre registradores, e altamente parametrizável.

4.6.2 Organização Javarray Combinacional

4.6.2.1 Organização Combinacional Restrita

Na versão combinacional, não existem registradores nas entradas dos *pe's*. As saídas de um *pe*, bem como os registradores globais, são mapeados diretamente para as entradas de outros *pe's*. Assim, um mesmo pe pode consumir dinamicamente várias vezes, tantas quanto forem as variações de suas entradas no decorrer do fluxo de processamento. No entanto, elimina-se o atraso dos registradores e todo o consumo estático que eles acarretam.

O problema da estabilidade

Como visto em seção anterior, a arquitetura Javarray permite leituras à memória no decorrer de seu fluxo de processamento. De fato, permite com que vários pe's concorram simultaneamente pelo acesso à memória, sendo tais acessos sequencializados no tempo. Adicionalmente a isso, cada acesso à hierarquia de memória pode levar um tempo indefinido, pois potencialmente podem ocorrer *misses* na busca dos dados.

O problema, portanto, é determinar quando, numa arquitetura combinacional, os resultados dos pe's estão estáveis, sendo que as entradas dos pe's podem ser resultados de potenciais leituras à memória. Propagando a mesma idéia para todo o array, há o problema de saber quando a computação da configuração está concluída e qual o momento em que uma pe deve realizar uma leitura à memória.

A seguir, desenvolve-se um processo para permitir computação combinacional na topologia Javarray. Ainda que a solução combinacional restrita não seja tão genérica, é obviamente mais rápida e de mais fácil implementação do que a organização sequencial. Sua motivação deriva da observação de que o elevado número de registradores utilizados na organização sequencial consome muita potência estática.

Espera-se que a redução no número de registradores venha a melhorar o consumo de energia do array. Perde-se, entretanto, a possibilidade de uso de técnicas de pipeline.

A solução combinacional restrita considera que nunca ocorram misses na memória. Dito de outra forma, considera-se que existe apenas uma única memória, contendo toda a aplicação. Essa consideração é aplicável ao domínio dos sistemas embarcados, onde certas aplicações não necessitam de uma hierarquia de memória.

Solução

O princípio geral utilizado é considerar cada acesso à memória como possuindo um tempo determinístico e tentar prever o tempo de estabilização do sinal de saída. Esse tempo é facilmente previsível, pois os pe's são idênticos e replicados. Supondo que não estamos usando nenhum tipo de protocolo assíncrono, consideremos que existe um registrador de um bit, representando "sinal pronto", apenas no último estágio de processamento da arquitetura, que informa a estabilização do resultado da computação do array.

Assume-se que esse bit é *setado* ao final de um tempo previsto (através da análise antecipada do número de estágios da configuração e também considerando os acessos à memória como se não ocorresse o *cache miss*), como verificado na figura 4.7. Tal bit pode ser utilizado futuramente para o chaveamento entre Javarray e FemtoJava, quando o mecanismo de integração for implementado.

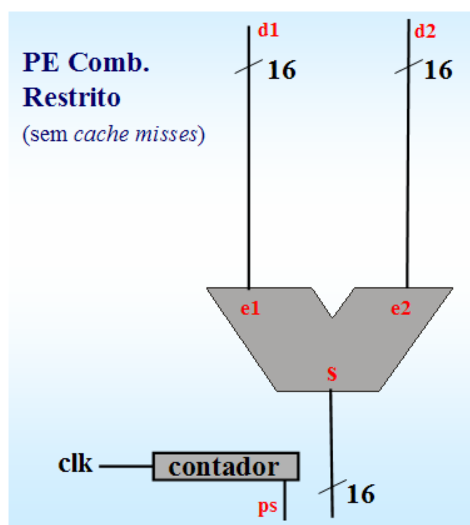


Figura 4.7: PE do Javarray Combinacional Restrito.

Consideremos agora apenas o acesso à memória: como saber o momento em que o resultado de uma *pe*, que acessa um endereço de memória, está estável para que esse dado seja de fato buscado? A solução encontrada é a mesma encontrada para o sinal de pronto do array: o momento do acesso à memória deve ser previsto antecipadamente.

A solução às questões de conclusão da computação do array e dos acessos à memória é implementada com o uso de registradores-contadores, que saturam num valor final pré-determinado indicando a estabilização dos sinais.

O tempo previsto para uma dada configuração do array é contado através de um registrador, que recebe *ticks* de um *clock* correspondente ao período de execução de um *pe*. A cada *tick* o registrador recebe um incremento. Cada pe possuirá o seu mecanismo de registrador-contador, que apenas estará ligado caso essa *pe* necessite realizar alguma leitura em memória (ou escrita, no caso da *pe* especial).

A necessidade de se prever antecipadamente os momentos de acesso à memória e o tempo de conclusão da computação no array, para *setar* o valor dos registradores-contadores, introduzem uma complexidade adicional à lógica de configuração (presente na etapa de mapeamento da configuração no array), que na organização sequencial estava implícita no próprio array, através do fluxo dos dados. Note-se que agora devemos garantir, através da configuração desses registradores-contadores, que nunca haverá uma colisão de acessos à memória. Em outras palavras, deve-se efetuar um escalonamento dos acessos à memória.

Para um mapeamento estático das configurações no array, isso não é problema. Entretanto, pode introduzir uma complexidade adicional numa alternativa de mapeamento dinâmico (via hardware) dessas configurações.

Esse mecanismo funciona bem quando *não* se está utilizando uma hierarquia de memórias, para que o acesso à memória seja previsível – caso que pode não ocorrer quando cache misses são introduzidos. A essa organização, chamou-se de organização Combinacional Restrita. Em se tratando de sistemas embarcados, essa organização é viável em muitos casos.

Para simplificação da descrição CACO-PS e desprezando-se o consumo de energia de tal mecanismo, visto que poucas pe's acessam a memória numa configuração, desconsiderou-se também o detalhamento da descrição desse mecanismo. O mecanismo foi descrito em nível comportamental, implementado no componente "reg_cont" no arquivo de descrição de componentes cacops_modulos.c, que pode ser encontrado no cd em anexo.

4.6.2.2 Organização Combinacional Geral

O mecanismo tratado na seção 4.6.2.1 funciona bem quando não ocorrem misses. Entretanto, isso pode ocorrer numa arquitetura que possua hierarquia de memória. Assim, como o atraso de busca do dado pela hierarquia de memória é previamente desconhecido, não basta simplesmente determinar um momento específico para a leitura em memória. Um mecanismo adicional é necessário de tal forma que se impeça que o sinal de pronto seja gravado no bit de pronto até que o dado de fato tenha sido buscado com um *hit*.

Um ajuste do mecanismo do caso Restrito, que não funciona com hierarquia de memória, para o caso Geral, que o possibilita, é simplesmente parar a contagem de todos os registradores-contadores ativos no momento de um *miss* qualquer na *cache*, voltando a ativar a contagem simultânea de todos apenas quando o valor do dado for retornado, como visto na figura 4.8. Dessa forma, "esconde-se" o tempo da penalidade do array, e simula-se, desse ponto de vista, uma memória de apenas um nível – como o é no caso da versão combinacional restrita.

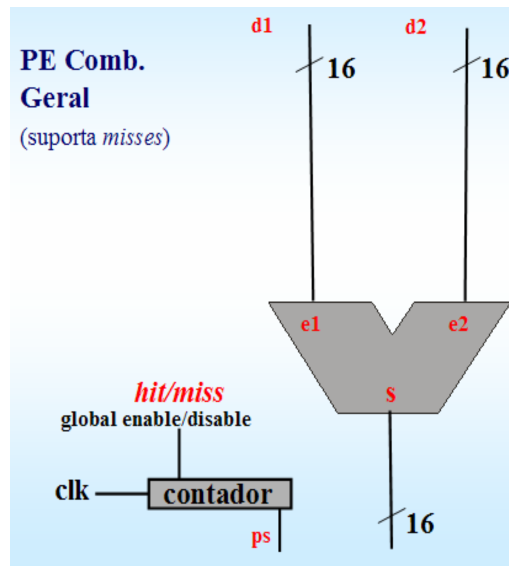


Figura 4.8: PE do Javarray Combinacional Geral.

Um modelo da versão restrita foi implementado em CACO-PS, mas não foi implementada a versão Geral. Para termos de análises do capítulo a seguir., ambas as versões são equivalentes, caso se considere que nunca ocorrem *misses*. Como realizaremos estimativas de alto nível, pode-se considerar que os dados obtidos para a versão combinacional restrita são idênticos aos da versão geral.

5 RESULTADOS

Os blocos básicos separados anteriormente para a determinação da arquitetura geral do Javarray foram avaliados sob as condições de desempenho e redução de consumo de energia, conforme a proposta da arquitetura.

A comparação da otimização produzida pelo Javarray utiliza o FemtoJava Multiciclo e o FemtoJava Low-Power (pipeline) como microcontroladores hospedeiros. O microcontrolador FemtoJava (ADÁRIO, 1997) é uma arquitetura baseada em pilha destinada a executar nativamente bytecodes Java. Características gerais do processador são: conjunto de instruções reduzido, arquitetura Harvard e pequeno tamanho. Esse processador foi específicamente desenvolvido para o mercado dos sistemas embarcados. A versão multiciclo do FemtoJava leva de 3 a 14 ciclos para executar uma instrução. A versão Low-Power (BECK, 2003b) utiliza um pipeline clássico de 5 estágios, mas com a presença adicional de registradores fazendo as vezes da pilha de operandos e variáveis locais (usados para manter valores das variáveis locais de um método), em vez de utilizar a memória principal para esse propósito, como é feito em outras máquinas de pilha encontradas na literatura.

Para analisar os resultados de energia e performance, modelos CACO-PS do Javarray em nível de transferência entre registradores foram desenvolvidos. O simulador de potência CACO-PS realiza uma aproximação em alto nível, considerando as transições dos bits simulando dinamicamente o consumo de potência no nível de portas lógicas, mas também considera o consumo de potência estática. Os resultados são dados em número de capacitâncias de gate que variam e podem ser calibrados para a tecnologia corrente.

A seguir, utiliza-se a aplicação SortBubble como exemplo da metodologia utilizada nos experimentos que foram realizados com as aplicações consideradas neste trabalho. Após isso, os resultados de performance, energia, potência e estimativas gerais de área são considerados. Por fim, demonstra-se a possibilidade de execução de computação baseada em stream no Javarray sequencial.

5.1 Experimentos

Como exemplificação dos métodos utilizados na análise das aplicações selecionados, apresenta-se a seguir passo-a-passo o experimento acerca da aplicação SortBubble. Para esse experimento, considerou-se os blocos básicos selecionados 6 e 7 da aplicação BubbleSort, já tratado em capítulo anterior. Os mesmos métodos utilizados na análise do SortBubble também foram aplicados na análise das demais aplicações.

Executou-se medições sobre o mapeamento desses blocos nas duas versões do Javarray – Sequencial e Combinacional Restrita –, e comparou-se o acoplamento dessas versões do Javarray à duas versões do FemtoJava – Multiciclo e Low-Power –, em relação à execução das aplicações em ambas as versões do FemtoJava sem o auxílio do Javarray.

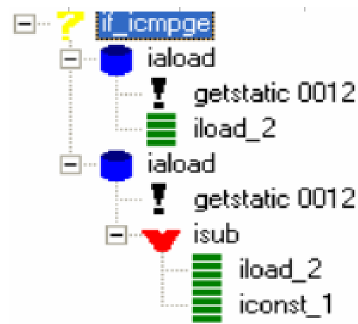
Prevendo-se um grande consumo estático de energia, aliado ao fato de se saber previamente que um array 3x8 é superdimensionado a todos os blocos básicos que se irá analisar, considerou-se as variações 3x8, 3x4 e 3x2 do Javarray em todas as demais análises, quando possível, sendo portanto uma forma mais justa de análise em se tratando de uma arquitetura com propósito aos sistemas embarcados – onde se conhece *a priori* algumas caracterísitcas das aplicações.

Em se tratando de um sistema embarcado, e para simplificar análises de consumo de memória de configuração e trocas de contexto – que neste trabalho não foram implementadas – está-se considerando arrays estaticamente mapeados e fixos para as respectivas configurações dos blocos considerados. Entretanto, os resultados obtidos demonstrarão haver margem para uma futura implementação de uma abordagem dinâmica.

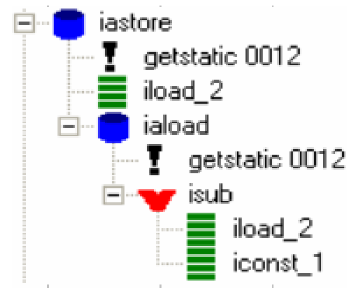
Então, com essas devidas considerações, realizou-se o mapeamento dos blocos de operandos 6, 7.a e 7.b para ambas as organizações Sequencial e Combinacional Restrita do Javarray (ambas 16 bits), além da execução do mesmo trecho de código na arquitetura FemtoJava Multiciclo 16 bits para comparações. Comparou-se os mesmos trechos de instruções com uma tabela de referência do FemtoJava Low-Power, também 16 bits, já utilizada em outras publicações e trabalhos do grupo, e que se encontra em anexo no cd.

O bloco básico 7.c foi desconsiderado para mapeamento no Javarray, por ter um tamanho muito pequeno, sendo que sua configuração não compensaria os possíveis ganhos, sendo portanto executado no processador hospedeiro. As árvores de dependência dos blocos básicos 6 e 7 estão repetidas na figura 5.1, para melhor clareza

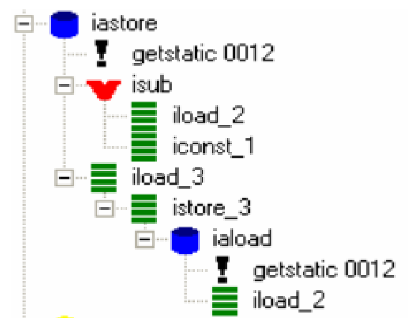
Dessa forma, pode-se realizar uma análise da otimização geral produzida pelo Javarray na execução do SortBubble, pois como já visto no capítulo 3, esses blocos básicos constituem a maior parte da execução do profile da aplicação. A seguir, detalha-se cada uma dessas análises.



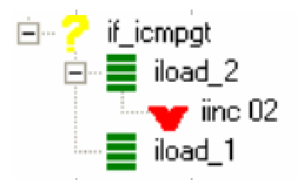
a) bloco básico 6



a) bloco de operandos 7.a do bloco básico 7



b) bloco de operandos 7.b do bloco básico 7



c) bloco de operandos 7.c do bloco básico 7

Figura 5.1: Árvores de dependência dos blocos básicos 6 e 7, com seus respectivos blocos de operandos.

Inicialmente, mapeou-se os blocos selecionados para a arquitetura Javarray em suas duas versões. As descrições arquiteturais CACO-PS para os experimentos realizados em todas as aplicações podem ser encontradas no cd de anexos.

Após isso, simulou-se a execução desses blocos no CACO-PS. Assim, os dados resumidos encontrados para os três blocos reconfigurados citados, quando rodados na organização Sequencial, encontram-se na tabela 5.1 abaixo (com energia dada em capacitâncias de gate por ciclo (cg's x ciclo) – métrica utilizada no CACO-PS). Note-se que foram feitos experimentos com 3 variações no tamanho do array: 3x8, 3x4 e 3x2 – mesmo que para o mapeamento desses blocos apenas um array 3x2 já seja suficiente, apenas a título de comparação.

Tabela 5.1: Estatísticas Javarray Sequencial.

bb 6 - Sortbubble – Sequencial	3x8	3x4	3x2
Ciclos	7	7	7
acessos à memória	2	2	2
potencia estática consumida	9,225	4,865	2,685
potencia consumida na memória	12,288	12,288	12,288
potencia consumida no array	65,894	35,374	20,114
potencia total consumida	78,182	47,662	32,402
bb 7.a - Sortbubble – Sequencial	**3x8**	**3x4**	**3x2**
Ciclos	8	8	8
acessos à memória	2	2	2
potencia estática consumida	9,225	4,865	2,685
potencia consumida na memória	12,288	12,288	12,288
potencia consumida no array	74,967	40,087	22,647
potencia total consumida	87,255	52,375	34,935
bb 7.b - Sortbubble – Sequencial	**3x8**	**3x4**	**3x2**
Ciclos	7	7	7
acessos à memória	2	2	2
potencia estática consumida	9,225	4,865	2,685
potencia consumida na memória	12,288	12,288	12,288
potencia consumida no array	65,561	35,041	19,781
potencia total consumida	77,849	47,329	32,069

Os mesmos blocos, agora rodados na organização Combinacional Restrita, mostram os dados encontrados na tabela 5.2.

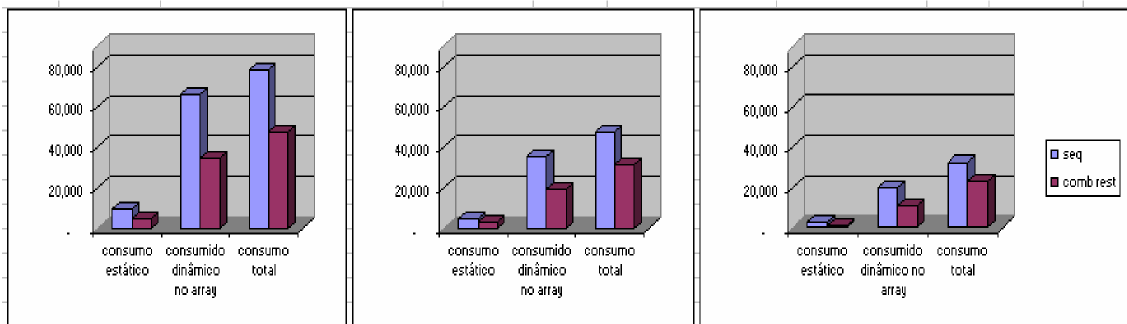
Tabela 5.2: Estatísticas Javarray Combinacional Restrito.

bb 6 - Sortbubble - Comb. Restr.	3x8	3x4	3x2
Ciclos	7	7	7
acessos à memória	2	2	2
potencia estática consumida	4,820	2,540	1,400
potencia consumida na memória	12,288	12,288	12,288
potencia consumida no array	34,752	18,792	10,812
potencia total consumida	47,040	31,080	23,100

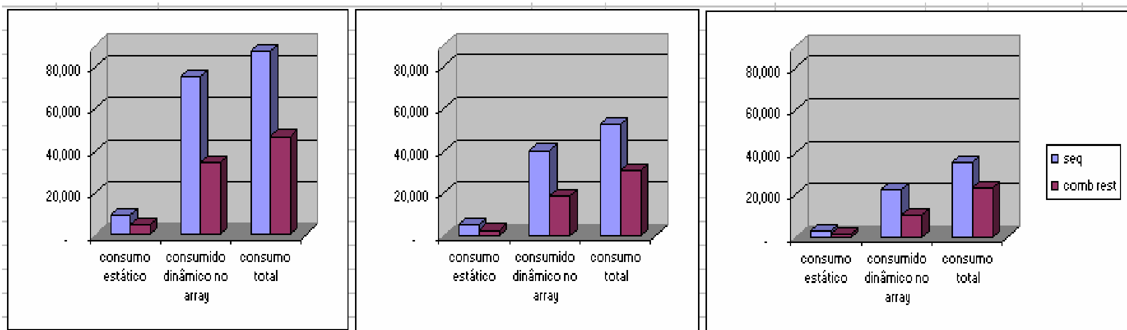
bb 7.a - Sortbubble - Comb. Restr.	3x8	3x4	3x2
Ciclos	7	7	7
acessos à memória	2	2	2
potencia estática consumida	4,820	2,540	1,400
potencia consumida na memória	12,288	12,288	12,288
potencia consumida no array	34,584	18,624	10,644
potencia total consumida	46,872	30,912	22,932

bb 7.b - Sortbubble - Comb. Restr.	3x8	3x4	3x2
Ciclos	7	7	7
acessos à memória	2	2	2
potencia estática consumida	4,820	2,540	1,400
potencia consumida na memória	12,288	12,288	12,288
potencia consumida no array	34,490	18,530	10,550
potencia total consumida	46,778	30,818	22,838

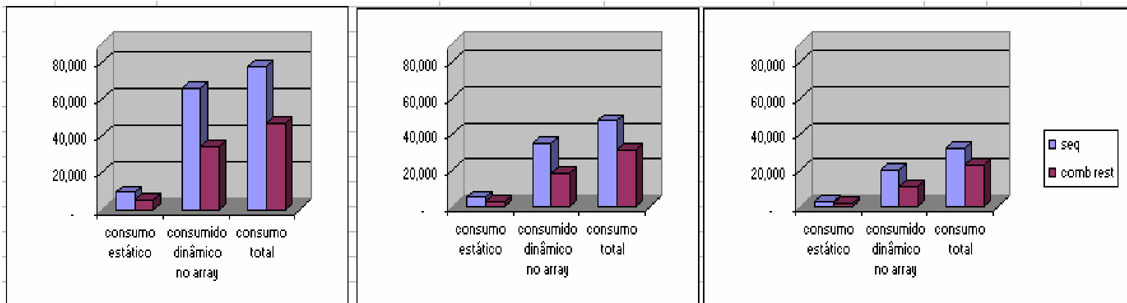
Através dessas tabelas, pode-se obter uma comparação entre a execução dos blocos na variação Sequencial e na variação Combinacional Restrita (figura 5.2).



a) bloco 6



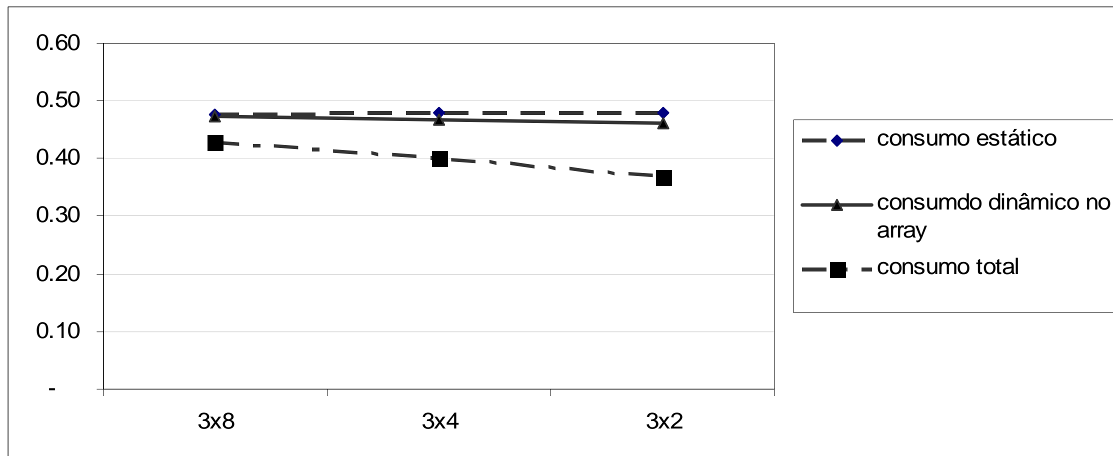
b) bloco 7.a



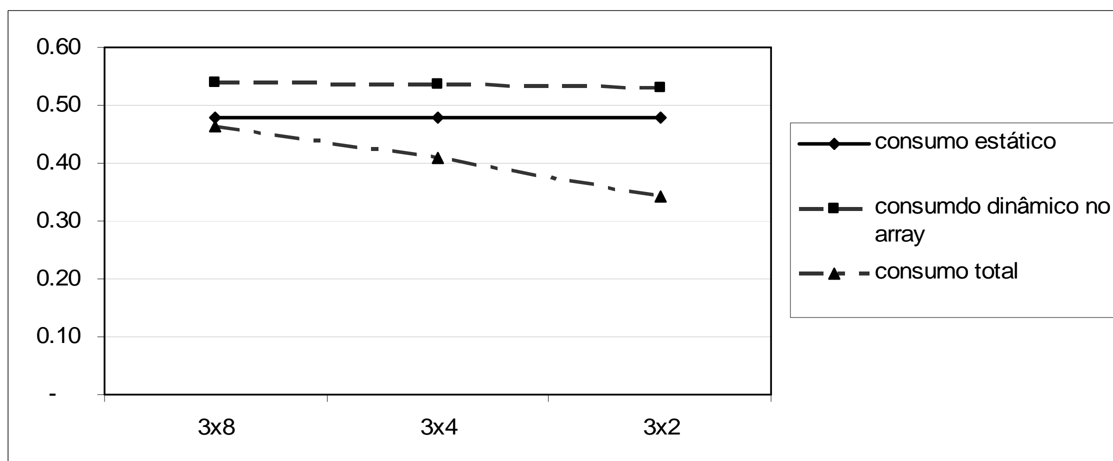
c) bloco 7.b

Figura 5.2: Comparação entre consumo de energia no Javarray Sequencial e Javarray Combinacional Restrito, considerando, em ordem, variações 3x8, 3x4 e 3x2 do array; resultados dados em "capacitâncias de gate por ciclo".

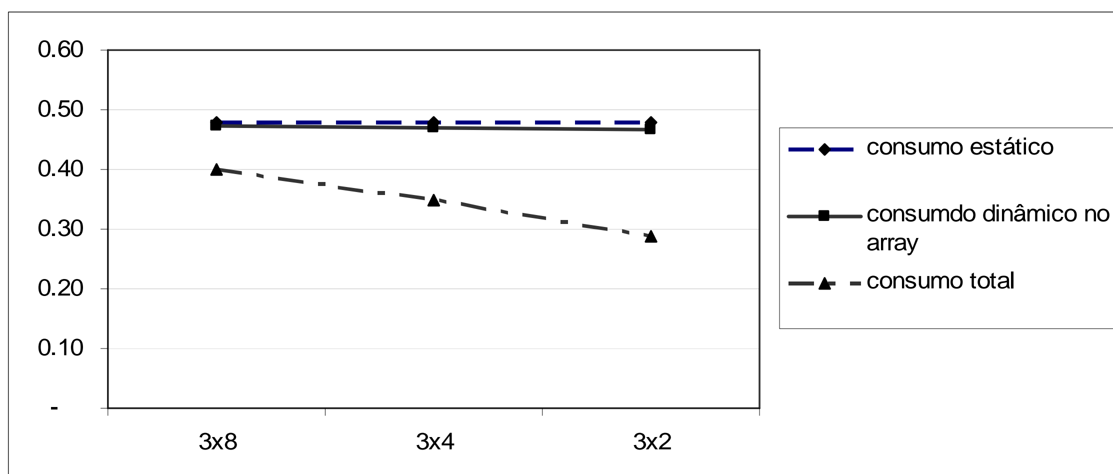
A tabela acima demonstra a economia de energia consumida em cada seção do array, da organização combinacional em relação à organização sequencial. Pode-se perceber que ocorre uma economia total entre aproximadamente 30% e 40%, sendo que essa redução quase chega a ser linear em relação à profundidade adotada para o array – a menos do consumo em memória, que é sempre constante e faz com que a *energia total consumida* não seja de fato linear, como pode ser observado na figura 5.3 – considerando-se o mesmo bloco básico 6, como exemplificação.



a) bb 6



b) bb 7.a



c) bb 7.b

Figura 5.3: Redução de consumo quase linear com a profundidade do array.

Agora, realizando-se a execução dos 3 blocos considerados no FemtoJava Multiciclo, obteve-se o seguinte conjunto de medições (tabela 5.3):

Tabela 5.3: Estatísticas da simulação do FemtoJava Multiciclo.

bb 6 - SortBubble - FemtoJava Multiciclo	
Ciclos	50
acessos à memória (ram)	23
potencia consumida na memória (ram)	94,208
potencia consumida no núcleo	60,198
potencia consumida na rom	3,832
potencia total consumida	158,246
Bb 7a - SortBubble - FemtoJava Multiciclo	
Ciclos	50
acessos à memória (ram)	25
potencia consumida na memória (ram)	102,400
potencia consumida no núcleo	54,903
potencia consumida na rom	3,072
potencia total consumida	160,375
Bb 7b - SortBubble - FemtoJava Multiciclo	
Ciclos	61
acessos à memória (ram)	31
potencia consumida na memória (ram)	126,976
potencia consumida no núcleo	68,282
potencia consumida na rom	3,584
potencia total consumida	198,842

Na tabela 5.3 pode-se perceber algumas medições adicionais às que vinham sendo utilizadas nas análises das execuções no Javarray: aqui, tem-se também a obtenção dos dados de consumo relativos à busca das instruções, que introduzem consumo e acessos à memória rom e algum consumo no próprio núcleo, não existentes nas considerações acerca do Javarray. O consumo de busca de instruções pode ser desconsiderado no Javarray porque um bloco inteiro de instruções pode ser determinado a partir apenas do endereço da primeira instrução do mesmo. Percebe-se também um número elevado de acessos à memória ram no FemtoJava. Algumas instruções realizam mais de um acesso, como o invokestatic. Isso se deve ao fato de que o FemtoJava Multiciclo implementa a pilha da JVM em memória.

Os mesmos blocos, agora simulados na arquitetura FemtoJava Low-Power, apresentam os dados mostrados na tabela 5.4 a seguir.

Tabela 5.4: Estatísticas da simulação do FemtoJava Low-Power.

FemtoJava Low-Power						
	Ram	**rom**	**Core**	**ciclos**	**Ram+rcore**	**total**
Bb 6 total	**40,948**	**8,473**	**183,359**	**81**	**224,307**	**232,780**
Bb 7ª	**77,803**	**15,661**	**322,265**	**143**	**400,068**	**415,729**
Bb 7b	**155,606**	**31,322**	**644,530**	**286**	**800,136**	**831,458**
Bb 7c	**274,357**	**55,456**	**1,150,154**	**510**	**1,424,511**	**1,479,967**
Bb 7 total	**507,766**	**102,439**	**2,116,949**	**939**	**2,624,715**	**2,727,154**

Até aqui considerou-se o consumo de energia dos blocos reconfigurados nas duas versões da arquitetura Javarray e também a execução dos mesmos blocos nas duas versões do FemtoJava. Entretanto, tem-se estimativas de consumo do Javarray apenas para a fase de execução dos blocos, e não se estimou nada ainda a respeito das fases de pré-processamento e pós-processamento. Portanto, para uma comparação mais justa da utilização do Javarray nas duas versões do FemtoJava, esse consumo deve ser também considerado. Esse consumo traduz-se por uma soma das respectivas parcelas de consumo das instruções de pré e pós-processamento nos processadores hospedeiros aos resultados já mostrados do Javarray.

Todas as demais operações-folha, que não são instruções de pré-processamento, são constantes após a primeira carga, e aqui são assumidos como de consumo zero no array, pois o consumo da primeira ocorrência dessas instruções é diluído em relação às várias repetições das instruções reconfiguradas durante a execução da aplicação.

Como todos os blocos de operandos considerados cabem adequadamente em apenas um estágio (dimensão de 3x2 pe's) e, como pela análise mostrada na figura 5.3, a alteração de profundidade do array provoca uma alteração quase que linear no consumo (a menos do consumo em memória, que é constante), reuniu-se apenas as medidas das variações 3x2 numa única tabela comparativa FemtoJava Multiciclo vs. Javarray, já somando as parcelas de pré e pós-processamento necessárias aos blocos executados no Javarray (tabela 5.5).

Tabela 5.5: Comparações entre Javarray e FemtoJava Multiciclo.

Comparações considerando pré e pós-processamento			3x2		3x2	
	FemtoJava Multiciclo	%	Javarray Seq	%	Javarray Comb Rest	%
bb 6 (total)						
consumo rom	3,832	100	0	-	0	-
consumo ram	94,208	100	25,487	27	25,487	27
consumo núcleo	60,198	100	40,594	67	31,292	52
consumo total	**158,246**	**100**	**66,081**	**42**	**56,779**	**36**
bb 7.a						
consumo rom	3,072	100	0	-	0	-
consumo ram	102,400	100	25,487	25	25,487	25
consumo núcleo	54,903	100	43,127	79	31,124	57
consumo total	**160,375**	**100**	**68,614**	**43**	**56,611**	**35**
bb 7.b						
consumo rom	3,584	100	0	-	0	-
consumo ram	126,976	100	24,576	19	24,576	19
consumo núcleo	68,282	100	48,453	71	29,340	43
consumo total	**198,842**	**100**	**73,029**	**37**	**53,916**	**27**
bb 7 (total)						
consumo rom	8,704	100	0	-	0	-
consumo ram	290,816	100	111,503	38	111,503	38
consumo núcleo	158,857	100	127,252	80	96,136	61
consumo total	**458,377**	**100**	**238,755**	**52**	**207,639**	**45**

A tabela acima apresentada reúne as informações recolhidas anteriormente para o FemtoJava Multiciclo e compara as variações 3x2 das organizações Sequencial e Combinacional Restrita do Javarray com relação ao hospedeiro. Cabe a observação de que os dados relativos ao *bb 7 total*, não são meramente a soma dos bb 7.a e bb 7.b; está-se considerado também a parcela bb 7.c como se executado no FemtoJava, pois este faz parte do bloco básico 7, mas não é mapeado para o Javarray por ter sido considerado inapropriado para o mapeamento, como já explicado.

A tabela 5.5 mostra na primeira coluna todos os blocos de operandos considerados para mapeamento no Javarray. Em seguida, na segunda coluna, mostra o consumo da execução desses blocos no FemtoJava Multiciclo original, sem o uso do Javarray, o que representa 100% do consumo original de energia. Depois, mostra-se os mesmos blocos caso se utilize o Javarray para otimizar esses blocos, tanto na sua versão sequencial como na combinacional. Percebe-se que o consumo original de energia, que é de 100%, é reduzido quando se utiliza o Javarray.

A análise de tabela 5.5 mostra que o Javarray é capaz de providenciar economias de até 73% (100% - 27%), no caso do bloco de operandos 7.b, com a utilização da versão combinacional.

Uma observação adicional é de que a estimativa para os casos reduzidos (3x2) do array, acima, estão superdimensionadas, pois a estrutura de memory_access também deveria ser reduzida para um número de entradas menor, relativo a redução do tamanho do array; mas apenas aterrou-se as entradas adicionais. Ou seja, há ainda um consumo estático adicional na estimativa apresentada, embora não tão relevante. Podemos considerar que se há um erro nessa estimativa, é mais provável que o erro esteja *para mais*.

Então, fez-se o mesmo tipo de análise comparativa para o FemtoJava Low-Power (tabela 5.6).

Tabela 5.6: Comparações entre Javarray e FemtoJava Low-Power

Comparações considerando pré e pós-processamento			3x2		3x2	
	FemtoJava Pipeline	%	Javarray Seq	%	Javarray Comb Rest	%
bb 6 (total)		100%				
consumo rom	6,510	100%	-	0%	-	0%
consumo ram	16,384	100%	16,384	100%	16,384	100%
consumo núcleo	25,715	100%	26,303	102%	17,001	66%
consumo total	**48,609**	**100%**	**42,687**	**88%**	**33,385**	**69%**
bb 7.a						
consumo rom	3,080	100%	-	0%	-	0%
consumo ram	16,381	100%	16,384	100%	16,384	100%
consumo núcleo	49,092	100%	28,836	59%	16,833	34%
consumo total	**68,553**	**100%**	**45,220**	**66%**	**33,217**	**48%**
bb 7.b						
consumo rom	3,080	100%	-	0%	-	0%
consumo ram	16,381	100%	16,384	100%	16,384	100%

consumo núcleo	49,092	100%	29,900	61%	20,669	42%
consumo total	**68,553**	**100%**	**46,284**	**68%**	**37,053**	**54%**
bb 7 (total)						
consumo rom	36,855	100%	-	0%	-	0%
consumo ram	7,188	100%	36,861	513%	36,861	513%
consumo núcleo	138,906	100%	62,347	45%	41,113	30%
consumo total	**182,949**	**100%**	**99,208**	**54%**	**77,974**	**43%**

Então, tendo-se a otimização produzida numa execução individual de cada um dos blocos, podemos obter as tabelas 5.7 e 5.8, que estimam a otimização global gerada pelo Javarray quando de uma execução completa do programa SortBubble – que se dá pela soma das parcelas calculadas nas tabelas 5.5 e 5.6 (respectivamente para o FemtoJava Multiciclo e para o FemtoJava Low-Power), multiplicadas pela representatividade dos blocos no *profile* do programa, que já foram vistas no capítulo 3.

Tabela 5.7: Economia de energia com o FemtoJava Multiciclo.

FemtoJava Multiciclo	representatividade	Econ. Seq	Total	Econ. Comb Rest	Total
bb 6	36.03%	58%	21%	64%	23%
bb 7	43.06%	48%	21%	55%	24%
economia total sortbubble			42%		47%

Tabela 5.8: Economia de energia com o FemtoJava Low-Power

FemtoJava Low-Power	representatividade	Econ. Seq	Total	Econ. Comb Rest	Total
bb 6	36.03%	12%	4%	31%	11%
bb 7	43.06%	46%	20%	57%	25%
economia total sortbubble			24%		36%

Como visto pela tabela 5.7, para estimar o total de economia produzido pelo Javarray para o algoritmo SortBubble, devemos lembrar que o bloco básico 6 representa 36.03% da execução do profile do SortBubble, enquanto que o bloco básico 7 representa 43.06%. Assim, multiplicando a economia relativa encontrada em cada um dos blocos pelo total da execução do profile, temos uma estimativa da economia global de energia produzida por cada um dos blocos, nas duas organizações Javarray consideradas. A soma das contribuições individuais de cada bloco básico otimizado nos dá o total de otimização produzida pelo Javarray. Com a utilização do Javarray Sequencial em relação à execução completa no FemtoJava Multiciclo temos uma economia total de energia de 42%, e uma economia de 47% com a utilização do Javarray Combinacional Restrito, em relação ao uso do FemtoJava Multiciclo sem o Javarray acoplado.

As mesmas considerações realizadas para a tabela 5.7 são válidas para a tabela 5.8. Pela tabela 5.8 pode-se perceber que a economia total de energia produzida pelo

Javarray acoplado ao FemtoJava Low-Power é de 24% para a versão sequencial e de 36% para a versão combinacional, relativo a execução do FemtoJava Low-Power sem o uso do Javarray.

A análise de desempenho também pode ser desenvolvida. Considerando-se o tempo de um ciclo relativo à execução de uma operação na ULA do FemtoJava como semelhante ao tempo de execução de uma operação em um *pe* do Javarray, e tendo-se as medidas em número de ciclos necessários para completar cada um dos blocos (vistos nas tabelas de 5.1 a 5.4), e ainda considerando-se o número de repetições dos blocos como visto no capítulo 3, pôde-se determinar a tabela 5.9, que mostra que a execução do SortBubble com o mesmo conjunto de dados, produziria os resultados com uma redução entre 32.85% e 58.55% no número de ciclos, dependendo da arquitetura FemtoJava hospedeira considerada, se o Javarray for utilizado.

Com relação à análise de performance, não se está considerando o impacto dos registradores no caminho do fluxo de dados. A versão combinacional do Javarray é obviamente mais rápida, mas, devido a restrições de modelagem, está-se considerando insignificante o atraso dos registradores para a computação de um *pe,* e está-se apresentando os resultados em ciclos de relógio, cada um relativo a execução de um *pe* do caminho crítico do grafo de dependência de instruções. Então, os resultados apresentados na tabela 5.9 são os mesmos para as versões combinacional e sequencial do Javarray, apesar de a versão combinacional executar de fato em um único grande ciclo de relógio.

Tabela 5.9: Estimativas de otimização de desempenho com o Javarray.

SortBubble - Estimativas de otimização de desempenho					
	ciclos FemtoJava Multiciclo	% profile	ciclos Javarray	Economia bloco	economia profile
bb 6	50	36.03%	18	64.00%	**23.06%**
bb 7	140	43.06%	62	55.71%	**23.99%**
soma		79.09%			**47.05%**
	ciclos FemtoJava Low-Power	% profile	ciclos Javarray	Economia bloco	economia profile
bb 6	9	36.03%	9	0.00%	**0.00%**
bb 7	106	43.06%	38	64.15%	**27.62%**
soma		79.09%			**27.62%**

Na tabela 5.9 mostrada acima há duas comparações em que pode-se ver a duração de execução, em ciclos, de cada bloco básico em cada uma das variações consideradas do FemtoJava. A representatividade desses blocos na execução total do profile é a mesma já vista anteriormente, e soma 79.09% da execução do Sortbubble. Ao lado, a duração de execução dos mesmos blocos se executados no Javarray, considerando-se também os ciclos necessários para pré e pós-processamento na arquitetura hospedeira em questão. A multiplicação da representatividade do bloco no *profile* pela representação dos ciclos de execução Javarray em relação aos ciclos originais FemtoJava, leva à próxima coluna. Por fim, pode-se estimar a economia produzida pela execução dos blocos no Javarray pelo percentual de execução reduzido com a utilização do Javarray subtraído da representatividade dos blocos no profile. A linha de “soma” refere-se à economia total, em ciclos, produzida pela utilização do Javarray na execução do Sortbubble.

Ou seja, cada um dos blocos executados no Javarray produz uma economia relativa à diferença do número necessário de ciclos para execução do bloco no FemtoJava e o número de ciclos necessários no Javarray, multiplicada pela representatividade desse bloco no profile.

Vale a observação de que os ciclos de execução no Javarray poderiam baixar: por restritções de modelagem, cada acesso à memória está custando 2 ciclos, mas poderia custar apenas 1.

Por fim, possuindo-se as estimativas de energia e as estimativas de performance, produz-se a tabela 5.10, que mostra a potência consumida pelo FemtoJava quando utilizando o Javarray acoplado em relação ao seu consumo de potência orginal sem o uso do Javarray.

Tabela 5.10: Potência consumida com Javarray acoplado.

Cálculos de Potência Consumida – em capacitâncias de gate						
	FemtoJava Multiciclo		Javarray Seq	% var	Javarray Comb Rest	% var
bb 6	3,165		3,671	116%	3,154	100%
bb 7	3,274		3,851	118%	3,349	102%
	FemtoJava Low-Power		Javarray Seq	% var	Javarray Comb Rest	% var
bb 6	5,401		4,743	88%	3,709	69%
bb 7	1,726		2,611	151%	2,052	119%

Pela tabela 5.10, pode-se ver que a potencia consumida fica muito próxima à consumida pelo FemtoJava Multiciclo, quanto o Javarray está acoplado a ele, apesar dos ganhos significativos de energia. Em alguns casos no acoplamento com o FemtoJava Low-Power a potencia chega até mesmo a ser menor que a do processador hospedeiro sem o auxílio do Javarray, mesmo com simultâneo aumento de desempenho e economia de energia. Isso se deve ao fato de o Javarray propiciar economias de energia maiores do que o aceleramento da performance, na maioria dos casos.

As análises expostas nesta seção acerca do SortBubble também foram realizadas para os blocos selecionados das aplicações IMDCT e Crane 16bits. A seguir segue um resumo dos resultados obtidos para todas as aplicações consideradas nos experimentos.

5.2 Performance

Conforme já descrito na seção anterior, os resultados de performance apresentados são dados em ciclos de relógio relativos à execução de um *pe*. Assim, deve-se considerar que a versão combinacional executará de fato num grande ciclo relativo à soma de todas as execuções dos *pe's.* Além disso, não se está considerando o impacto dos registradores, por restrições de modelagem e simulação, e portanto, está-se

considerando os tempos de execução das versões sequencial e combinacional do Javarray como semelhantes.

A tabela 5.11 a seguir compila os resultados de performance obtidos para as três aplicações embarcadas em análise neste trabalho.

Tabela 5.11: Economia de Ciclos produzida pelo Javarray.

Economia de Ciclos		Javarray acoplado a			
		Multiciclo		**Low-Power**	
blocos	% profile	economia no bloco	economia global	economia no bloco	economia global
12 imdct	81%	67%	54%	57%	46%
14 imdct	13%	64%	8%	51%	7%
Imdct	**100%**		**62%**		**53%**
6 sortbubble	36%	64%	23%	0%	0%
7 sortbubble	43%	56%	24%	64%	28%
sortbubble	**100%**		**47%**		**28%**
8 crane	10%	65%	6%	0%	0%
31 crane	3%	25%	1%	8%	0%
35 crane	3%	60%	2%	38%	1%
39 crane	7%	48%	4%	0%	0%
44 crane	4%	31%	1%	55%	2%
51 crane	3%	65%	2%	55%	2%
62 crane	4%	72%	3%	61%	2%
crane 16 bits	**100%**		**18%**		**8%**

Na tabela 5.11 acima, pode-se perceber o potencial de otimização da performance dos blocos básicos proporcionado pelo Javarray, que vão de 31% a 72%, quando acoplado ao FemtoJava Multiciclo – produzindo aumento de performance, respectivamente, de 1.44 e 3.57 vezes. No caso do acoplamento com o FemtoJava Low-Power há menos margem para otimizações, pois essa arquitetura já é mais focada em desempenho, constituíndo-se numa versão com *pipeline*. Mesmo assim, produz-se economias de ciclos dos blocos reconfigurados que chegam até 64%.

Com as otimizações produzidas nos blocos individuais, e considerando-se a representação desses blocos nos *profiles* das aplicações, produziu-se economias globais que vão de 8% a 62%, dependendo da aplicação, da versão Javarray e da versão FemtoJava utilizados, o que representa ganhos de performance entre 1.08 e 2.63 vezes. Observe-se que no caso do Crane, os blocos reconfigurados possuem uma representatividade no *profile* muito menor do que os blocos selecionados para as outras aplicações – ou seja, as economias são menores que nas demais aplicações, mas para um potencial de otimização também muito menor.

5.3 Energia

A tabela 5.12 compila as estatísticas de economia de energia levantadas para todas as aplicações consideradas. Considera-se ambas as versões do Javarray acopladas a cada uma das versões do FemtoJava.

Tabela 5.12: Economia de energia produzida pelo Javarray.

		Javarray Combinacional		Javarray Sequencial	
Economia de Energia		**Multiciclo**	**Low-Power**	**Multiciclo**	**Low-Power**
blocos	% profile	economia no bloco	economia no bloco	economia no bloco	economia no bloco
12 imdct	81%	68%	43%	57%	10%
14 imdct	13%	70%	47%	58%	12%
Imdct	**100%**	**64%**	**41%**	**54%**	**10%**
6 sortbubble	36%	64%	31%	58%	12%
7 sortbubble	43%	55%	57%	48%	46%
sortbubble	**100%**	**47%**	**36%**	**42%**	**24%**
8 crane	10%	63%	53%	60%	44%
31 crane	3%	51%	14%	42%	0%
35 crane	3%	54%	33%	51%	22%
39 crane	7%	59%	58%	56%	49%
44 crane	4%	69%	43%	65%	29%
51 crane	3%	69%	53%	65%	38%
62 crane	4%	76%	61%	73%	48%
crane 16 bits	**100%**	**22%**	**17%**	**20%**	**12%**

Na tabela 5.12 acima, pode-se perceber economias globais de energia que vão de 17% a 64% quando se utiliza o Javarray Combinacional, dependendo da aplicação e da versão acoplada do FemtoJava. Para o Javarray Sequencial, produz-se economias que vão de 10% a 54%. As mesmas observações sobre a representatividade dos blocos reconfigurados do Crane, realizadas na seção anterior, são válidas também aqui.

Uma análise qualitativa dos dados obtidos indica que a redução do consumo de energia obtida com a utilização do Javarray em acoplamento ao FemtoJava Multiciclo se deve em muito ao fato de a versão multiciclo ser a menos eficiente dentre as versões do FemtoJava em termos de consumo – apropriadamente em detrimento da área. Para isso, o FemtoJava Multiciclo implementa a pilha quase toda em memória, e consome muito, portanto. Como o Javarray elimina a necessidade de utilização da pilha para os blocos configurados, economiza-se muito nesse ponto; além de poupar-se ciclos de execução com a eliminação das dependências de dados, podendo-se até mesmo chegar a eliminar a necessidade de algumas instruções.

Já com relação ao FemtoJava Low-Power, o ganho produzido pelo Javarray se dá em boa parte no fato de que o Javarray permite o acesso simultâneo a 3 entradas, necessárias à execução da instrução iastore. No FemtoJava Pipeline gasta-se muitos ciclos e acessos à pilha para realizar tal operação , que no Javarray pode ser executada em apenas 2 ciclos, sendo um para execução e 1 para escrita do valor na memória.

Decorrente da possibilidade de se armazenar a configuração de blocos básicos, e baseado no conceito de que os blocos básicos não contém saltos internos, pode-se economizar o consumo dos ciclos de busca e decodificação de todo o conjunto de instruções pela simples identificação do endereço da primeira instrução do bloco. Isso elimina consumo em rom e também consumo no núcleo do microcontrolador hospedeiro.

Assim, ao se identificar a busca da primeira instrução referente a um bloco básico já previamente configurado para o Javarray, basta executar essa configuração e saltar a busca da próxima instrução ao endereço posterior ao da última instrução do bloco básico, apropriadamente.

Uma observação adicional é a de que a integração do Javarray ao FemtoJava requer lógica adicional e possivelmente memória para armazenar as configurações. Esse mecanismo, que também consome energia, não foi considerado na análise deste trabalho, pois ainda é indefinido. Considerou-se aqui que os arrays estão mapeados estaticamente, ocupando área, portanto. Ainda que tais mecanismos de tradução dinâmica não tenham sido implementados, imagina-se que as consideráveis reduções de consumo obtidas sejam suficientes para deixar margem propícia para a introdução dos mesmos, e ainda permitindo economia de energia com o aumento de desempenho.

5.4 Potência

Para cada bloco básico reconfigurado também foram obtidos os consumos de potência introduzidos pelas duas versões do Javarray. Da mesma forma que nas estimativas de energia, a tabela 5.13 compara o consumo de potência alcançado pelo acoplamento do FemtoJava com o Javarray contra o consumo original do FemtoJava sem o Javarray. Logo, resultados próximos a 100% representam o consumo normal de potência da respectiva versão do FemtoJava sem o uso da unidade reconfigurável. Resultados abaixo disso apontam para reduções do consumo de potência produzidos pelo uso do Javarray.

Apesar de um aumento de potência ser esperado devido ao aumento de performance, pode-se perceber que esse aumento é pequeno. Em muitos casos – principalmente aqueles onde o Javarray é acoplado à versão Low-Power do FemtoJava – uma simultânea redução na potência consumida foi obtida juntamente com economia de energia e aumento de performance. Isso não é absurdo e ocorre essencialmente quando a economia de energia obtida é proporcionalmente maior do que os ganhos em performance. Esse fato pode ser observado principalmente com o FemtoJava Low-Power porque ele não permite tanta margem para aceleração de desempenho quanto a versão Multiciclo.

Tabela 5.13: Potência consumida com introdução do Javarray.

Potência	**Javarray Combinacional**		**Javarray Sequencial**	
blocos	Multiciclo	Low-Power	Multiciclo	Low-Power
12 imdct	**97%**	**45%**	130%	209%
14 imdct	**85%**	**41%**	119%	**68%**
6 sortbubble	100%	**69%**	116%	**88%**
7 sortbubble	102%	119%	118%	151%
8 crane	184%	**47%**	207%	**56%**
31 crane	**66%**	**94%**	**77%**	125%
35 crane	113%	106%	121%	125%
39 crane	**78%**	**42%**	**85%**	**51%**
44 crane	**44%**	126%	**50%**	158%
51 crane	**89%**	105%	101%	137%
62 crane	**85%**	**98%**	**98%**	133%

Perceba-se que os resultados de consumo de potência da versão sequencial são próximos aos da versão combinacional. Para melhor visualisação, um gráfico (figura 5.4) compara essas duas versões do Javarray, mostrando a pequena degradação da versão sequencial sobre a combinacional, quando acoplado às duas versões do FemtoJava.

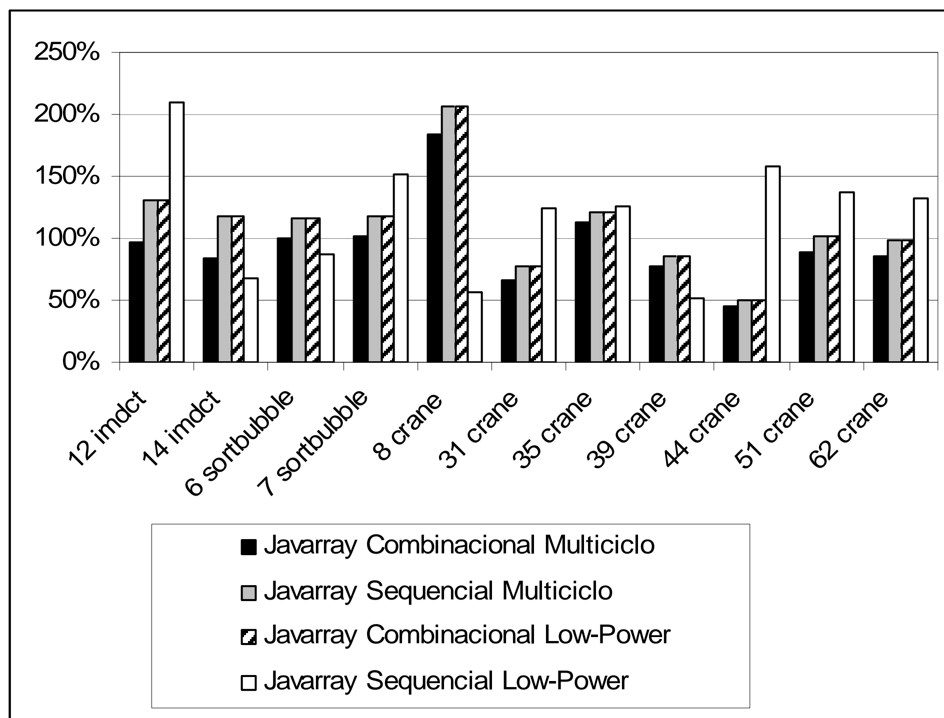


Figura 5.4: Comparação gráfica da potência consumida pelo Javarray entre as diferentes possibilidades de versões FemtoJava hospedeiras.

5.5 Estimativas gerais de Área

A versão do Javarray Sequencial foi implementada em VHDL. Os dados obtidos para o Quartus II Web Editon 2.2 mostram uma área aproximada de 25.000 elementos lógicos (le's) – dependendo de alguns parâmetros de configuração. A área obtida é grande se comparada à utilizada pelo FemtoJava, abaixo de 3.000 le's, principalmente pela grande replicação de ALUs existente no array. Entretanto, esse valor é mais significativo para demonstrar a complexidade do projeto do que como efeito do tamanho do mesmo, já que FPGAs não são adequadas para o mapeamento de arquiteturas de granularidade grossa.

A área de cada PE possui na sua versão com multiplicador um total de 614 le's. O impacto da replicação de multiplicadores em todos os PEs é grande, já que um PE sem multiplicador tem sua área reduzida para 273 le's. Considerando-se um arranjo de (3x8) + 1 PEs, e considerando a área utilizada por cada PE, compõe-se uma área de 25 x 614 le's, o que dá um total de 15.350 le's. Ou seja, aproximadamente 10.000 le's foram utilizados para interconexões e demais estruturas de registradores e acesso à memória, correspondente a 40% da área total.

Realizando o mesmo cálculo agora para o array parametrizado como 3x4, obtém-se aproximadamente 12.000 le's de área total e 13 x 614 le's utilizados nos PEs, num total de 7.982 le's. A proporção ainda se mantém, sendo aproximadamente 36% da área utilizada nas demais estruturas, demonstrando a alta regularidade estrutural e baixa complexidade do projeto.

A escalabilidade do projeto é quase linear para aumentos na profundidade do array. Pouca área adicional foi ocupada pelas interconexões e pelos recursos adjacentes, devido principalmente à alta localidade dos mesmos, de acordo com sua definição arquitetural. A redução de área, buscando economizar em multiplicadores nas ALUs e reduzir a área utilizada em interconexões são fortes possibilidades de exploração futura.

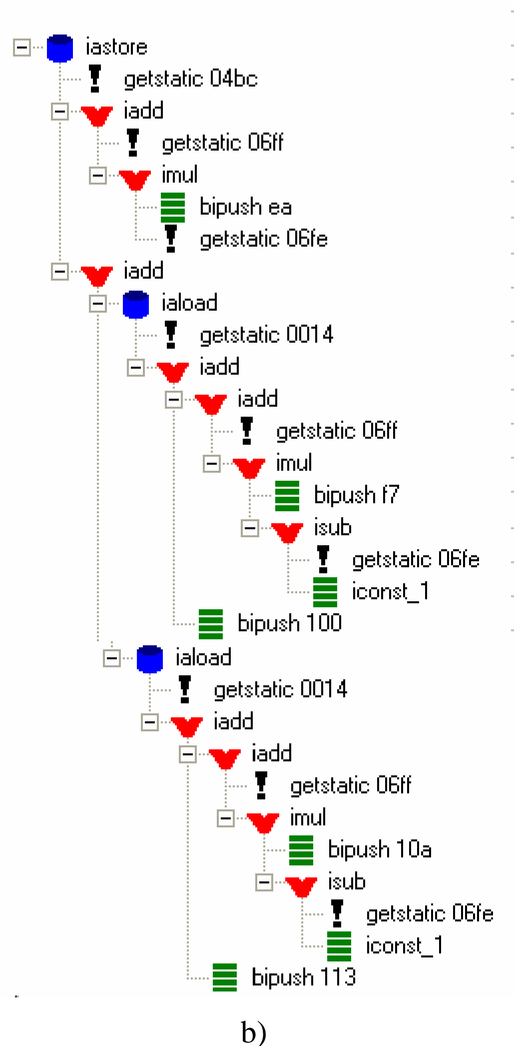
As considerações acima referem-se à versão sequencial – que foi a implementada em VHDL.

5.6 Execução baseada em Stream

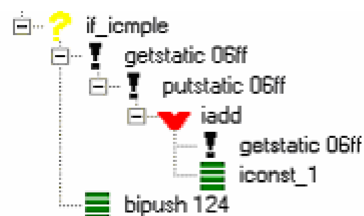
A versão combinacional do Javarray é a que apresenta os melhores desempenhos em energia e performance. Entretanto, sendo combinacional, todos os *pe's* computados ficam impossibilitados de serem reaproveitados até o término da computação da instrução reconfigurada. A existência de registradores nas entradas dos pe's, na versão sequencial do Javarray, permite um melhor aproveitamento do array, uma vez que os pe's podem ser reaproveitados logo após sua computação tenha sido efetuada. Assim, pode-se programar um pipeline de instruções reconfiguradas, permitindo a exploração de computação baseada em stream, através da aplicação da já conhecida técnica de *software pipelining* ao Javarray.

Nesta seção será exemplificado, com o uso do bloco básico 12 da aplicação IMDCT, a possibilidade de exploração de execução *stream* no Javarray. É importante salientar que alguns mecanismos, como os registradores de passagem e multicontextos nos *pe's*, necessários para a utilização de pipeline no Javarray, não foram implementados. Esta seção serve apenas como demonstrativa das potencialidades futuras da versão sequencial do Javarray.

As árvores de dependência dos blocos de operandos 12.a e 12.b do bloco básico 12 do IMDCT são apresentados na figura 5.5, a seguir.



b)



a)

Figura 5.5: a) Bloco de operandos 12.a do IMDCT; b) bloco de operandos 12.b.

As instruções dos blocos da figura 5.5 executam em paralelo, a partir das folhas, até atingir as instruções-raíz, conforme as tabelas 5.14 e 5.15 abaixo. As distribuições em “colunas” servem apenas para facilitar a visualização do escalonamento das instruções, e não possuem relação alguma com as colunas de *pe’s* do Javarray.

Tabela 5.14: Execução paralela das instruções do bloco de operados 12.a.

	coluna1	coluna2	coluna 3	coluna 4	coluna 5	Coluna 6	ordem de execução:
linha1	-	-	getstatic	iconst	getstatic	iconst	ciclo1
linha2	-	-	bipush	isub	bipush	isub	ciclo2
linha3	-	-	getstatic	imul	getstatic	imul	ciclo3
linha4	-	-	bipush	iadd	bipush	iadd	ciclo4
linha5	bipush	getstatic	getstatic	iadd	getstatic	iadd	ciclo5
linha6	gestatic	imul	-	iaload	-	iaload	ciclo6
linha7	getstatic	iadd	-	-	iadd	-	ciclo7
linha8	-	iastore	-	-	-	-	ciclo8

Tabela 5.15: Execução paralela das instruções do bloco de operandos 12.b.

	coluna1	coluna2	ordem de execução:
linha1	getstatic	iconst_1	ciclo1
linha2	iadd	-	ciclo2
linha3	putstatic	-	ciclo3
linha4	getstatic	bipush	ciclo4
linha5	-	if_icmple	ciclo5
linha6	-	-	ciclo6
linha7	-	-	ciclo7
linha8	-	-	ciclo8

Note-se que ambas as distribuições de instruções 12.a e 12.b apresentadas na tabela 5.14 podem executar em paralelo, já que blocos de operandos são independentes entre si. No entanto, há uma instrução no bloco 12.b que altera o valor do endereço 0x06ff, e esse endereço é acessado por um *getstatic* no bloco 12.a – causando uma antidependência. Assim, o *putstatic* só pode ser executado depois do *getstatic*. Abaixo, a Figura 5.6 demonstra uma configuração com a execução do putstatic atrasada, permitindo a execução dos blocos em paralelo.

bloco 12.b			**bloco 12.a**						
getstatic	iconst_1		-	-	getstatic	iconst	getstatic	iconst	iconst
add	bipush		-	-	bipush	isub	bipush	isub	Isub
	If_icmple		-	-	**getstatic**	imul	**getstatic**	imul	Imul
-			-		bipush	iadd	bipush	iadd	Iadd
-	-		Bipush	**getstatic**	getstatic	iadd	getstatic	iadd	Iadd
putstatic	-		Gestatic	imul	-	iaload	-	iaload	iaload
-	-		getstatic	iadd	-	-	iadd	-	-
-	-		-	iastore	-	-	-	-	-

Figura 5.6: Execução dos blocos de operando 12.a e 12.b em paralelo.

Perceba-se em negrito os *getstatics* que acessam o endereço 0x06ff. A seta pontilhada indica que o *putstatic* está sendo executado depois do último *getstatic* que utiliza o endereço 0x06ff.

Note-se que o *putstatic* pode ser suprimido juntamente com o *getstatic*, da árvore, desde que se guarde o valor calculado em *iadd* para posterior execução do *putstatic*. O *getstatic* pode ser suprimido pois ele apenas irá retornar o valor calculado em *iadd*, que pode ser *bypassado* diretamente para o *if_icmple*, como foi feito acima. No entanto, o *putstatic* ainda deve ser executado ao final da execução do bloco básico 12, pois algum outro bloco do resto da aplicação ainda pode tentar acessar o mesmo endereço, posteriormente.

Note-se que as instruções *iaload* e *iastore* levam 2 ciclos para executar. No primeiro ciclo, ambas as instruções calculam uma soma de suas entradas. No segundo ciclo, o *iaload* busca um dado na memória calculada, e o *iastore* escreve um dado na memória.

Agora, imagine que a ordem de execução seja: primeiro executa-se o *iastore* e depois o *iaload*, que por sua vez referencia o mesmo endereço atualizado por iastore. Como ambas as instruções executam em 2 ciclos, é possível que a execução do iaload inicie antes do iastore, desde que se realize um *forward* do dado de *iastore* para *iaload*. Nesse caso, a leitura em memória não é realizada pelo *iaload*, pois já se tem o dado vindo de iastore, como mostrado na figura 5.7 abaixo.

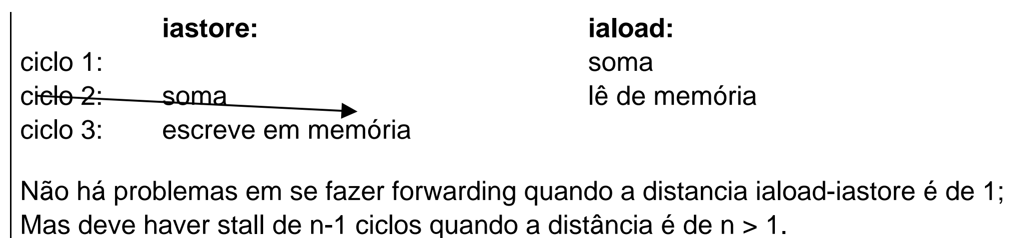


Figura 5.7: Utilização de *forward* em *iastore* para execução antecipada de *iaload*.

Deve-se observar que nem sempre o endereço acessado por iastore é o mesmo que será acessado por iaload. Apenas mostra-se que, caso seja, é possível realizar um forward, permitindo que a execução do iaload inicie mesmo antes do iastore iniciar sua execução.

Dinamicamente, deve-se fazer uma comparação dos endereços calculados por ambas as instruções: se o valor calculado por iaload for igual ao calculado por iastore, faz-se o forward; senão, as instruções são independentes, e procede-se normalmente com as escritas e leituras necessárias.

Esse bloco cujos grafos foram apresentados é um bloco de um *loop*. Isso é facilmente verificável pelo formato do bloco de operandos 12.b, que incrementa uma variável. Ou seja, o bloco 12.a irá executar várias vezes em seguida, sem que outro bloco além dos 12.a e 12.b seja executado. Dessa forma, garantidamente as dependências entre *putstatic-getstatic* e *iastore-iaload* são restritas a esses blocos

enquanto eles estiverem executando. O *putstatic* ainda deve ser executado para garantir dependencias posteriores, como já foi observado.

Sendo assim, é possível fazer-se um *pipeline* desses blocos reconfiguráveis. Imaginando que ambos os blocos de operandos possam ser executados em paralelo, como foi feito acima, tem-se o diagrama temporal apresentado na figura 5.8.

ciclo	Bloco 12.b		Bloco 12.a						Bloco 12.b		Bloco 12.a					
1	ge	ic			ge	ic	ge	ic								
2	ia	bi			bi	es	bi	is								
3		if			ge	im	ge	im	ge	ic			ge	ic	ge	ic
4					bi	ia	bi	ia	ia	bi			bi	es	bi	is
5			bi	ge	ge	ia	ge	ia		if			ge	im	ge	im
6	pu		ge	im		ial		ial					bi	ia	bi	ia
7			ge	ia			ia				bi	ge	ge	ia	ge	ia
8				ias					pu		ge	im		ial		ial
9											ge	ia			ia	
10												ias				

Figura 5.8: Diagrama temporal do pipeline de instruções reconfiguráveis do bloco básico 12 do IMDCT.

Na figura 5.8, há setas que indicam o *forward* dos dados da instrução *iastore* para a saída das instruções *ialoads*, no ciclo 7, caso seja necessário. Perceba-se que houve um *stall* de 2 ciclos para permitir alinhamento necessário entre *iastore* e *ialoads* para o *forward* entre as duas execuções dos blocos paralelos: o segundo bloco incia sua execução no ciclo 3.

As setas indicam *forwards* de dados para os *getstatics* posteriores, pois o *putstatic* ainda não terá sido executado quando os valores dos *getstatics* forem necessários – ou seja, quando há uma antidependência entre o *getstatic* e o *putstatic*, sendo que o valor de *iadd* deve ser guardado para que o putstatic seja executado posteriormente.

Uma observação adicional é que, de fato, sendo possível o *forward* representado pelas linhas inteiras, a execução do *putstatic* só é necessária na última interação - para possíveis dependencias com blocos posteriores ao bloco 12. Todos os *getstatics* que recebem dados de *forward* não precisam ser executados, economizando processamento. A grande vantagem do *pipeline* de instruções reconfiguradas é que não se faz mais superposição de instruções simples, e sim superposição de grafos inteiros de instruções! Dessa forma, o IPC efetivo do *pipeline*, neste caso, fica em 18.50, muito além do que o original de 2.84, sem *pipeline*, caso apenas se executasse os dois blocos intercaladamente. Além disso, o número de instruções executadas *de fato* é bem menor, já que se utiliza muitos *forwards* de dados entre instruções. A economia de instruções executadas pode ser ainda maior, caso se considere ramos constantes das árvores que não precisariam ser também executados após o primeiro processamento.

Para esse *pipeline* funcionar, configura-se os pe's apropriadamente e a cada 2 ciclos dispara-se a execução da configuração com novos dados. Registradores de passagem devem guardar valores intermediários a serem utilizados para *forward*.

6 CONCLUSÕES

Neste trabalho, apresentou-se o desenvolvimento de uma arquitetura reconfigurável de granularidade grossa destinada ao aumento de performance e economia de energia de aplicações embarcadas baseadas em Java. A técnica adotada acelera apenas os blocos básicos mais representativos das aplicações, selecionados por *profile*.

Selecionou-se um conjunto de 3 aplicações embarcadas Java já possuídas pelo grupo de pesquisas do LSE, tendo cada uma dessas aplicações características distintas, na busca de uma maior representação quanto tipos de aplicações consideradas nas análises posteriores.

A partir de uma análise estática do *profile* uma aplicação Java, blocos básicos representativos da maior parte dos ciclos de execução gastos no processamento do processador hospedeiro, e que possuam um número razoável de instruções para justificar sua reconfiguração, são selecionados para terem seus blocos de operandos transformados numa instrução reconfigurável Javarray.

Foi mostrado que por a JVM ser baseada em uma máquina de pilha, os grafos de dependência de instruções das aplicações Java possuem algumas restrições práticas que os fazem similares a árvores binárias, o que permite o projeto de uma arquitetura reconfigurável na forma de array, com alta localidade de recursos e interconexões, tornando o projeto altamente escalável, facilitando no gerenciamento da complexidade de projeto.

Além disso, a reconfiguração de blocos básicos na arquitetura Javarray permite a redução no número de instruções executadas, a exploração do ILP intrínseco existente nas aplicações Java, uma redução no número de acessos à pilha e à memória, uma menor necessidade de busca e decodificação de instruções e menor número de interações no *datapath* do processador hospedeiro, contribuíndo para a redução do consumo de energia e aceleração de performance.

Dessa forma, para o conjunto de aplicações consideradas, conseguiu-se alcançar economias globais de energia entre 10% e 64%, dependendo do processador hospedeiro e da aplicação em questão. Ao mesmo tempo, conseguiu-se economias de 8% a 62% no número de ciclos necessários para executar essas aplicações, o que representa acelerações de desempenho de até 2.6 vezes. Mostrou-se também que mesmo com ganhos em performance e economia de energia, pode-se obter simultânea redução na potência consumida; e que mesmo onde há aumento de consumo de potência, o consumo introduzido pelo Javarray é pequeno, na maioria dos casos. Mesmo para o Crane 16bits, que é uma aplicação voltada ao controle, bons resultados em performance

e energia foram alcançados, indicando a possibilidade de êxito do uso das arquiteturas reconfiguráveis em aplicações de propósitos mais gerais.

Mostrou-se também a aplicabilidade de técnicas de software pipelining, caso se utilize a versão sequencial do Javarray com alguns recursos adicionais, como registradores de passagem. Para trabalhos futuros, percebeu-se diversos pontos de otimização na arquitetura Javarray, como a possibilidade de introdução de restrições nas interconexões, para poupar área, e a possibilidade de redução no número de registradores, por exemplo. Outras possibilidades arquiteturas também foram percebidas, como a utilização de multicontextos para aumentar a utilização do array e a possibilidade de uso de uma *trace cache* para executar blocos reconfigurados maiores, através de especulação.

REFERÊNCIAS

ACHUTHARAMAN, R. et al. Exploiting Java-ILP on a Simultaneous Multi-Trace instruction Issue (SMTI) Processor. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM , IPDPS, 2003. **Proceedings…** Los Alamitos: IEEE Computer Society, 2003.

ADÁRIO, A. M. S. **Arquiteturas Reconfiguráveis.** 1997. 47f. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

ADÁRIO, A. M. S.; BOSCHETTI, M. R.; BAMPI, S. Extending Sequencing Graphs for Reconfigurable Applications Modeling. In: UFRGS MICROELECTRONICS SEMINAR, SIM, 16., 2001, Santa Maria. **Proceedings…** Porto Alegre: Instituto de Informática da UFRGS, 2001.

BARROSO, L. A. et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE,27., 2000. **Proceedings…** New York: ACM Press, 2000. p. 282-293.

BECK FILHO, A. C. S.; MATTOS, J. C. B.; WAGNER, F.; CARRO, L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 16., SBCCI, 2003, São Paulo, Brazil. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2003. p. 349-354.

BECK FILHO, A. C. S.; CARRO, L. A VLIW Low Power Java Processor for Embedded Applications. In: INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, 12., IFIP, 2003. **Proceedings…** [S.l.: s.n.], 2003b.

BECK FILHO, A. C. S.; GOMES, V. F.; CARRO, L. Exploiting Java Through Binary Translation for Low Power Embedded Reconfigurable Systems. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 18., 2005, Florianópolis. **Proceedings…** Los Alamitos: IEEE, 2005. p. 92-97.

BECKER, J. et al. Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems. In: FCCM, 2000, Napa, CA, USA. **Proceedings…** Los Alamitos: IEEE, 2000.

BECKER, J.; VORBACH, M. Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC), In:

ANNUAL SYMPOSIUM ON VLSI , ISVLSI, 2003. **Proceedings…** Los Alamitos: IEEE Computer Society, 2003.

BERTIN, P.; TOUATI, H. PAM Programming Environments: Practice and experience. In: WORKSHOP ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1994, Napa, CA. **Proceedings…** Los Alamitos: IEEE Computer Society, 1994. p. 133-138,

BONDALAPATI, K.; PRASANNA, V.. Reconfigurable Computing Systems. **Proceedings of the IEEE**, Piscataway, v. 90, n. 7, p. 1201-1217, July 2002.

BUELL, D.; ARNOLD, J.; ATHANAS, P. Athanas. SPLASH2: FPGAs in a Custom Computing Machine. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 1996.

CADAMBI, S.; WEENER, J.; GOLDSTEIN, S.; SCHMIT, H.; THOMAS, D. Managing Pipeline-reconfigurable FPGAs. In: INTERNATIONAL SYMPOSIUM ON FPGA, 1998. **Proceedings…** [S.l : s.n], 1998.

CHRISTOFOROS, K. E.; PATTERSON, D. A New Direction for Computer Architecture Research. **Computer**, [S.l.], v.31 n. 11, p. 24-32, Nov. 1998.

CIFUENTES, C.; LEWIS, B.; UNG, D. **Walkabout – A Retargetable Dynamic Binary Translation Framework**. [S.l]: SUN Microsystems, 2002. (SMLI TR-2002-106).

COMPTON, K.; HAUCK, S. Reconfigurable computing: A survey of systems and software. **ACM Computing Surveys**, New York, v. 34, n. 2, p. 171-210, June 2002.

CONTE, Thomas M. et al. Challanges to Combining General-Purpose and Multimedia Processors. **IEEE Computer Magazine**, Los Alamitos, v. 30, n. 12, p. 33-37, Dec. 1997.

DEHON, A. DPGA Utilization and Application. In: INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE GATE ARRAYS, FPGA, 1996. **Proceedings…** [S.l.]: ACM/SIGDA, 1996, p. 115-121.

DEHON, A. **Reconfigurable Architectures for General-Purpose Computing**. 1996. PhD Thesis. Massachusetts Institute of Technology.

EBELING, C. et al. RaPiD: Reconfigurable Pipelined Datapath**.** In: INTERNATIONAL WORKSHOP ON FIELD PROGRAMMABLE LOGIC AND COMPILERS, 6., 1996. **Proceedings…** [S.l.] **:**Springer-Verlag, 1996. p. 126-135. (Lecture Notes in Computer Science).

GOLDSTEIN, S. C. et al. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 26., 1999. **Proceedigns…** [S.l.]: IEEE, 1999. p 28-39.

GOLDSTEIN, S. et al. PipeRench: A Reconfigurable Architecture and Compiler. In: INTERNATIONAL WORKSHOP ON SYSTEMS, ARCHITECTURES, MODELING, AND SIMULATION, SAMOS, 3., 2003, Samos, Greece. **Proceedings…** [S.l.: s.n.], 2003.

GUCCIONE, S. A.; GONZALEZ, M.. Classification and Performance for Reconfigurable Architectures. In: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC APPLICATION, FPG, 1995. **Field-Programmable Logic and Applications**. London: Springer, 1995, p. 439-448. (Lecture Notes in Computer Science, v. 975).

GUPTA, R. K., MICHELI, G. D. Hardware-software co-synthesis for digital systems. **IEEE Design & Test**, Los Alamitos, v. 10, n. 3, p. 29-41, July 1993.

HARTEINSTEIN, R. et al. Using the KressArray for Reconfigurable Computing. In: CONFERENCE ON CONFIGURABLE COMPUTING: TECHNOLOGY AND APPLICATIONS, 1998, Boston, USA. **Proceedings…** [S.l.]: SPIE, 1998. p.150-161.

HARTENSTEIN, R. A Decade of Reconfigurable Computing: A Visionary Retrospective. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION, DATE, 2001, Munich, Germany. **Proceedings…** Los Alamitos: IEEE Computer Society, 2001. p. 642-649.

HAUCK, S. Configuration Prefetch for Single Context Reconfigurable Coprocessor. In: INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE GATE ARRAYS, FPGA, 6., 1998, Montery. **Proceedings…** [S.l.]: ACM/SIGDA, 1998. p. 65-74.

HAUCK, S. et al. The Chimaera Reconfigurable Functional Unit. In SYNPOSIUM ON FPGA-BASED CUSTOM COMPUTING MACHINES, 5., 1997, Napa Valley. **Proceedings…** Los Alamitos: IEEE, 1997. p. 87-96.

HAUCK, S.; LI, Z.; SHWABE, E. Configuration compression for the Xilinx XC6200 FPGA. In: SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, FCCM, 1998. **Proceedings…** Los Alamitos: IEEE, 1998b.

HAUCK, S. The roles of FPGAs in reprogrammable systems. **Proceedings of the IEEE**, Piscataway, v. 86, n 4, p. 615-638, Apr. 1998.

HAUSER, J.; WAWRYNEK, J. Garp: a MIPS Processor with a Reconfigurable Coprocessor. In: ANNUAL IEEE SYMPOSIUM ON FPGAS FOR CUSTOM COMPUTING MACHINES, FCCM, 5., 1997. **Proceedings…** Los Alamitos: IEEE, 1997. p. 24-33.

ITO, S. A.; CARRO, L.; JACOBI, R. P. System Design Based on Single Language and Single-Chip Java ASIP Microcontroller. In: DESIGN AUTOMATION AND TEST IN EUROPE, 2000, Paris, France. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 2000. p.703-707.

KASTRUP, B.; BINK, A.; HOOGERBURGGE, J. ConCISe a Compiler-drive CPLD-based Instruction Set Accelerator. In: ANNUAL SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 7., 1999, Napa Valley, CA. **Proceedings…** Los Alamitos: IEEE, 1999. p. 92-100.

KREUZINGER, J. et al. The Komodo Project: Thread-based Event Handling Supported by a Multihreaded Java Microncontroller. In: EUROMICRO CONFERENCE, 25., 1999. **Proceedings...** Los Alamitos: IEEE, 1999. p. 2122-2128.

LAWTON, G. Moving Java into Mobile Phones. **Computer**, Los Alamitos, v. 35, n. 6, p. 17-20, 2002.

LEITE, N. J.; BARROS, M. A. A Highly Reconfigurable Image Processor based on Functional Programming. In: INTERNATIONAL CONFERENCE ON IMAGE PROCESSING, 1994, Austin. USA. **Proceedings…** Los Alamitos: IEEE, 1994.

LEWIS, T. Information Appliances: Gadget Netopia. **Computer**, Los Alamitos, v.31, n. 1, p. 59-68, Jan., 1998.

LI, Z.; COMPTON, K.; HAUCK, S. Configuration Caching Management Techniques for Reconfigurable Computing. In: SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, FCCM, 2000. **Proceedings…** Los Alamitos: IEEE, 2000. p. 22-36.

LODI, A. et al. A VLIW Processor With Reconfigurable Instruction Set for Embedded Applications. **IEEE Journal of Solid-State Circuits**, Dallas, v. 38, n. 11, p. 1876-1886, Nov. 2003.

LU, G. et al. The MorphoSys Parallel Reconfigurable System. In: EUROPEAN CONFERENCE ON PARALLEL PROCESSING, 1999. **Proceedings…** [S.l.: s.n.], 1999.

MARSHALL, A. et al. A Reconfigurable Arithmetic Array for Multimedia Applications. In: INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, FPGA, 1999, Monterey. **Proceedings…** [S.l.]: ACM/SIGDA, 1999. p. 135-143.

McATEER S. **Java will be the Dominant Handset Platform**. 2002. Disponível em: <www.microjava.com/articles/perspective/zelos/>. Acesso em: 20 jan. 2004.

MEI, B. et al. DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures. In: INTERNATIONAL CONFERENCE ON COARSE-GRAINED RECONFIGURABLE ARCHITECTURES, FPL, 2002. **Proceedings…** Los Alamitos: IEEE, 2002. p. 166-173.

MITSUSYAMA, Y. et al. A Dynamically Reconfigurable Hardware-Based Cipher Chip. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2001, Yokohama, Japan. **Proceedings…** [S.l.]: ACM, 2001.

MORENO, N. B.; ARAÚJO, G. C. S. de. **Alocação de Recursos em Arquiteturas Reconfiguráveis**. 2003. Exame de Qualificação Específico, Laboratório de Sistemas de Computação – Instituto de Computação – UNICAMP, Campinas.

MULCHANDANI D. Java for Embedded Systems. **Internet Computing**, [S.l.], v.31, n.10, p. 30-39, May 1998.

O'CONNOR, J. M.; TREMBLAT, M. Picojava-I: the Java Virtual Machine in Hardware. **IEEE Micro**, [S.l.], v. 17, n. 2, p. 45-53, Mar./Apr. 1997.

PAGE, I. Reconfigurable Processor Architectures. **Microprocessors and Microsystems**, Guildford, May 1996.

RABAEY, J. Reconfigurable Computing: The Solution to Low Power Programmable DSP. In: INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, ICASSP, Munich, Germany, 1997. **Proceedings…** [S.l.: s.n.], 1997.

RADUNOVIC, B.; MILUTINOVIC, V. A Survey of Reconfigurable Computing Architectures. In: INTERNATIONAL WORKSHOP ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 8., 1998, Tallin, Estonia. **Proceedings…** [S.l.: s.n.], 1998, p. 376-385.

RAZDAN, R.; SMITCH, M. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 27., 1994. **Proceedings…**New York: ACM Press, 1994. p. 172-180.

RUPP, C. et al. The NAPA adaptive processing architecture. In: IEEE SYMPOSIUM ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1998. **Proceedings…** [S.l.]: IEEE, 1998. p. 28-37.

SANKARALINGAM, K. et al. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 30., ISCA, 2003, San Diego. **Proceedings…** New York: ACM Press, 2003.

SHIRAZI, N.; LUK, W.; CHEUNG P. Automating production of run-time reconfigurable designs. In: IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, FCCM, 1998. **Proceedings…** [S.l.]: IEEE, 1998. p. 147-156.

SHIRAZI, W. L.; GUO, S.; SHEUNG, P. Pipeline morphing and virtual pipelines. In Lectur Notes is Computer Science. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 1997. **Field-Programmable Logic and Applications.** Berlin: Springer, 1997. p. 111-120. (Lecture Notes in Computer Science, v. 1304).

SILVA, I. S. Arquiteturas Reconfiguráveis: Teoria e Prática. In: ESCOLA DE MICROELETRÔNICA DA SBC-SUL, EMICRO, 3., 2001, Santa Maria. **Livro Texto**. Porto Alegre: Instituto de Informática da UFRGS, 2001. p. 161-186.

STITT, G.; VAHID F. Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic. **IEEE Design and Test of Computers**, Los Alamitos, v. 19, n. 6, p. 36-43, Nov./Dec. 2002.

SUN MICROSYSTEMS. **PicoJava-II Microarchitecture Guide**. [S.l.], 1999.

VDC-CORP. **The Embedded Software Strategic Market Intelligence**. Java in Embedded Systems. Disponível em: <http:/www.vdc-corp.com/>. Acesso em: 20 jan. 2004.

WAINGOLD, E. et al. Baring it all to Software: RAW Machines. **Computer**, Los Alamitos, v. 30, n. 12, p. 86-93, Sept. 1997.

WAWRSYNEK, J. et al. Spert-II: A Vector Microprocessor System. **Computer,** Los Alamitos**,** v. 29, n. 3, p. 79-86, Mar. 1996.

WILCOX, K.; MANNE, S. **Alpha Processors:** A History of Power Issues and a Look to the Future. 1999. Disponível em : < http://citeseer.ist.psu.edu/context/1227750/0 >. Acesso em: mar. 2005.

WITTIG, R.; CHOW, P. OneChip: An FPGA processor with reconfigurable logic. In: IEEE SYMPOSIUM ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1996. **Proceedings…** [S.l.]: IEEE, 1996.

CD ANEXO

Um CD juntado a este trabalho possui todas as tabelas, gráficos, códigos e aplicações referenciados no texto, organizados nas pastas seguintes.

Experimentos CACO-PS

Contém as descrições Javarray configuradas para os experimentos citados na dissertação.

Java Reference

Contém cópia da documentação on-line da JVM.

Javarray Analysers

Contém os fontes dos programas criados para analisar os *profiles* das aplicações Java utilizadas.

Modelo Quartus – Javarray Seqüencial

Contém os fontes do modelo VHDL da versão seqüencial do Javarray.

Módulos CACO-PS Criados

Contém descrições dos componentes CACO-PS criados para o Javarray.

CACO-PS.zip

Contém a instalação do CACO-PS.

Manual_CACO-PS.pdf

Contém um manual do CACO-PS.