

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

THIAGO BERTICELLI LÓ

**Virtualização de Hardware e Exploração da
Memória de Contexto em Arquiteturas
Reconfiguráveis**

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

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, março de 2012.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Ló, Thiago Berticelli

Virtualização de Hardware e Exploração da Memória de Contexto em Arquiteturas Reconfiguráveis / Thiago Berticelli Ló – Porto Alegre: Programa de Pós-Graduação em Computação, 2012.

98 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2012. Orientador: Luigi Carro.

1.Arquiteturas Reconfiguráveis. 2.Sistemas Embarcados
3.Virtualização de Hardware 4.Memória de Contexto I. Carro, Luigi. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitora de Pós-Graduação: Prof. Aldo Bolten Lucion

Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb

Coordenadora do PPGC: Prof. Alvaro Freitas Moreira

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, que com muito carinho e apoio, não mediram esforços para que eu chegasse até essa etapa da minha vida.

Agradeço aos meus colegas do laboratório de Sistemas Embarcados, que durante essa formação demonstraram-se amigos fiéis, me ajudando sempre que precisei.

Agradeço a orientação do professor Luigi Carro, pelo interesse e tempo dedicado a este trabalho.

SUMÁRIO

AGRADECIMENTOS	3
LISTA DE ABREVIATURAS E SIGLAS	6
LISTA DE FIGURAS	7
LISTA DE TABELAS	9
RESUMO.....	10
ABSTRACT.....	11
1 INTRODUÇÃO.....	13
1.1 Motivação	14
1.1.1 Área ocupada pelo sistema reconfigurável	14
1.1.2 Impacto da Memória de Contexto	15
1.2 Contribuições deste trabalho	16
1.3 Organização deste trabalho	17
2 CONTEXTUALIZAÇÃO	19
2.1 Classificação das Arquiteturas Reconfiguráveis.....	19
2.1.1 Acoplamento	19
2.1.2 Granularidade.....	20
2.1.3 Reconfigurabilidade	20
2.1.4 Mecanismos de Reconfiguração	21
2.2 Memória de contexto	21
2.3 Virtualização de Hardware.....	22
2.3.1 Particionamento temporal	22
2.3.2 Execução virtualizada	23
2.3.3 Máquina virtual.....	24
2.4 ARISE	24
2.5 Exemplos de Arquiteturas Reconfiguráveis Propostas	25
3 DESCRIÇÃO DA ARQUITETURA RECONFIGURÁVEL	29
3.1 Estrutura.....	30
3.2 Interconexões.....	32
3.3 Tradução binária	34
3.4 Memória de Contexto	36
3.5 Análise dos resultados	37
3.5.1 Ambiente de Simulação	38
3.5.2 Desempenho.....	38
4 VIRTUALIZAÇÃO DE HARDWARE.....	47

4.1	Visão temporal da execução na arquitetura reconfigurável.....	47
4.2	Virtualização de hardware por meio da técnica de <i>pipeline</i> de estágio reconfigurável	51
4.2.1	<i>Pipeline</i> virtual de 1 estágio.....	52
4.2.2	<i>Pipeline</i> virtual de 2 estágios	53
4.2.3	Implicações da utilização da virtualização por meio de <i>Pipeline</i>	56
4.3	<i>Pipeline</i> virtual	61
4.4	Resultados.....	71
5	MECANISMOS DE GERENCIAMENTO DA MEMÓRIA DE CONTEXTO .	79
5.1	Explorando grão de acesso a memória	79
5.1.1	Configuração total.....	82
5.1.2	Configuração parcial.....	83
5.2	Busca sob demanda.....	86
5.2.1	Operações básicas	87
5.2.2	Consumo de energia.....	88
5.2.3	Área.....	89
6	CONCLUSÕES E TRABALHOS FUTUROS.....	91
6.1	Trabalhos Futuros	92
6.1.1	Virtualização do tradutor binário	92
6.1.2	Interconexões das unidades funcionais com Rede Omega	92
6.1.3	Adição de unidades reconfiguráveis mais complexas	93
6.1.4	Política de troca das configurações na memória de contexto	93
6.1.5	Explorar as técnicas de virtualização em outras arquiteturas reconfiguráveis ...	93
6.1.6	Impacto da memória de contexto em outras arquiteturas reconfiguráveis	93
	REFERÊNCIAS.....	95

LISTA DE ABREVIATURAS E SIGLAS

PDA	Personal Digital Assistants
ILP	Instruction Level Parallelism
ASIP	Application Specific Instruction Set Processor
IPC	Instructions Per Cycle
CMOS	Complementary Metal-Oxide-Semiconductor
RC	Reconfigurable Computing
UF	Unidades Funcionais
RISC	Reduced Instruction Set Computer
PISA	Portable Instruction Set Architecture
RFU	Reconfigurable Function Unit
GPP	General Purpose Processor
VLIW	Very Long Word Instruction
FPGA	Field Programmable Gate Array
CLB	Configurable Logic Block
ULA	Unidade Lógica Aritmética
GDD	Grafo de Dependência de Dados
PE	Processing Elements
SRAM	Static Random Access Memory
FIFO	First In First Out
RTL	Register Transfer Level
TB	Tradutor Binário
PC	Program Counter
VHDL	VHSIC Hardware Description Language
FAT	File Allocation Table

LISTA DE FIGURAS

Figura 1.1: Estimativa do consumo de energia.	15
Figura 2.1: Otimização da matriz reconfigurável através da ferramenta ARISE.....	24
Figura 2.2: Arquitetura do PipeRench (GOLDSTEIN, 1999).....	26
Figura 2.3: Virtualização de cinco estágios do <i>pipeline</i> em três estágios físicos (GOLDSTEIN, 1999).	27
Figura 3.1: Organização do sistema DIM. (1) Matriz reconfigurável; (2) processador; (3) TB (4) Memória de Contexto, juntamente com as caches de Instrução e de Dados do processador.	30
Figura 3.2: Visão geral da matriz reconfigurável.	31
Figura 3.3: Interconexões entre as unidades funcionais.	33
Figura 3.4: Exemplo de uma mapeamento de instruções.	34
Figura 3.5: Alocação das unidades funcionais para determinada sequência de instruções.	36
Figura 3.6: Memória de Contexto.	37
Figura 3.7: Relação de instruções/salto das aplicações avaliadas no trabalho.	38
Figura 3.8: Aceleração média para cada configuração.	40
Figura 3.9: Aceleração para Configuração 1 para cada aplicação do <i>benchmark Mibench</i>	40
Figura 3.10: Aceleração para Configuração 2 para cada aplicação do <i>benchmark Mibench</i>	41
Figura 3.11: Aceleração para Configuração 3 para cada aplicação do <i>benchmark Mibench</i>	41
Figura 3.12: Aceleração para Configuração 4 para cada aplicação do <i>benchmark Mibench</i>	42
Figura 3.13: Aceleração obtida para cada configuração, fixando o tamanho da memória de contexto em 64 entradas.	43
Figura 3.14: Distribuição do reuso das configurações na execução de todo <i>benchmark</i>	43
Figura 3.15: Distribuição do reuso x número de instruções das configurações na execução de todo <i>benchmark</i>	44
Figura 3.16: Distribuição do reuso x número de instruções das configurações na execução da aplicação <i>GSMC</i>	45
Figura 4.1: Execução de uma configuração com 6 níveis (parte inicial).	48
Figura 4.2: Execução de uma configuração com 6 níveis (parte final).	49
Figura 4.3: Inserção do registrador na linha de contexto para desativação dos níveis já processados.	51
Figura 4.4: Nível com registradores nas saídas das linhas de contexto.	52
Figura 4.5: Virtualização por meio de 1 estágio físico.	52
Figura 4.6: Virtualização por meio de 2 estágios físicos.	54

Figura 4.7: Mapeamento dos níveis virtuais nos 2 estágios.	55
Figura 4.8: Variação da granularidade de cada estágio.	57
Figura 4.9: Diferentes formas de realizar somas sucessivas por solução combinacional (a) e sequencial (b).	58
Figura 4.10: Caminho crítico de um circuito de somador <i>Ripple-Carry</i> em cascata.	59
Figura 4.11: Modificação da estrutura do estágio para que o sinal de saída possa ter origem combinacional (b) ou registrada (c).	62
Figura 4.12: Matriz não virtualizada composta de ULAs.	63
Figura 4.13: Execução de uma configuração de 6 níveis não virtualizada(parte inicial).	64
Figura 4.14: Execução de uma configuração de 6 níveis não virtualizada(parte final)..	65
Figura 4.15: Caminho combinacional entre os estágios.	66
Figura 4.16: Virtualização por meio de 3 estágios físicos.	67
Figura 4.17: Execução de uma configuração de 6 níveis virtualizada em 3 estágios(parte inicial).	68
Figura 4.18: Execução de uma configuração de 6 níveis virtualizada em 3 estágios(parte final).	69
Figura 4.19: Mapeamento dos níveis virtuais nos níveis físicos (estágios).	70
Figura 4.20: Tempo de execução em função do número de níveis.	72
Figura 4.21: Tempo de execução em função do número de níveis.	73
Figura 4.22: Relação entre o tempo de computação das técnicas de virtualização e o tempo de computação do circuito combinacional.	73
Figura 4.23: Comparativo da área ocupada entre as técnicas de virtualização em função do número de níveis.	74
Figura 4.24: Aceleração para a matriz com número de níveis ilimitados para cada aplicação do <i>Mibench</i>	77
Figura 4.25: Aceleração média para cada configuração da matriz reconfigurável.	77
Figura 5.1: Exploração da memória de contexto dividindo a configuração em partes. .	80
Figura 5.2: Energia total de acesso a uma configuração com diferentes larguras de porta.	81
Figura 5.3: Consumo de energia para carregar uma configuração com diferentes grãos de acesso.	82
Figura 5.4: Energia consumida com diferentes grãos de acesso utilizando reconfiguração total.	83
Figura 5.5: Energia consumida com diferentes grãos de acesso utilizando a reconfiguração parcial.	84
Figura 5.6: Fragmentação externa da memória de contexto.	85
Figura 5.7: Memória de contexto proposta.	86
Figura 5.8: Mecanismo da memória de contexto proposta.	88
Figura 5.9: Consumo de energia pelo sistema reconfigurável.	89

LISTA DE TABELAS

Tabela 3.1: Grupos de unidades funcionais e suas operações.	32
Tabela 3.2: Diferentes configurações da matriz reconfigurável analisadas.	39
Tabela 3.3: Porcentagem do número de níveis utilizados na execução de todo benchmark.	44
Tabela 3.4: Distribuição instruções executadas na arquitetura reconfigurável na execução de todo <i>benchmark</i>	44
Tabela 3.5: Valores de área relativa ocupada pela matriz reconfigurável e o processador MIPS R3000, a capacidade necessária da memória de contexto (64 entradas) e a aceleração.	46
Tabela 4.1: Atraso de somadores <i>Rippe-Carry</i> e <i>Carry-Lookahead</i> em cascata.	60
Tabela 4.2: Atraso das unidades lógicas e aritméticas (ULAs) em cascata.	60
Tabela 4.3: Atraso da matriz reconfigurável com 6 ULAs em paralelo e 6 linhas de contexto.	61
Tabela 4.4: Atraso para a matriz de ULAs com 6 níveis.	71
Tabela 4.5: Atraso para a matriz de ULAs com 18 níveis.	72
Tabela 4.6: Configuração 1 da matriz reconfigurável.	75
Tabela 4.7: Resultado para a configuração 1.	75
Tabela 4.8: Configuração 2 da matriz reconfigurável.	76
Tabela 4.9: Resultados para configuração 2.	76
Tabela 5.1: Área da memória de contexto (em mm ²) para as diferentes configurações com distintas larguras de porta.	84
Tabela 5.2: Energia consumida (mJ) na execução de todas as aplicações do <i>benchmark</i>	88
Tabela 5.3: Comparativo da área (em mm ²) da memória de contexto do mecanismo original e o proposto.	90

RESUMO

Arquiteturas reconfiguráveis têm se demonstrado uma potencial solução para lidar com a crescente complexidade encontrada em sistemas embarcados. Para se alcançar ganhos em desempenho, é preciso uma grande redundância das unidades funcionais, acarretando o aumento da área ocupada pelas unidades funcionais. Uma das propostas deste trabalho será de explorar o espaço de projeto, visando à redução da área e da energia. Para isto, serão apresentadas duas técnicas de virtualização de hardware, sendo as mesmas semelhantes a um *pipeline* de estágios reconfiguráveis. Ambas as técnicas alcançaram mais de 94% de redução da área. Outro aspecto a ser explorado em uma arquitetura reconfigurável é o impacto em área e energia causado pela inserção da memória de contexto. Assim, este impacto será demonstrado neste trabalho e duas abordagens que modificam a memória de contexto serão propostas: a primeira abordagem baseia-se na exploração da largura ideal da porta da memória combinado com número de acessos, para que se minimize a energia consumida na busca dos bytes de configuração; a segunda abordagem possui um mecanismo de gerenciamento das configurações por meio de listas ligadas, que permite que as configurações sejam acessadas parcialmente. As duas abordagens apresentaram redução de energia de até 98%, podendo ser utilizadas em sistemas que apresentam tanto a reconfiguração parcial como a total.

Palavras-Chave: sistemas embarcados, arquiteturas reconfiguráveis, exploração da memória de contexto, virtualização de hardware, redução de energia.

Hardware Virtualization and Investigation of Context Memory in Reconfigurable architectures

ABSTRACT

Reconfigurable architectures have shown to be a potential solution to the problem of increasing complexity found in embedded systems. However, in order to achieve significant performance gains, large quantities of redundant functional units are generally necessary, with a corresponding increase in the area occupied by these units. This thesis explores the design space with the objective of reducing both area and energy consumption, and presents two hardware virtualization techniques, similar to reconfigurable pipeline stages, which achieve a reduction in area of more than 94%. The use of context memory in reconfigurable architectures has a significant impact in terms of area and energy, as is clearly demonstrated by initial experimental results. Two novel context memory architectures are presented: the first approach is being based on an exploration of the balance point between memory port width and number of accesses, in order to reduce the energy consumed during fetching of the configuration bytes; the second approach presents a configuration management mechanism using hardware linked lists, and that allows segmented access to configuration settings. Both approaches demonstrate energy reduction of up to 98% and can be adopted in both partial and atomic reconfiguration architectures.

Keywords: embedded systems, reconfigurable architectures, context memory exploration, hardware virtualization, energy reduction.

1 INTRODUÇÃO

O avanço da tecnologia de circuitos integrados permitiu uma maior integração dos transistores ao longo dos anos. Vários produtos eletrônicos de consumo têm se beneficiado com isso, possibilitando adicionar diversas funcionalidades em um único dispositivo eletrônico, como nos modernos telefones celulares. Apesar de várias características estarem convergindo para o domínio de propósito geral, esses sistemas ainda são chamados de sistemas embarcados. Estes devem executar várias aplicações heterogêneas que apresentam um comportamento variado, que podem ser classificadas como orientadas a controle (*Controlflow*), ou como orientadas a dados (*Dataflow*). Ainda, os sistemas embarcados estão progressivamente mais complexos e necessitam de computação de alto desempenho e também estão ligados a uma série de limitações, tais como alimentação, energia, memória, custo e *time-to-market*. Todos estes fatores tornam o projeto de sistemas embarcados um grande desafio.

A dificuldade em obter o melhor desempenho na execução de somente um conjunto de aplicações conduz o projetista a desenvolver uma arquitetura eficiente somente para realizar esta tarefa. O resultado desse tipo de abordagem produz um processador para uma aplicação específica (*Application Specific Processor - ASIP*) (JAIN, 2001). Tal conceito produz processadores não flexíveis, apesar de apresentar alto desempenho e baixo consumo de energia quando comparados aos processadores de propósito geral. No segmento de sistemas embarcados é proveitosa a utilização desse tipo de processadores. Devido aos compromissos de projeto (custo, tamanho do código, desempenho e potência) (KUCUKCAKAR, 1999), um projeto customizado para um conjunto de aplicações fornece a eficiência necessária para um sistema embarcado. Entretanto, a acelerada convergência de funcionalidades para um mesmo dispositivo tornará inviável a utilização de ASIPs para executar múltiplas funções em futuros dispositivos embarcados. Exemplos dessa convergência são *tablets* e *smartphones*, já que cada nova funcionalidade a ser instalada após a fabricação do produto pode possuir um comportamento diferente daquele anteriormente previsto. Consequentemente, a quantidade de paralelismo (*Instruction Level Parallelism - ILP*) a ser explorado em cada nicho de aplicação é muito díspar (WALL, 1991). Desta maneira, os processadores ASIPs como tradicionalmente concebidos não conseguem manipular eficientemente essa propriedade.

Processadores superescalares exploram o paralelismo em tempo de execução, no nível de instruções, podendo fornecer um alto índice de aceleração na execução de

aplicações. Entretanto, utiliza-se essa abordagem somente quando o projeto do sistema não possui restrições de potência, devido ao alto consumo de potência do mecanismo de detecção de paralelismo desta abordagem. Mesmo essa abordagem, no entanto, está chegando ao limite teórico de exploração de paralelismo. Como é mostrado em (FLYNN, 2005) e (SIMA, 2004), não há grandes novidades em tais sistemas: eles estão enfrentando sérios problemas para aumentar a taxa de instruções executadas por ciclo de relógio (*Instructions Per Cycle* - IPC). Os recentes aumentos no desempenho ocorreram principalmente devido ao aumento em frequência do relógio por meio do emprego de pipelines profundos.

Uma nova abordagem que surgiu no mercado como solução para os problemas relatados anteriormente são as arquiteturas reconfiguráveis (GUPTA, 1993) (STITT, 2002). Esse tipo de arquitetura, ao contrário de processadores ASIP, provê flexibilidade de execução, pelo fato de ter a capacidade de se adaptar ao comportamento de execução das aplicações. Além disso, as arquiteturas reconfiguráveis são capazes de explorar de forma mais eficiente o paralelismo de uma aplicação.

As arquiteturas reconfiguráveis são compostas de unidades de lógica reconfigurável. Essas unidades podem implementar operações aritméticas, funções lógicas ou ainda funções mais complexas. As interconexões entre as unidades também são reconfiguráveis. Para uma dada sequência de operações ou instruções, existe um conjunto de bits de controle, chamado contexto, responsável tanto por definir a operação da unidade reconfigurável quanto definir as interconexões entre as unidades, determinando, assim, a funcionalidade do circuito. Cada contexto é armazenado em uma memória especial, chamada memória de contexto, que é uma das maiores responsáveis pela flexibilidade provida pelos sistemas reconfiguráveis.

A ideia básica das arquiteturas reconfiguráveis para acelerar as aplicações é de descobrir partes do código que possuem grande impacto no tempo de execução, chamadas de *kernels*, e executá-las na arquitetura reconfigurável (HUTCHINGS, 1997). A descoberta de um *kernel* pode ser realizada em tempo de compilação ou execução, dependendo da técnica utilizada. Ganhos são obtidos pela natureza de execução das aplicações, ou seja, essa técnica explora a repetição da execução na unidade reconfigurável dos *kernels* previamente descobertos. Além disso, com a utilização da lógica combinacional, em vez da sequencial, é possível obter enormes ganhos de desempenho com economia de energia (VENKATARAMANI, 2001) (STITT, 2002), ao preço do aumento da área e de potência consumida.

1.1 Motivação

1.1.1 Área ocupada pelo sistema reconfigurável

A utilização dessa abordagem implica na inserção de vários blocos de unidades funcionais, de sistema de interconexão entre as unidades e de uma memória grande suficiente para armazenar uma ou mais sequências de bits de configuração. Para se alcançar ganhos em desempenho, há a necessidade de uma grande redundância das unidades funcionais. O acréscimo do número de unidades funcionais acarreta o aumento da área ocupada pelas unidades funcionais e as interconexões entre as mesmas. Também é necessária uma memória de contexto maior, uma vez que se faz necessário um maior número de bits de configuração. Consequentemente, há um acréscimo considerável envolvendo a área no sistema final.

1.1.2 Impacto da Memória de Contexto

Muitos trabalhos que exploram a redução da área de arquiteturas reconfiguráveis somente focam seus esforços em otimizar as unidades funcionais e as interconexões, o impacto da memória de contexto tem sido negligenciado. Como será mostrado, a memória de contexto tem uma grande significância tanto em potência, como em desempenho e área.

Se o processo de reconfiguração em um determinado sistema ocorre muitas vezes, de modo que haja um grande número de mudanças de contexto em um curto período de tempo, a potência média da memória do contexto pode ser tão grande ou até maior que a potência dissipada para a realização da computação. Para arquiteturas reconfiguráveis que usam um pequeno número de configurações, que atuam em *Kernels* muito específicos, a potência dissipada pela memória de contexto pode ser amortizada por meio do uso contínuo ao longo do tempo, ou seja, a configuração é carregada uma vez e executada várias vezes sem necessidade de reconfiguração.

Em arquiteturas reconfiguráveis como em Goldstein (1999) e Marshall (1999) pode-se estimar a energia total consumida observando a potência dissipada em todas as unidades funcionais, além de estimar o número total de bits de configuração com o intuito de obter o tamanho e, portanto, a potência de acesso à memória de reconfiguração. Ao mesmo tempo é necessário saber quantos bits estão sendo programados por reconfiguração de forma a poder estimar a largura da porta da memória, uma vez que essa tem um enorme impacto sobre a potência da memória e desempenho.

Com base nessas estimativas, a Figura 1.1 mostra a energia consumida para diferentes acelerações, ou seja, são baseadas em diferentes tamanhos de matrizes reconfiguráveis. A aceleração depende basicamente do conjunto de aplicações e do número de unidades funcionais (UF) disponível na arquitetura reconfigurável. O número de UFs afeta diretamente a potência da arquitetura e a potência da memória de reconfiguração, uma vez que mais bits de reconfiguração significa uma memória de reconfiguração maior. Podemos observar que, embora matrizes reconfiguráveis maiores signifiquem maior aceleração, a quantidade de bits extra na largura porta da memória tem um grande preço, e com o aumento do tamanho do matriz, a energia também aumenta, uma vez que a aceleração extra não compensa o aumento da potência.

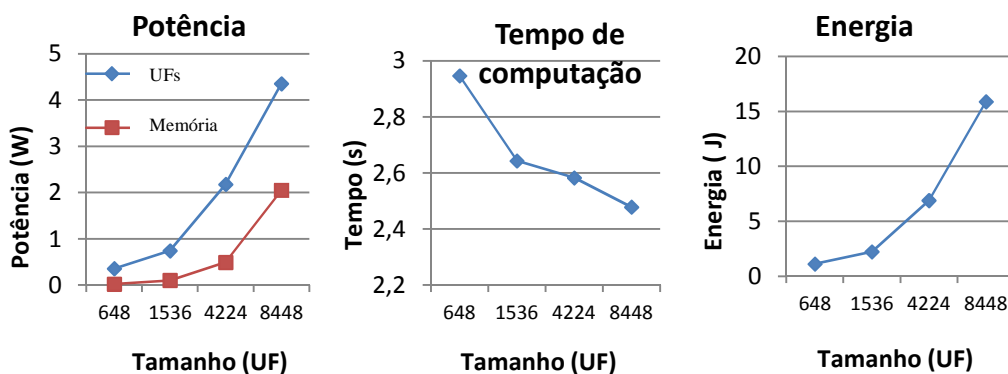


Figura 1.1: Estimativa do consumo de energia.

Contudo, em arquiteturas como (BECK, 2005) e (LYSECKY, 2004), em que a reconfiguração é dinâmica, ou seja, as configurações são realizadas em tempo de execução, a estimativa apresentada na Figura 1.1 não é válida, porque o número de acessos à memória de reconfiguração depende da execução da aplicação.

Trabalhos recentes têm mostrado que a reconfiguração dinâmica é uma tendência clara tanto para arquiteturas de grão fino (*fine grain*) quanto grão grosso (*coarse grain*). Mostrando diversos benefícios que não podem ser alcançados pelos sistemas de reconfiguração estática (BECK, 2008) (CLARK, 2005) (LYSECKY, 2004), o que reforça a necessidade de otimizações na memória de contexto. Por todos esses fatores citados é necessário aprimorar as técnicas para redução de área e potência para que a aplicação de sistemas reconfiguráveis não se torne proibitivo em sistemas embarcados.

1.2 Contribuições deste trabalho

Neste contexto, o grupo de Sistemas Embarcados da UFRGS desenvolveu um sistema reconfigurável chamado DIM (*Dynamic Instruction Merging*). O projeto obteve os primeiros resultados em 2005 com o trabalho (BECK, 2005), quando foi apresentada a técnica de tradução binária. O sistema era composto de um processador Java, uma unidade funcional reconfigurável e o tradutor binário; posteriormente, em (BECK, 2006) foi demonstrada uma variação da técnica de tradução binária. Nessa abordagem foi utilizado um modelo de um processador RISC que executa o conjunto de instruções PISA.

Desde então aprimoramentos foram realizados com novos trabalhos. Em sua grande parte os trabalhos têm como objetivo minimizar a área e consumo de energia e aumentar o desempenho do sistema. Nos trabalhos (RUTZIG, 2008) (RUTZIG, 2008a), os autores abordaram o problema considerando a abstração de unidades funcionais. Foi desenvolvida uma ferramenta chamada ARISE, que dado um conjunto de aplicações, explícita a quantidade de unidades funcionais necessárias em cada linha (unidades funcionais disponíveis em paralelo), bem como a quantidade de níveis da arquitetura reconfigurável. Desta forma, consegue-se gerar uma arquitetura otimizada com menor área e potência consumida, que comparada a da arquitetura original, chega a uma redução de área de até 90%, porém com redução média de 41%. Para minimizar o consumo de potência foi utilizada a técnica de *Sleep Transistor* (TSCHANZ, 2003), atingindo uma redução de até 5,5 vezes. Cabe salientar que esses valores de redução de área e energia correspondem somente às UFs e não ao sistema como um todo.

Em outro trabalho (FERREIRA, 2008), os pesquisadores abordam o problema de redução de área otimizando as interconexões. Partindo-se do fato de que aproximadamente 50% da área desta arquitetura reconfigurável são interconexões, o trabalho sugere que as unidades funcionais sejam conectadas ao contexto por Redes Omega (LAWREI, 1975), e não por multiplexadores. Com essa abordagem espera-se uma redução de 33% na área de interconexão e de 17% na área total em média. Novamente as otimizações são aplicadas somente nas interconexões da matriz reconfigurável.

A proposta deste trabalho é explorar o espaço de projeto utilizando a mesma arquitetura reconfigurável citada anteriormente também visando à redução da área e da energia, considerando a necessidade de abordagens distintas das apresentadas. A ideia básica é reutilizar as UFs que já realizaram o seu processamento e configurá-las para realizar o processamento da etapa seguinte, técnica semelhante à de um *pipeline*. Essa

abordagem utiliza as técnicas de virtualização de hardware, quais permitem reduzir a área ocupada pelas UFs e apresentam melhor eficiência no uso das unidades. Em uma das técnicas de virtualização apresentadas no Capítulo 4, diferentemente de um *pipeline*, não há no caminho crítico um registrador no final de cada estágio; assim, não se perde a vantagem em desempenho que o circuito combinacional proporciona.

A outra abordagem é relacionada à memória de contexto. Essa exploração do espaço de projeto apresenta grande importância, pois a memória de contexto tem impacto significativo em relação ao consumo de energia, área e desempenho em arquiteturas reconfiguráveis. Também são apresentadas novas alternativas para gerenciamento da memória de contexto em sistemas reconfiguráveis. Para isso foram desenvolvidas novas organizações para a memória de contexto, que permitem obter menor consumo de energia e utilização mais eficiente do espaço de memória.

Ambas as abordagens utilizadas têm mínima perda de aceleração que o sistema reconfigurável inicial proporciona. Além disso, as técnicas apresentadas nos trabalhos anteriormente citados também podem ser aplicadas conjuntamente para obter um sistema ainda mais otimizado.

1.3 Organização deste trabalho

No Capítulo 2 serão apresentados alguns estudos relacionados às arquiteturas reconfiguráveis, além de apresentar o impacto da memória de contexto, considerando diferentes arquiteturas reconfiguráveis.

No Capítulo 3 será apresentada a arquitetura reconfigurável utilizada como estudo de caso deste trabalho. Além disso, serão expostos dados de desempenho que o sistema reconfigurável pode alcançar.

As técnicas de virtualização de hardware, bem como os resultados serão discutidos no Capítulo 4. Nesse capítulo são ilustradas três técnicas de virtualização de hardware com suas respectivas vantagens e desvantagens.

No Capítulo 5 são detalhados os mecanismos propostos para gerenciamento da memória de contexto. Os resultados obtidos também são discutidos neste capítulo. Por fim, o Capítulo 6 apresentará as conclusões finais e trabalhos futuros.

2 CONTEXTUALIZAÇÃO

Arquiteturas reconfiguráveis são basicamente compostas de blocos lógicos que podem ser configurados, podendo variar a granularidade do bloco tanto como operações bit lógico quanto funções complexas. O caminho de dados também pode ser configurado, o que permite grande flexibilidade e adaptação na operação que se deseja desempenhar.

Arquiteturas reconfiguráveis vêm sendo propostas em trabalhos com propósito de aceleração de aplicações. Esses trabalhos demonstraram ganhos significativos em aceleração (GUPTA, 1993; GAJSKI, 1998; HENKEL, 1999; VENKATARAMANI, 2001). Alguns estudos como (STITT, 2002), além de acelerar a execução conseguem diminuir o consumo de energia, quando comparados ao sistema original.

2.1 Classificação das Arquiteturas Reconfiguráveis

Uma arquitetura reconfigurável é formada de uma *Reconfigurable Function Unit* (RFU) e uma unidade capaz de realizar a reconfiguração da RFU. Na literatura, ainda não existe um senso comum de classificação, embora a RFU seja caracterizada por alguns aspectos principais como acoplamento, granularidade, reconfigurabilidade e mecanismo de reconfiguração (COMPTON, 2002).

2.1.1 Acoplamento

O acoplamento define como a RFU é conectada ao processador principal e como a interface entre eles opera. Tal aspecto também inclui as questões de como os dados são transferidos e a forma de sincronização entre as partes.

Em um projeto de uma arquitetura reconfigurável a escolha do tipo de acoplamento entre a RFU e o Processador de Propósito Geral (*General Purpose Processor* - GPP) tem impacto direto na eficiência do sistema. De acordo com a implementação da RFU, pode-se classificá-la em fortemente ou fracamente acoplada ao processador.

Quando a RFU é implementada como uma unidade funcional dentro do processador essa é chamada de fortemente acoplada. Esta abordagem tem um alto desempenho, pois a comunicação ocorre dentro do processador. Outra forma de implementar uma RFU é como um coprocessador. Normalmente, a escolha desse tipo de acoplamento está ligada à área disponível de silício dentro do núcleo. Nesse caso, a RFU é interligada por barramento, obtendo desempenho inferior em relação à opção anterior devido ao custo de comunicação. Onechip (CARRILLO, 2001) e Nano (WIRTHLIN, 1994) são exemplos de arquiteturas fortemente acopladas.

No acoplamento fraco o custo de comunicação é ainda maior. A RFU anexada é um exemplo desse tipo de abordagem, ficando localizada no barramento que conecta a memória cache e a interface de entrada/saída. O custo de comunicação é alto, entretanto, é menor do que a abordagem de acoplamento independente. Essa última se comunica com o RFU por meio do barramento de entrada/saída- dentre todas as abordagens esta é a mais fracamente acoplada. Chimaera (HAUCK, 1997), GARP (HAUSER, 1997) e REMARC (MIYAMORI, 1998) são exemplos de arquiteturas fracamente acopladas.

2.1.2 Granularidade

A granularidade da unidade reconfigurável define o nível de manipulação de dados. Cada RFU pode ser implementada por meio de diferentes tipos e tamanhos de blocos funcionais. Por exemplo, pode-se formar uma RFU somente com somadores independentes de um bit, ou simplesmente, utilizar um somador de 32 bits como unidade básica. Esse tipo de escolha de projeto é denominado granularidade da RFU.

Em unidades de reconfiguração de granularidade fina, as menores partes (ou blocos) possíveis para reconfiguração normalmente são portas lógicas- abordagem muito eficiente para operações em nível de bits. Para operações paralelas de bits, como unidades aritméticas, a utilização de granularidade grossa é mais adequada. Nas RFUs de granularidade grossa as unidades funcionais são maiores, como unidades lógicas e aritméticas.

A granularidade escolhida para os blocos funcionais tem influência no número de bits necessários para reconfiguração. Uma RFU formada por somadores de um bit como unidade básica reflete em um elevado número de bits para realizar uma soma de 32 bits. Contudo, se for realizada a implementação de um somador de 32 bits como uma unidade funcional, abstrai-se a complexidade da configuração dos 32 somadores na reconfiguração da RFU, tornando o controle mais simples para a execução da mesma tarefa. Todavia, toda a lógica empregada na unidade do somador somente pode realizar esta operação específica. Caso outra função seja necessária deve-se acrescentar outra unidade funcional para este propósito. Essa é uma decisão de projeto e deve ser baseada nos requisitos e restrições do projeto como: conjunto de operações que devem ser executadas na RFU, área de armazenamento dos bits de reconfiguração, tempo de reconfiguração e desempenho.

Como exemplo de arquiteturas de grão fino é possível citar GARP (HAUSER, 1997) e Chimaera (HAUCK, 1997); já as arquiteturas REMARC (MIYAMORI, 1998) e Rapid (CRONQUIST, 1999) são classificadas como de grão grosso.

2.1.3 Reconfigurabilidade

A configuração da lógica programável pode suceder em momentos diferentes. Se ela ocorrer somente antes do início da execução – caso em que a unidade é denominada configurável. Por outro lado, a unidade é dita reconfigurável apenas quando é possível configurá-la depois da inicialização. A aplicação pode ser dividida em diferentes blocos (sequências definidas de instruções que podem ser executadas na RFU) e, de acordo com a execução, a RFU pode ser reconfigurada de acordo com a necessidade de cada bloco.

A reconfiguração da RFU é realizada de forma mais simples de ser implementada se a unidade configurável permanecer bloqueada em cada execução. Entretanto, se a unidade for reconfigurada durante a execução pode-se aumentar o desempenho, pois o tempo de reconfiguração diminui pelo fato de ocorrer em paralelo à execução. Para que

isso seja possível, a unidade reconfigurável deve ser dividida em segmentos que podem ser reconfigurados de forma independente um do outro. Esse processo de reconfiguração em partes é chamado de reconfiguração parcial (COMPTON, 2002).

2.1.4 Mecanismos de Reconfiguração

Os mecanismos de reconfiguração podem ocorrer em tempo de compilação ou em tempo de execução. Os mecanismos de reconfiguração que ocorrem em tempo de compilação extraem durante esta fase, partes dos programas que podem ser executados de forma mais eficiente na RFU. Porém, essas técnicas são dependentes de alguma ferramenta que realiza a análise do código fonte e o modifica para compatibilizar com a RFU - um exemplo dessa abordagem foi proposto em (HUTCHINGS, 1997; HUTCHINGS, 1999). A desvantagem desse método consiste no aumento do tempo de projeto, devido à inserção de mais uma etapa em seu fluxo. Outro prejuízo é o de não prover compatibilidade de software, sendo necessária a re-compilação dos programas para que esse faça uso da RFU.

Quando o mecanismo de reconfiguração é dinâmico as desvantagens acima citadas não ocorrem, pois em tempo de execução são detectadas partes do código da aplicação que podem ser executadas mais eficientemente na RFU. Como exemplo pode-se citar Lysecky (2006), trabalho pioneiro dessa abordagem. As vantagens providas por essa abordagem são a compatibilidade de software e consequentemente a diminuição do *time-to-market* do dispositivo.

2.2 Memória de contexto

Como explicitado anteriormente, em muitos trabalhos de arquiteturas reconfiguráveis o objetivo principal é otimizar as unidades reconfiguráveis, negligenciando o impacto que a memória de contexto desempenha no sistema reconfigurável.

Vários estudos que exploram técnicas de redução de energia ou aumento de desempenho da memória foram realizados para processadores superescalares, VLIW e processadores de múltiplos núcleos. Como exemplos de algumas técnicas vale mencionar o uso de associatividade, o tamanho e o tamanho da linha de memória cache programáveis (ZHANG, 2003); *Tag Encoding* (ZHANG, 2007); *Scratchpad Memory* (VERMA, 2004); e compressão de instruções (BENINI, 1999).

No entanto, nenhuma das opções acima estão relacionadas ou podem ser aplicadas diretamente a sistemas reconfiguráveis. Diferentemente de memórias convencionais de instruções/dados, a memória de contexto armazena configurações. Embora ambas requeiram tempos de acesso de alta velocidade, a memória de contexto apresenta uma diferença importante: o tamanho da palavra de memória pois, o número de bits de saída é várias ordens de grandeza maior que o tamanho de palavra de memórias convencionais, aumentando consideravelmente o consumo de energia. Caso utilizem-se memórias convencionais para implementar a memória de contexto, seria necessário realizar vários acessos consecutivos para carregar a configuração. Essa abordagem pode ser inviável devido ao tempo gasto para carregar a configuração, prejudicando um dos objetivos, como o desempenho.

Como explicitado anteriormente, a quantidade de lógica disponível na unidade reconfigurável impacta diretamente o tamanho da palavra de configuração, já que mais

bits são necessários para configurar as unidades. Por outro lado, o número de unidades reconfiguráveis é um dos principais fatores que determinam o desempenho: quanto maior o número de unidades, maior é o potencial de ganho. Arquiteturas de granularidade fina que trabalham no nível de bit, tais como aquelas baseadas em FPGAs (*Field Programmable Gate Array*), exigem mais bits de configuração se comparados com os sistemas de granularidade grossa.

No entanto, mesmo em arquiteturas de granularidade grossa o número de bits de um único contexto é muito significativo. Na próxima seção, onde são apresentados exemplos de arquiteturas reconfiguráveis, será ressaltada a grande quantidade de bits necessários para configuração do conjunto de unidades reconfiguráveis das arquiteturas.

Os primeiros estudos sobre a importância da memória de contexto em sistemas reconfiguráveis foi apresentado em (KIN, 2006). Os autores - ao utilizar como estudo de caso a arquitetura MorphoSys - mostram que a memória de contexto gasta aproximadamente 43% do consumo total de energia do sistema. Em (KIN, 2007) os pesquisadores, utilizando o mesmo estudo de caso, apresentam uma técnica de compressão para reduzir o consumo de energia.

2.3 Virtualização de Hardware

Inicialmente o conceito de hardware virtual foi criado em analogia à memória virtual. Nessas memórias as páginas podem ser trocadas entre a memória principal e o disco rígido pelo processo chamado de paginação. Esse mecanismo permite que aplicações possam endereçar uma memória que fisicamente não existe; da mesma forma, a virtualização de hardware em um sistema de computação reconfigurável permite que aplicações utilizem mais hardware que fisicamente existe. Esse mecanismo é realizado por meio do processo de reconfiguração, modificando a funcionalidade do hardware de acordo com a necessidade da aplicação. Atualmente a virtualização de hardware tem um conceito mais amplo, que vem sendo usado para descrever técnicas de mapeamento e arquiteturas. Esses processos permitem certo grau de independência entre a aplicação mapeada e as capacidades reais da arquitetura alvo.

A virtualização de hardware pode ser uma técnica interessante em sistemas embarcados, particularmente em sistemas que utilizam arquiteturas reconfiguráveis, onde aplicações devem ser mapeadas e executadas em hardware com limitações de área. Em arquiteturas reconfiguráveis as operações são organizadas espacialmente, ao passo que em processadores as operações são estruturadas principalmente no tempo (e.g. cada instrução é executada em um ciclo do processador). Enquanto os processadores podem executar aplicações tão grandes quanto a capacidade da memória, arquiteturas reconfiguráveis encontram problemas quando o mapeamento da aplicação excede o tamanho da matriz reconfigurável. Com o conjunto de técnicas de virtualização de hardware é possível superar essa limitação explorando a reconfigurabilidade dos dispositivos.

Em (PLESSL, 2004) o autor classifica as abordagens de virtualização de hardware em três categorias: Particionamento temporal (*temporal partitioning*), Execução virtualizada (*virtualized execution*) e Máquina virtual (*virtual machine*).

2.3.1 Particionamento temporal

Essa abordagem de virtualização é utilizada para permitir que uma aplicação de tamanho arbitrário seja mapeada em um dispositivo reconfigurável com capacidade de

hardware insuficiente. Para que esse processo seja realizado, a aplicação é dividida em partes menores que serão possíveis de serem mapeadas no dispositivo reconfigurável e posteriormente executadas sequencialmente para obter o resultado final.

O particionamento temporal foi a primeira abordagem estudada, pois era uma necessidade devido aos blocos lógicos limitados dos dispositivos reconfiguráveis. Apesar dos dispositivos atuais possuírem grande quantidade lógica reconfigurável, essa abordagem ainda é válida porque permite a redução da área e, conseqüentemente, do custo.

As arquiteturas DPGA (DEHON, 1996), Chameleon (XINAN, 2000), DRLE (FUJII, 1999), Time-multiplexed FPGA (TRIMBERGER, 1997) e Zippy (ENZLER, 2003) são alguns exemplos que utilizam essa abordagem de virtualização.

2.3.2 Execução virtualizada

Esse método de enfoque tem o objetivo de alcançar certo nível de independência de dispositivo dentro de uma família de dispositivos. Para atingir esse objetivo, a aplicação é especificada em um modelo de programação que define algumas unidades atômicas de computação, comumente chamadas de páginas de hardware.

Nesse modelo, uma aplicação é especificada como um conjunto de tarefas e suas respectivas interações. Entende-se por tarefa um conjunto de operações organizadas dentro de uma página de hardware, portanto, uma aplicação nada mais é que um conjunto de páginas de hardware que interagem entre si. A execução da arquitetura é definida para toda família do dispositivo, ou seja, todos os dispositivos da família suportam a abstração deste modelo de programação definido, composto das páginas de hardware e canais de comunicação que permitem interações entre as mesmas. Dessa forma, pode haver uma quantidade diferente de recursos em cada dispositivo, como número de páginas, número de execuções simultâneas ou ainda número de tarefas que podem ser armazenadas no dispositivo.

Uma vez que todas as implementações da arquitetura suportam o mesmo modelo de programação, uma aplicação pode ser executada em qualquer membro da família de dispositivos sem a necessidade de recompilação, atingindo, assim, a independência do dispositivo. Essa independência é comparável à família de processadores que mantém compatibilidade definindo o mesmo conjunto de instruções para diferentes implementações. Entretanto, a execução virtualizada requer um sistema que de alguma forma resolva em tempo de execução os conflitos de recursos e o escalonamento das tarefas de forma adequada.

Uma grande vantagem dessa abordagem é a possibilidade de exploração do espaço de projeto. A independência dos dispositivos, ou seja, dispositivos com diferentes recursos (número de UFs, por exemplo), permite que o projetista explore a relação entre o custo e o desempenho, maximizando os ganhos para cada projeto específico. Além disso, essa compatibilidade permite explorar os futuros avanços tecnológicos que resultam em dispositivos maiores e mais rápidos.

Alguns exemplos de arquiteturas reconfiguráveis que possuem execução virtualizada são PipeRench (GOLDSTEIN, 2000), Score (CASPI, 2000) e WASMI (FUJII, 1999). Essa abordagem de virtualização é a que mais se assemelha à que será proposta neste trabalho, portanto, para finalidade de comparação, as arquiteturas PipeRench e Score serão mais discutidas na seção de exemplos de arquiteturas.

2.3.3 Máquina virtual

Nesse método o objetivo é alcançar um alto nível de independência do dispositivo de acordo com os conceitos de máquinas virtuais e portabilidade de código. Para isso, uma aplicação é especificada para uma arquitetura de execução abstrata e uma máquina de hardware virtual é responsável pelo remapeamento da especificação da arquitetura abstrata para a arquitetura de execução concreta. Exemplos de trabalhos relacionados a máquinas virtuais podem ser encontrados em (HA, 2000) e (HA, 2002).

2.4 ARISE

A ferramenta ARISE (do inglês *Automatic Resources Investigation System based on application Execution*) explora de forma automática, através de grafos de dependência de dados e instruções (GDD), a execução de uma aplicação e fornece o número de recursos necessários para acelerá-la.

No trabalho (RUTZIG, 2008) os autores utilizam a arquitetura apresentada no Capítulo 3 para demonstrar a redução de área obtida com o uso da ferramenta. A partir do modelo original (Figura 3.2), que possui o formato retangular, a ferramenta constrói um modelo otimizado removendo unidades funcionais que não são utilizadas durante a execução do conjunto de aplicações, assim, reduzindo a área sem perda de desempenho. Também é possível explicitar a restrição máxima de perda de desempenho que o dispositivo final pode apresentar em relação ao sistema original para que a ferramenta possa remover as unidades pouco utilizadas e com isso obter maior redução de área.

A Figura 2.1 mostra um exemplo do funcionamento da ferramenta ARISE. Na Figura 2.1(a) é apresentada como é realizada a alocação da sequência de instruções na matriz reconfigurável sem otimização, de acordo com o GDD gerado. Como se pode observar, uma grande quantidade de unidades funcionais não é utilizada.

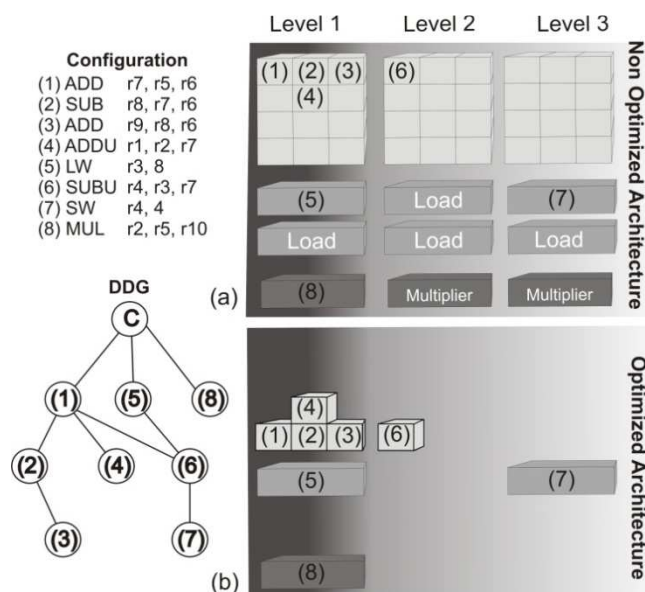


Figura 2.1: Otimização da matriz reconfigurável através da ferramenta ARISE.

O resultado final é um modelo otimizado, como apresentado na Figura 2.1(b) onde cada linha da matriz reconfigurável possui um diferente número de unidades funcionais, o qual varia de acordo com as aplicações analisadas. Portanto, a redução de área é

dependente das aplicações que serão analisadas e do tamanho da matriz reconfigurável (número de unidades funcionais disponíveis).

2.5 Exemplos de Arquiteturas Reconfiguráveis Propostas

O sistema GARP (HAUSER, 1997) funciona como um coprocessador acoplado a um processador que estende o conjunto de instruções MIP-II e utiliza FPGA como lógica reconfigurável. Portanto, é classificado como uma arquitetura fracamente acoplada de granularidade fina. A comunicação entre o processador e a RFU é realizada por meio de instruções dedicadas.

A arquitetura reconfigurável GARP é composta por entidades chamadas de blocos. A arquitetura fixa o número de colunas de blocos em 24, enquanto o número de linhas de blocos é definido para cada implementação, tendo no mínimo 32 linhas. Uma linha é constituída por um bloco de controle e o restante por blocos lógicos, que correspondem aproximadamente a CLBs da série Xilinx (MAHESWARAN, 1994).

Cada bloco nessa arquitetura necessita de exatamente 64 bits (8 bytes) de configuração para especificar a fonte da entrada dos dados, a função de cada bloco e a saída de dados. Logo, uma configuração de 32 blocos requer aproximadamente 6KB, assumindo que o caminho de dados para a memória externa seja de 128 bits. Assim, para carregar uma configuração completa seriam necessários 384 acessos à memória. A arquitetura suporta reconfiguração parcial, entretanto, a configuração mínima é de uma linha, lembrando que mais de uma configuração não pode estar ativa simultaneamente.

O sistema reconfigurável CHESS (MARSHALL, 1999) é uma arquitetura de granularidade grossa desenvolvida pelos laboratórios da HP. A unidade reconfigurável é chamada RAA (*Reconfigurable Arithmetic Array*) e tem o propósito de ser utilizada com aplicações multimídia. A RAA é composta de ULAs de 4 bits, em que cada unidade possui 16 funções diferentes - se necessário, é possível agrupar várias ULAs para operar palavras maiores de 4 bits. As ULAs são conectadas umas às outras por meio de um barramento de 4 bits. Os componentes responsáveis pela interconexão são caixas de comutação (*switchboxes*). Cada ULA é adjacente a quatro caixas de comutação e vice-versa. Dessa forma, uma ULA tem um barramento de entrada e de saída em todos os quatro lados, além de ser capaz de se comunicar com qualquer uma das oito ULAs que a circundam.

A RAA é composta de 512 ULAs. Como cada ULA necessita de 100 bits de configuração, o número estimado de bytes para uma única configuração é de aproximadamente 6400, incluindo o mecanismo de interconexão.

PipeRench é uma arquitetura de granularidade grossa, fracamente acoplada e focada no processamento em *pipeline* de *streams* de dados. O princípio básico do PipeRench (GOLDSTEIN, 1999) é chamado *pipelined reconfiguration*. Isso significa que um determinado *kernel* é quebrado em pedaços, e esses pedaços podem ser reconfigurados e executados sob demanda, de acordo com as necessidades do *kernel*. O conjunto de estágios de *pipeline* são chamados de *stripes*. Como se pode observar na Figura 2.2, cada *stripe* tem uma interconexão (*Interconnection Network*) e um conjunto de elementos de processamento (*Processing Elements* – PEs). A funcionalidade dos PEs é especificada por 42 bits de configuração, o que significa que cada *stripe* necessita de 672 bits de configuração. Os dados de configuração são armazenados em 22 memórias SRAMs, cada uma com 246 palavras de 32 bits. Essa arquitetura tem 16 *stripes*

implementados em hardware, mas ao utilizar a técnica de virtualização suporta 256 *stripes*. Para armazenar uma configuração que utiliza 256 *stripes* são necessários 21504 bytes de configuração.

Quando a virtualização é utilizada a potência aumenta significativamente, principalmente devido à contínua operação com a memória de reconfiguração. Esse aumento pode ser observado em (GOLDSTEIN, 1999), que descreve a implementação do PipeRench na tecnologia de processo 0.18 nm e demonstra que quando a virtualização é requerida a dissipação de potência aumenta aproximadamente 30%.

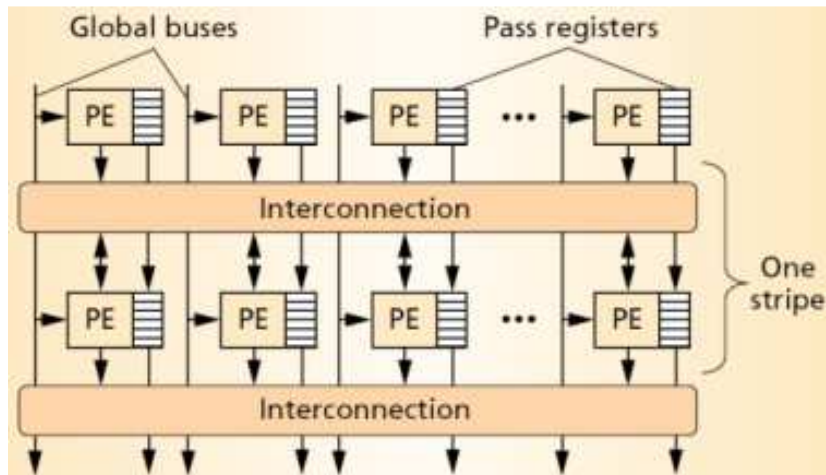


Figura 2.2: Arquitetura do PipeRench (GOLDSTEIN, 1999).

Essa arquitetura realiza a virtualização dos estágios através da reconfiguração do pipeline. A Figura 2.3 mostra a execução de uma aplicação que utiliza 5 *stripes* virtuais (Figura 2.3(a)) em 3 *stripes* físicos (Figura 2.3(b)) da arquitetura. Neste exemplo, cada *stripe* é configurado em 1 ciclo e executado nos próximos 2 ciclos - é possível observar que no quarto ciclo o quarto estágio virtual é configurado. Como o primeiro estágio físico já havia terminado os dois ciclos de execução, está livre para que seja mapeada uma próxima configuração, portanto, é nesse estágio físico que o quarto estágio virtual é mapeado. Esse mecanismo de mapeamento de estágios ocorre sequencialmente para os demais ciclos, sendo preciso somente três estágios físicos para mapear quantos estágios virtuais forem necessários.

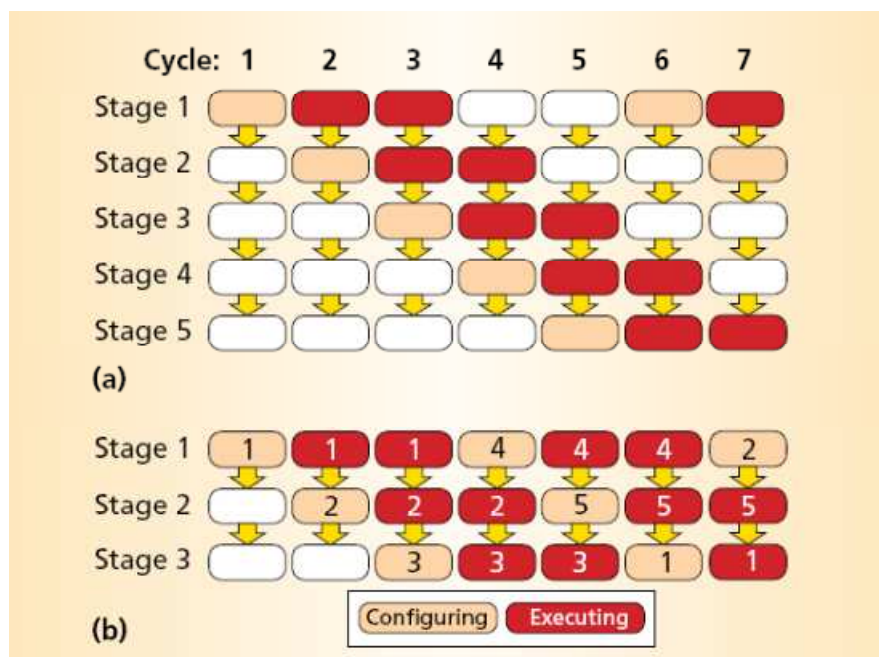


Figura 2.3: Virtualização de cinco estágios do *pipeline* em três estágios físicos (GOLDSTEIN, 1999).

Durante a execução, as configurações são utilizadas para reconfigurar dinamicamente a arquitetura de acordo com a necessidade. As configurações são geradas por meio de ferramentas desenvolvidas para essa arquitetura, que exploram os grafos de fluxo de dados. Essa etapa é realizada de forma estática, durante a fase de compilação; tal abordagem de virtualização é classificada como execução virtualizada, segundo (PLESSL, 2004).

Outra arquitetura que possui um modelo de execução virtualizada é a arquitetura Score (CASPI, 2000) - projetada para executar um conjunto de aplicações de *streaming*. Uma aplicação é definida como um grafo de nodos que são conectados por uma fila FIFO de tamanho ilimitado, onde cada nodo representa um operador. A organização dos operadores é definida pelo grafo de dependências de dados. A função dos operadores é especificada pela linguagem TDF, basicamente uma descrição RTL com uma sintaxe especial (*C-like*) para o tratamento do fluxo de dados de entrada e de saída do operador. A memória usada pelos operadores é alocada em páginas de memória de tamanho fixo. A arquitetura híbrida é composta de um processador e uma matriz reconfigurável. A matriz reconfigurável é composta de páginas equivalentes e de computação independente, blocos de memória configurável e interconexões com buffer. A avaliação é realizada por meio de simulação.

O sistema de *runtime* é executado no processador, consistindo de um mecanismo de instanciação, que interpreta o grafo de computação e informa o escalonador quais as tarefas que devem ser alocadas, e de um mecanismo de escalonamento, que gerencia a alocação de recursos, o mapeamento dos operadores para execução das páginas e o roteamento. O sistema de execução também implementa um compartilhamento de tempo da computação das páginas entre os operadores.

Nas arquiteturas apresentadas observa-se que a quantidade de bytes de configuração é grande, conseqüentemente necessitando uma memória de contexto com elevada capacidade de armazenamento e desempenho. Porém, a memória de contexto é pouco

explorada, negligenciando sua importância para o sistema reconfigurável como um todo. No Capítulo 5 este tema será retomado e será demonstrado o impacto energético no sistema causado pela memória de contexto. Além disso, serão apresentadas duas técnicas para minimizar o consumo de energia.

3 DESCRIÇÃO DA ARQUITETURA RECONFIGURÁVEL

A arquitetura reconfigurável que será descrita neste trabalho foi desenvolvida pelo grupo de Sistemas Embarcados da UFRGS e é chamada de *Dynamic Instruction Merging* (DIM). Como citado anteriormente, o projeto obteve os primeiros resultados em 2005 com o trabalho (BECK, 2005), em que foi apresentada a técnica de tradução binária. Posteriormente, em (BECK, 2006), foi demonstrada uma variação dessa técnica que utilizava um modelo de um processador RISC que executa o conjunto de instruções PISA. Para este trabalho a segunda abordagem foi adotada ao empregar o processador MIPS R3000 como alvo da implementação do sistema reconfigurável. Em resumo, esse sistema reconfigurável se classifica como dinâmico, de grão grosso e fortemente acoplado.

O diagrama de blocos do sistema reconfigurável é apresentado na Figura 3.1. O sistema é formado pela matriz reconfigurável (Figura 3.1(1)), pelo processador MIPS 3000 - representado pelos estágios IF (*Instruction Fetch*), ID (*Instruction Decode*), EX (*Execute*), DF (*Data Fetch*) e WB (*Write Back*) (Figura 3.1(2)), pelo Tradutor Binário – constituído pelos estágios DEC (Decodificação das instruções), VE (Verificação de Estouro na configuração), AT (Atualização das Tabelas) e EC (Escrita na memória de contexto) (Figura 3.1(3)), pela memória de Contexto e as Caches de dados e instruções (Figura 3.1(4)).

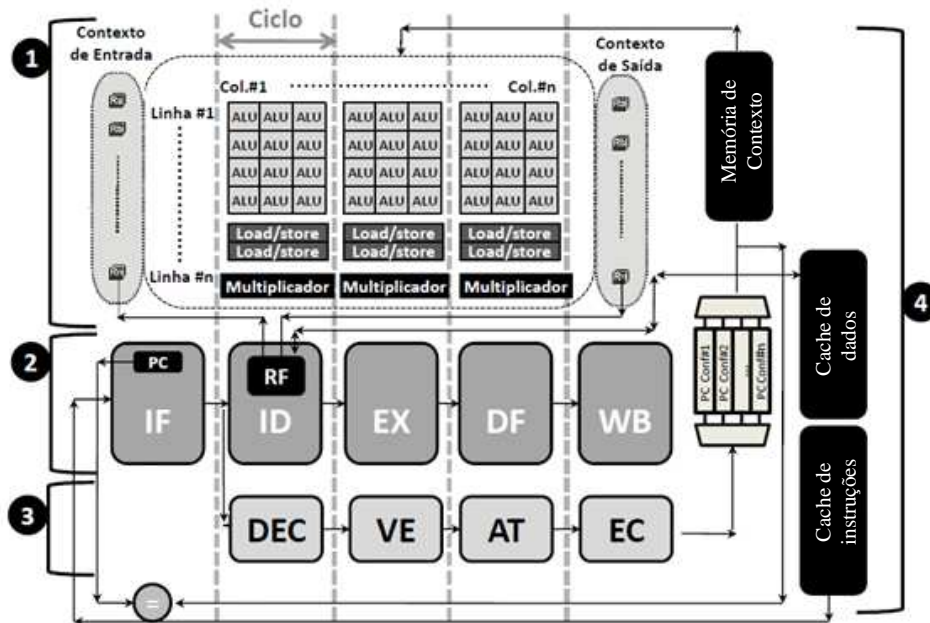


Figura 3.1: Organização do sistema DIM. (1) Matriz reconfigurável; (2) processador; (3) TB (4) Memória de Contexto, juntamente com as caches de Instrução e de Dados do processador.

3.1 Estrutura

A Figura 3.2 apresenta a estrutura da arquitetura reconfigurável. A matriz reconfigurável é organizada em duas dimensões, permitindo a execução de maneira paralela e sequencial. Na paralela, as UFs que estão dispostas horizontalmente na estrutura iniciam sua execução no mesmo instante. Já no modo sequencial, instruções alocadas em diferentes linhas iniciam sua execução em diferentes instantes, de acordo com suas dependências de dados.

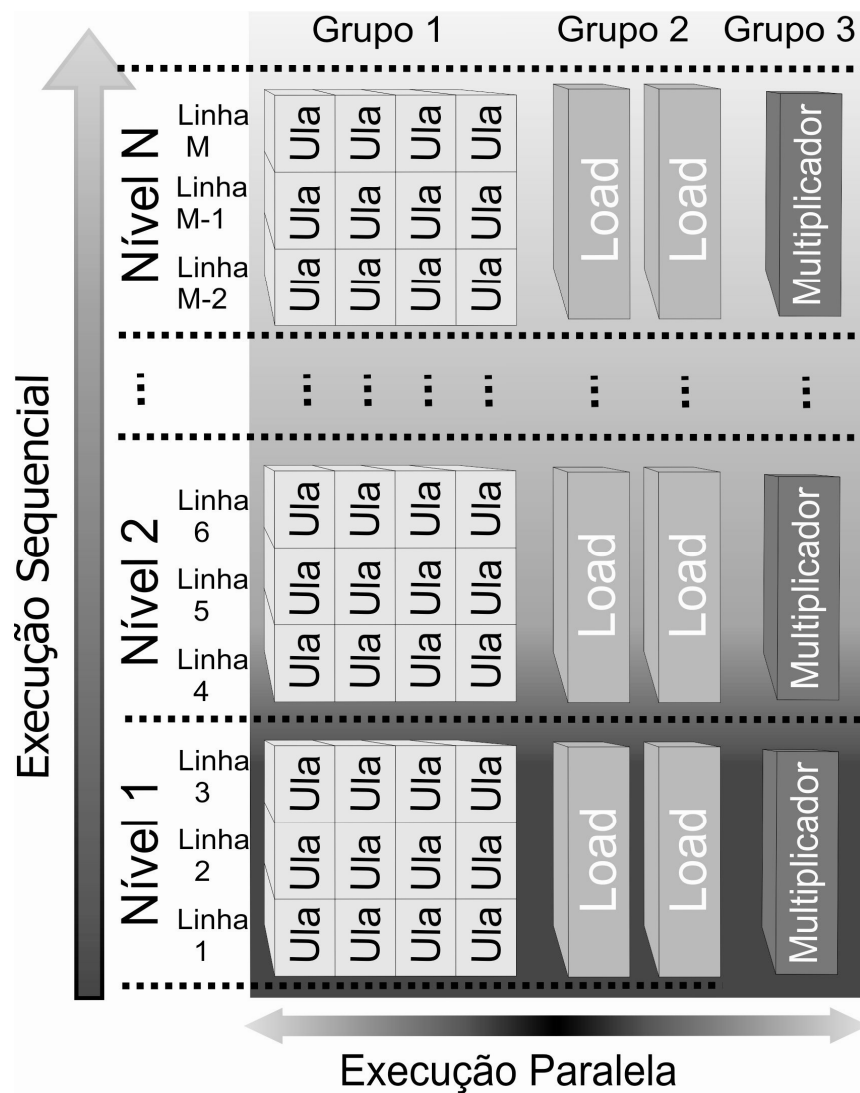


Figura 3.2: Visão geral da matriz reconfigurável.

As UFs são divididas em 3 grupos, conforme a Tabela 3.1. Um nível da arquitetura reconfigurável corresponde a três ULAs em sequência, uma unidade de *load/store* ou uma unidade de multiplicação. A unidade funcional é o grão mínimo de reconfiguração, motivo pelo qual é classificada como arquitetura de grão grosso.

Cada nível é composto de três linhas e possui atraso equivalente a um ciclo do processador. Assim, até 3 operações lógicas e aritméticas com dependência de dados podem ser executadas em um ciclo equivalente do processador. É importante salientar que não existem barreiras temporais entre os níveis, ou seja, a execução dentro da estrutura reconfigurável é realizada de forma combinacional.

Devido à regularidade da arquitetura, pode-se facilmente parametrizar o número de UFs em cada grupo, dessa forma modificando o número de operações que podem ser executadas paralelamente ou ainda parametrizar o número de níveis da arquitetura para que mais operações possam ser alocadas numa mesma configuração (de forma sequencial). Essa facilidade de parametrização permite otimizar a arquitetura de forma a atender às restrições de área, potência e desempenho impostas de cada projeto.

Tabela 3.1: Grupos de unidades funcionais e suas operações.

<i>Unidades Funcionais</i>	<i>Operações executadas</i>
Grupo 1: Unidade Lógica e Aritmética	Soma, Subtração, Deslocamento, Comparação, Lógica de Bit
Grupo 2: <i>Load/Store</i>	Leitura/Gravação de dado na memória
Grupo 3: Multiplicador	Multiplicação

3.2 Interconexões

As unidades funcionais que compõem a estrutura da arquitetura reconfigurável são interligadas basicamente por multiplexadores, como se pode observar na Figura 3.3. A Figura 3.3(a) detalha os multiplexadores de entrada. Esses realizam as conexões entre o barramento de contexto e as entradas das unidades funcionais. Na Figura 3.3(b) estão representados os multiplexadores de saída, que conectam a saída das unidades funcionais a uma linha do barramento de contexto.

Para cada linha de contexto existe um multiplexador com o número de entradas igual ao número de unidades funcionais da linha da matriz reconfigurável. Em termos de área ocupada, em média as interconexões correspondem a cerca de 50% da área total.

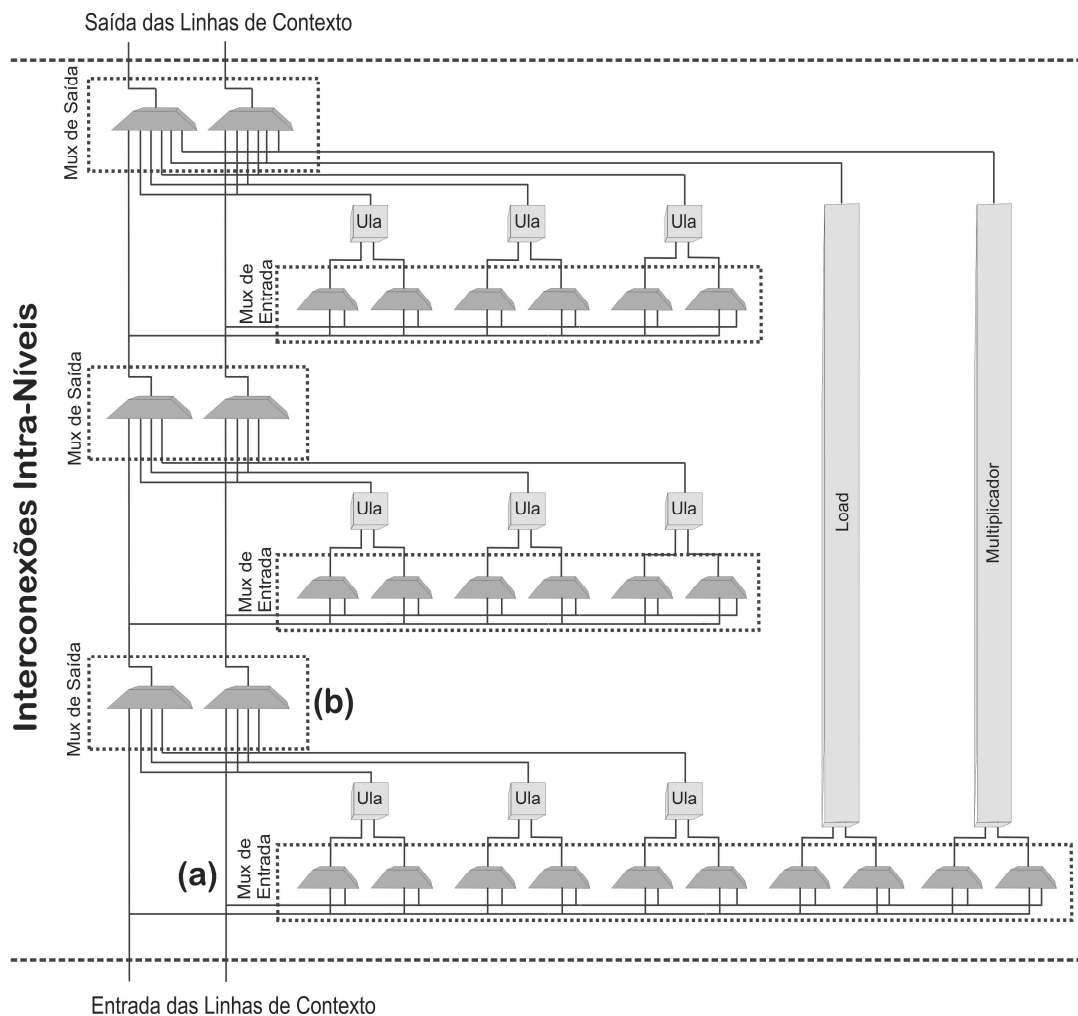


Figura 3.3: Interconexões entre as unidades funcionais.

Na Figura 3.4 é ilustrado um exemplo do caminho de dados para as seguintes instruções:

1. $R_b = R_b \times R_a$
2. $R_a = R_a + R_b$

Na execução, os valores dos registradores usados são buscados do contexto de entrada, as UFs operam sobre esses dados e escrevem os resultados parciais no contexto de saída. Depois da execução completa, os valores do barramento de contexto de saída são gravados nos seus registradores de destino.

Os bits de reconfiguração são responsáveis pelo roteamento dos dados entre o barramento de contexto e as UFs, além de definirem quais operações as UFs executarão e qual registrador será mapeado em cada linha de contexto.

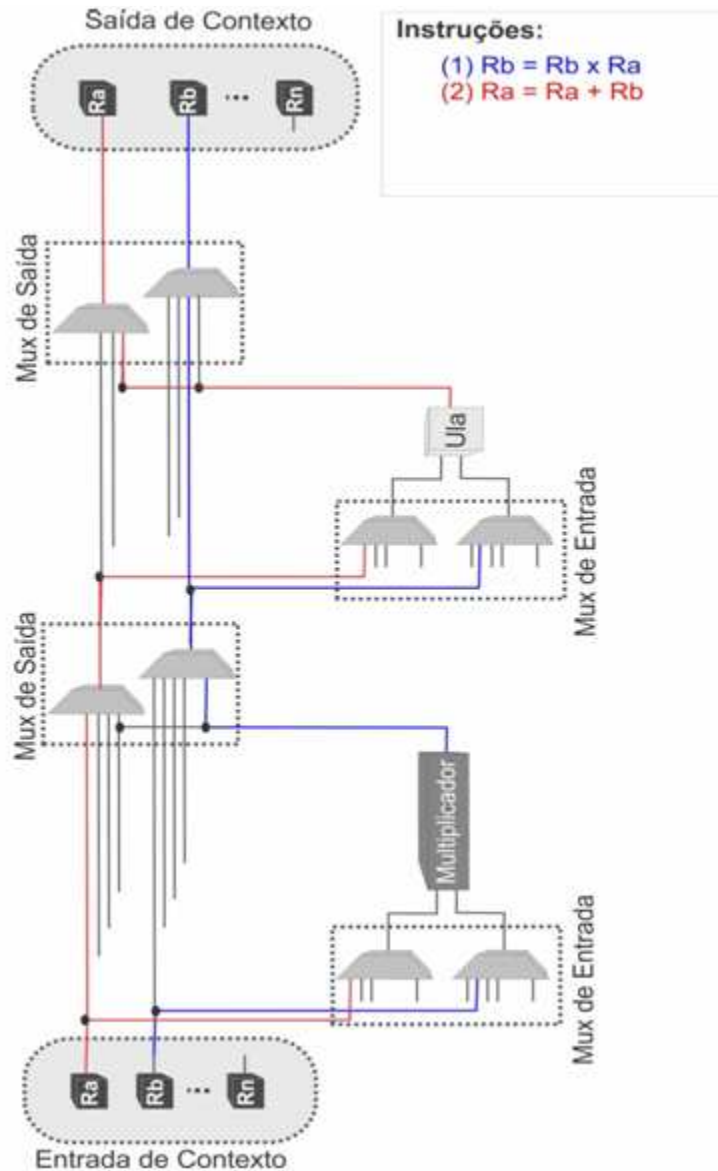


Figura 3.4: Exemplo de uma mapeamento de instruções.

3.3 Tradução binária

O conceito de tradução binária (ALTMAN, 2000) (ALTMAN, 2001) é muito amplo e pode ser aplicado em vários níveis de abstração. Basicamente, existe um sistema, que pode ser implementado em hardware ou software, responsável por analisar o programa em execução. Com isso, algum tipo de transformação é feito no código com o objetivo de manter a compatibilidade com o software (o reaproveitamento de código legado, sem necessidade de recompilação) e de fornecer meios para melhorar o desempenho.

O mecanismo de tradução binária (TB) utilizado nesta arquitetura é implementado completamente em hardware e opera em paralelo ao processador (BECK, 2006a). Em tempo de execução, a unidade de TB detecta a sequência de instruções que podem ser executadas na arquitetura reconfigurável. Tal sequência é traduzida em uma configuração por meio do mecanismo de TB, e armazenada na memória de contexto. Essas sequências são indexadas pelo registrador denominado Contador de Programa (*Program Counter - PC*), que é muito similar ao que as ferramentas de tradução estática

utilizam a fim de encontrar trechos de códigos (*kernels*) para transformá-la em instruções reconfiguráveis. Porém, muito mais simples se comparado aos processos das ferramentas, podendo assim ser implementadas em hardware e processadas em tempo de execução.

Resumindo o algoritmo, para cada instrução recebida, a primeira tarefa é a verificação das dependências de dados. Os operandos fonte são comparados com um mapa de bits de registradores de destino de cada linha. Se a linha atual e todas as linhas acima não têm um registrador alvo igual a um dos operandos fonte da instrução atual, essa instrução pode ser alocada na mesma linha, na coluna mais à esquerda possível, dependendo do grupo de unidades funcionais a que esta pertence. Quando essa instrução é colocada na mesma linha, o mapa de bits do registrador alvo é atualizado. Dessa forma, para cada instrução é necessário analisar apenas uma linha do mapa de bits. Para cada linha, há também a informação sobre quais registradores devem ser atualizados no banco de registradores do processador e quais devem ser gravados na memória. Assim, é possível atualizar os registradores (*Write Back*) que não serão utilizados novamente em paralelo com a execução de outras operações.

Na Figura 3.5 é ilustrada a alocação de uma sequência de instruções utilizando o mecanismo de tradução binária. Pode-se observar que a segunda instrução possui uma dependência verdadeira com a primeira, portanto, ela deve ser alocada em uma linha superior. Na terceira instrução ocorre a mesma dependência em relação à segunda. A quarta instrução possui dependência verdadeira somente com a primeira, logo, ela pode ser alocada na segunda linha. A quinta instrução por não apresentar dependências pode ser alocada na primeira unidade funcional livre.

A sexta instrução possui dependência verdadeira com a quinta instrução e, como a unidade funcional dessa instrução leva o tempo de um nível para ser executada, a instrução seis deve ser alocada no próximo nível. A sétima instrução é dependente do resultado da sexta e deve ser alocada somente após o término da mesma. Como a unidade não pode começar a operar no meio de um nível, a sétima instrução deve ser alocada no nível seguinte. A oitava instrução não possui dependência e pode ser alocada na primeira unidade funcional livre. Por fim, a nona instrução não é suportada pela arquitetura; a alocação é finalizada e a configuração é armazenada na memória de contexto.

No exemplo de trecho de código mostrado na Figura 3.5, as instruções de 1 a 8 seriam executadas em oito ciclos do processador. Porém, se executada na arquitetura reconfigurável, essa sequência será processada em três ciclos de relógio equivalentes do processador. Se desprezarmos os outros tempos envolvidos, como de busca e reconfiguração, o potencial de aceleração seria de aproximadamente 2,67 quando comparado ao desempenho do processador base. Mais detalhes podem ser vistos em (BECK, 2006a).

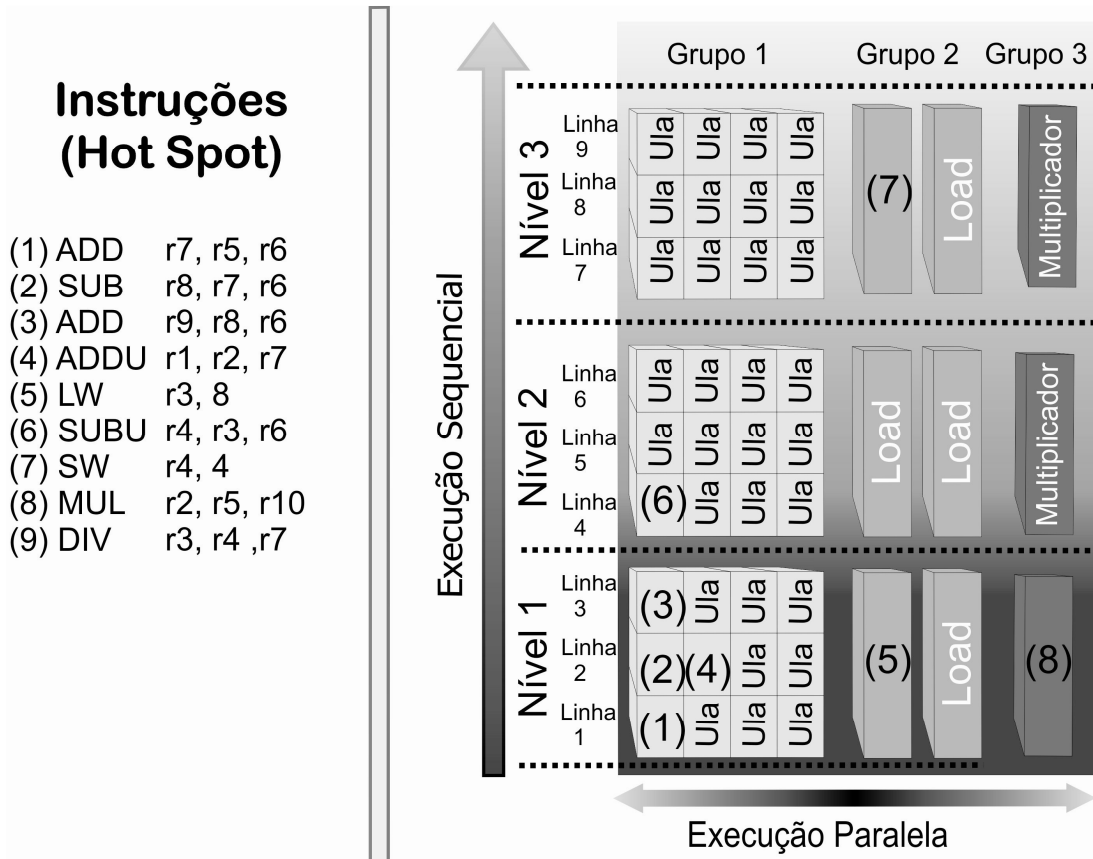


Figura 3.5: Alocação das unidades funcionais para determinada sequência de instruções.

Durante a execução, o PC da instrução atual é comparado com os salvos em uma tabela, a fim de verificar se existe uma configuração com o mesmo valor do PC atual. Se uma configuração é encontrada, ela é obtida a partir da memória de contexto. Outros sistemas reconfiguráveis trabalham de uma forma muito semelhante, contudo, em vez de ter as configurações indexadas pelo PC em uma tabela, há no código instruções que indicam onde se encontram as configurações na memória de contexto.

O restante do processo funciona da seguinte maneira: inicialmente, os valores do contexto de entrada são obtidos a partir do banco de registradores, enquanto os bits de configuração são adquiridos a partir da memória de contexto. Logo, essa configuração é executada levando um determinado número de ciclos de relógio do processador equivalente. Finalmente, os resultados são escritos de volta no banco de registradores do processador, o PC é atualizado e o processador continua normalmente.

3.4 Memória de Contexto

A memória de contexto é umas maiores responsáveis pela flexibilidade provida pelos sistemas reconfiguráveis, já que armazena as configurações – essas, também conhecidas como contextos, são conjuntos de bits de controle que definem a funcionalidade do circuito reconfigurável.

A memória de contexto é formada por duas tabelas, como representado na Figura 3.6. A primeira tabela (Figura 3.6(a)) armazena os PCs das configurações geradas e o ponteiro para o conjunto de bits de configuração. Cada posição desta tabela é comparada com o PC atual do processador para verificar se existe alguma configuração

desenvolvimento de novas técnicas para acessar os bits de configuração na memória de contexto.

3.5.1 Ambiente de Simulação

Em nosso estudo utilizaremos a ferramenta ARISE (RUTZIG, 2008), que é um simulador da arquitetura utilizada como estudo de caso. Essa ferramenta é modelada em *SystemC*, e o sistema modelado é composto do processador MIPS R3000, mecanismo de tradução binária e a matriz reconfigurável. A partir desse modelo é possível extrair resultados de desempenho, número de requisições à memória de reconfiguração e número de níveis de cada configuração utilizada. Uma versão na linguagem de descrição de hardware em VHDL foi modelada para obter os valores de energia.

A ferramenta Cacti 6.0 (MURALIMANO HAR, 2007) foi utilizada para modelar diferentes arranjos da memória de contexto e extrair o consumo de potência, energia de acesso, área e tempo de acesso. Os experimentos foram realizados utilizando a tecnologia CMOS 90 nm.

O sistema foi avaliado com o *Mibench Benchmark Suite* (GUTHAUS, 2001). Esse *benchmark* foi escolhido porque contém uma grande faixa de aplicações com diferentes comportamentos quando comparado com outros conjuntos de *benchmarks*, cobrindo tanto aplicações orientadas a controle (*controlflow*) como orientadas a dados (*dataflow*), encontradas em modernos telefones portáteis, por exemplo. Nós utilizaremos todas as aplicações do *benchmark* que não contém computações de ponto flutuante representativos.

A Figura 3.7 apresenta a relação instruções/salto das aplicações que serão avaliadas neste trabalho. As aplicações com a menor relação instruções/salto são consideradas *controlflow*, enquanto as aplicações com maior relação instruções/salto são classificadas como *dataflow*.

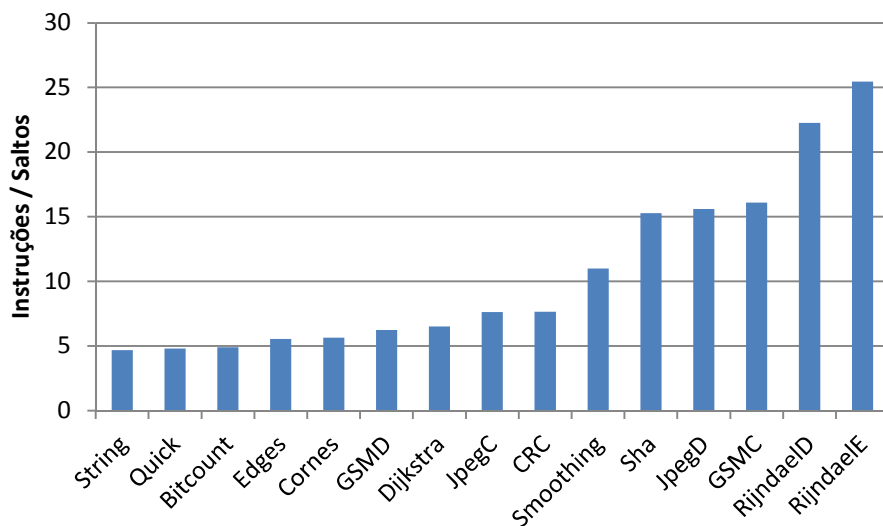


Figura 3.7: Relação de instruções/salto das aplicações avaliadas no trabalho.

3.5.2 Desempenho

Para avaliar os resultados foram consideradas quatro configurações da matriz reconfigurável. Na Tabela 3.2 são apresentadas as configurações da matriz reconfigurável analisadas. Cada configuração se difere basicamente pelo número de

unidades funcionais, variando o número de unidades funcionais em cada nível e o número de níveis da matriz reconfigurável.

A Configuração 1 é a menor delas, com somente oito níveis, um multiplicador, duas unidades de *Load/Store* e oito ULAs por nível. A Configuração 2 tem duas vezes mais níveis que a configuração anterior e seis *Load/Store* por nível. A Configuração 3 por sua vez, tem duas vezes o número de níveis da Configuração 2 e doze ULAs por nível. As Configurações 3 e 4 possuem o mesmo número de UFs por nível, porém a Configuração 4 possui 64 níveis, o dobro da Configuração 3.

Nesta tabela é mostrada também o número de bytes de configuração necessários para armazenar uma configuração. Quanto maior a configuração, mais bytes são necessários serem armazenados.

Tabela 3.2: Diferentes configurações da matriz reconfigurável analisadas.

	Conf. 1	Conf. 2	Conf. 3	Conf. 4
# Níveis totais	8	16	32	64
# Colunas totais	24	48	96	192
# ULA / nível	8	8	12	12
# Multiplicadores / nível	1	2	2	2
# Ld St / nível	2	6	6	6
# Bytes de controle	124	142	165	165
# Bytes de configuração / nível	87	106	128	128
# Bytes de configuração totais	820	1.838	4.261	8.357

A Figura 3.8 apresenta a aceleração média das quatro configurações para cada aplicação variando o número de entradas da memória de contexto. As Figuras 3.9, 3.10, 3.11 e 3.12 mostram a aceleração de cada aplicação analisada para as Configurações 1, 2, 3 e 4 respectivamente. A aceleração apresentada tem como base a execução das aplicações pelo sistema reconfigurável em relação ao processador MIPS R3000.

É possível verificar nos gráficos das Figuras 3.8 a 3.12 que o aumento de entradas da memória de reconfiguração leva a um aumento da aceleração devido principalmente ao fato que a probabilidade de encontrar uma sequência de código anteriormente traduzida e indexada pelo PC é alta. Com isso, mais sequências de códigos serão executadas na matriz reconfigurável.

Matrizes reconfiguráveis diferentes têm distintos níveis de desempenho, visto que matrizes com mais unidades funcionais por linha podem explorar melhor o paralelismo. O número de níveis igualmente pode influenciar, já que mais instruções podem ser traduzidas em lógica combinacional, tornando a execução mais rápida. Isso pode ser observado nas Configurações 3 e 4, que têm o mesmo número de UFs por nível, porém número de níveis diferentes.

É possível observar nas Figuras 3.9, 3.10, 3.11 e 3.12 algumas características de cada aplicação com relação ao aumento de configurações que podem ser armazenadas na memória de contexto. Na aplicação *Sha*, a aceleração praticamente satura quando a memória possui dezesseis posições, ou seja, essa aplicação possui um pequeno número de *kernels*. Essa quantidade de *kernels* provavelmente é maior que oito, pois é possível observar o grande aumento da aceleração de oito para 16 posições de memória. Essa saturação na aceleração também pode ser observada em outras aplicações como a *String* (com 128 posições), *RjindaelE* e *RjindaelD* (com 32 posições).

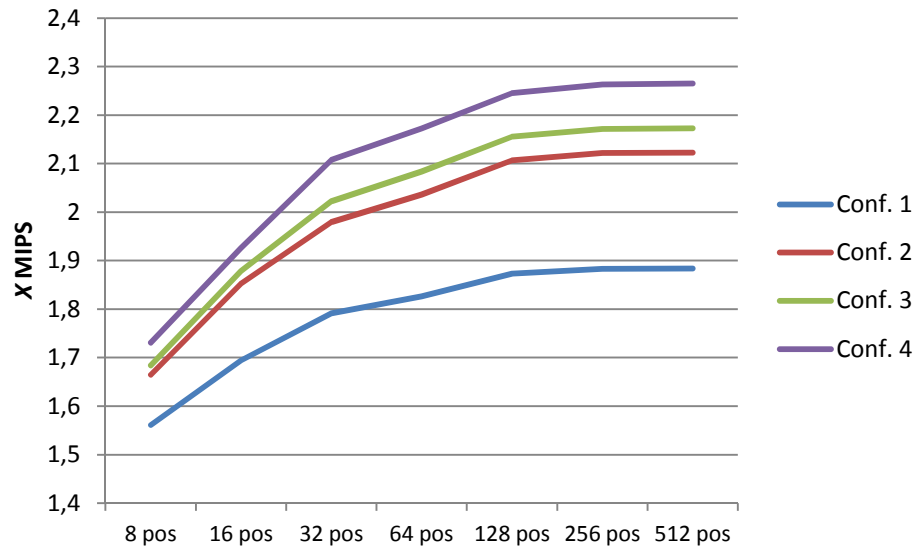


Figura 3.8: Aceleração média para cada configuração.

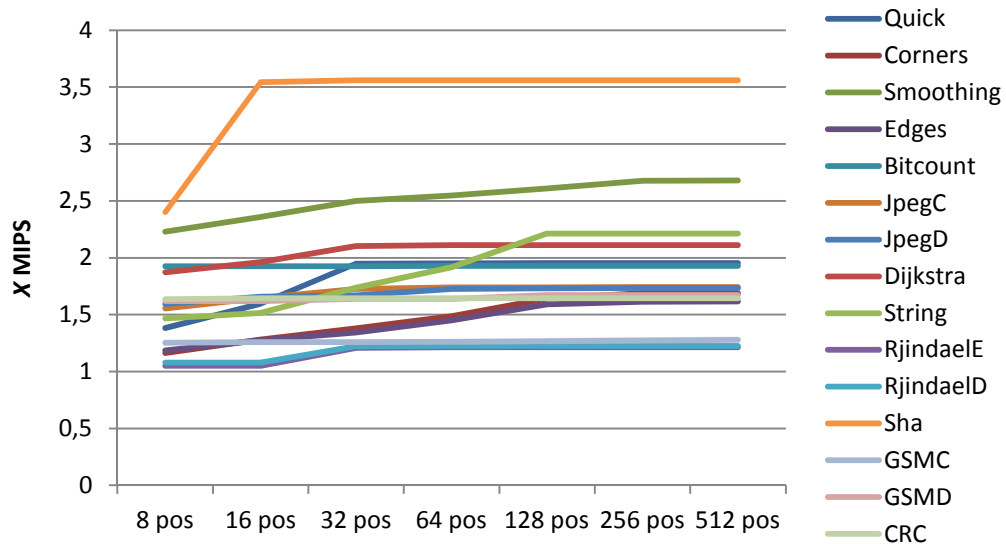


Figura 3.9: Aceleração para Configuração 1 para cada aplicação do benchmark *Mibench*.

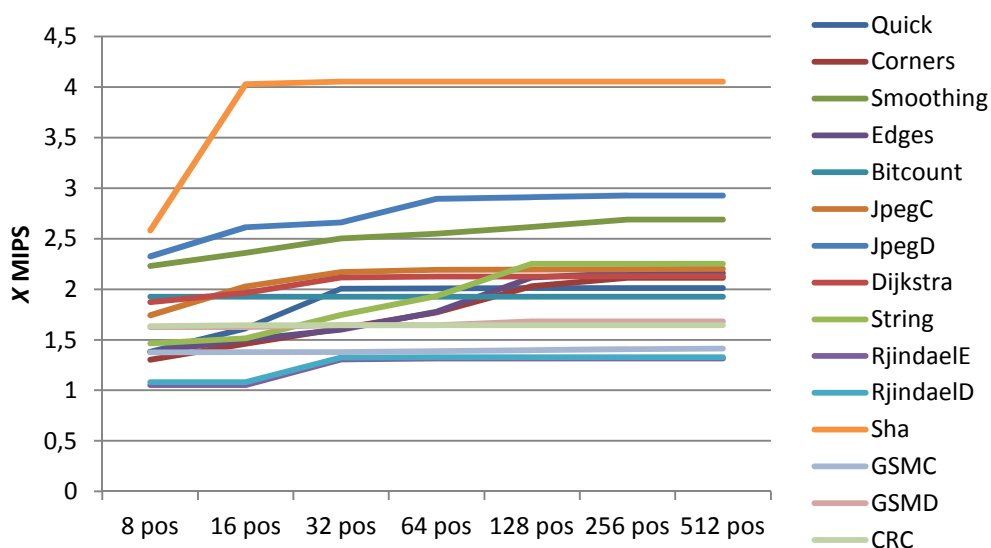


Figura 3.10: Aceleração para Configuração 2 para cada aplicação do *benchmark Mibench*.

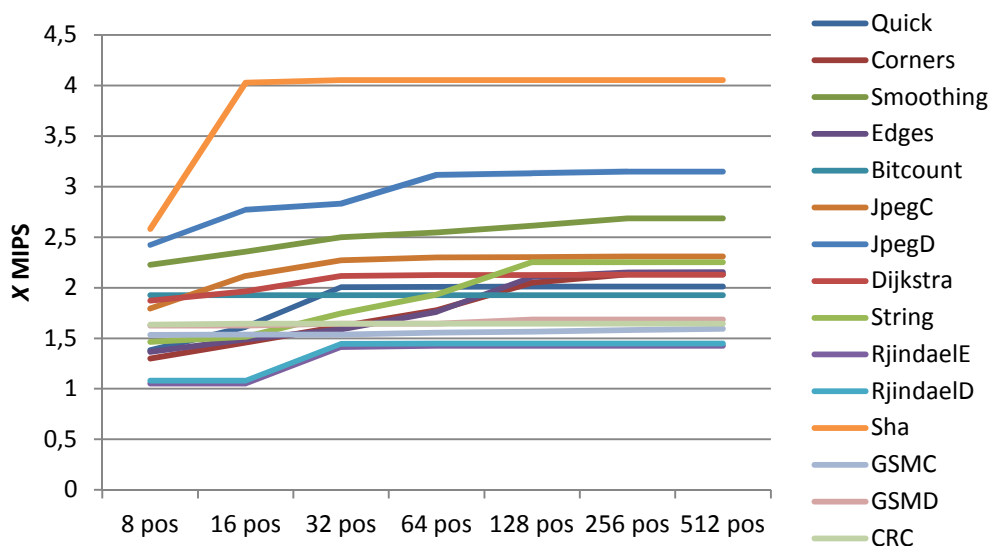


Figura 3.11: Aceleração para Configuração 3 para cada aplicação do *benchmark Mibench*.

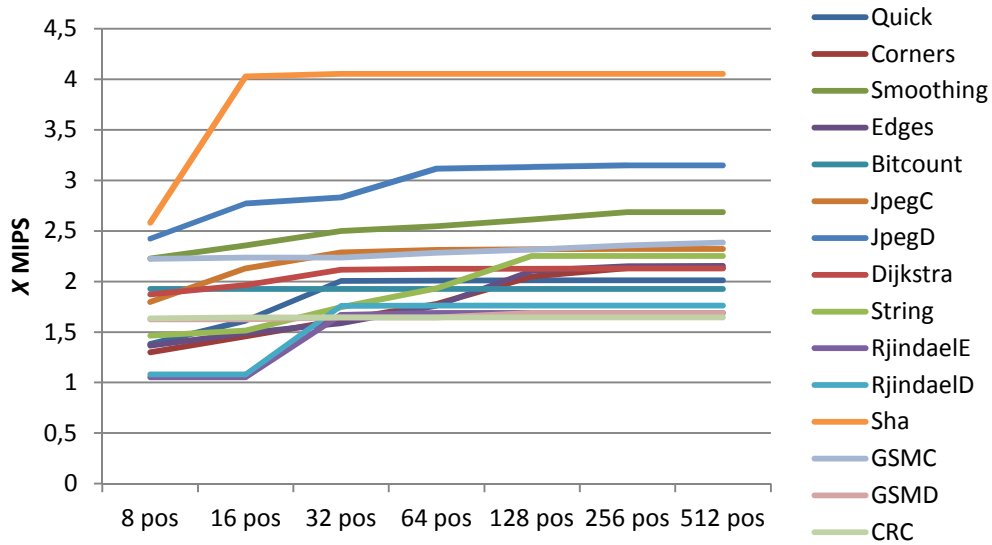


Figura 3.12: Aceleração para Configuração 4 para cada aplicação do *benchmark Mibench*.

Na Figura 3.13 é apresentada a aceleração obtida para cada configuração apresentada na Tabela 3.2, fixando o tamanho da memória de contexto em 64 entradas.

Na Figura 3.13 pode-se observar características das aplicações em relação ao aumento de unidades disponíveis. Para aplicação *Sha*, uma matriz reconfigurável com tamanho da Configuração 2 é suficiente, visto que a aceleração satura após esse ponto. Já para as aplicações *Smoothing*, *String* e *Dijkstra* não é necessário uma matriz reconfigurável maior que a Configuração 1. Porém, as aplicações como *JpegD* e *GSM* conseguem obter maior desempenho com o aumento de unidades e níveis disponíveis.

Na aplicação *GSM* é possível visualizar um salto na aceleração da Configuração 3 para a 4. O aumento se deve principalmente pela união de *kernels* que antes, devido à matriz não possuir UFs disponíveis, eram divididos e agora fazem parte de uma única configuração. Assim, economiza-se o tempo de uma nova reconfiguração e gravação dos resultados, beneficiando o desempenho.

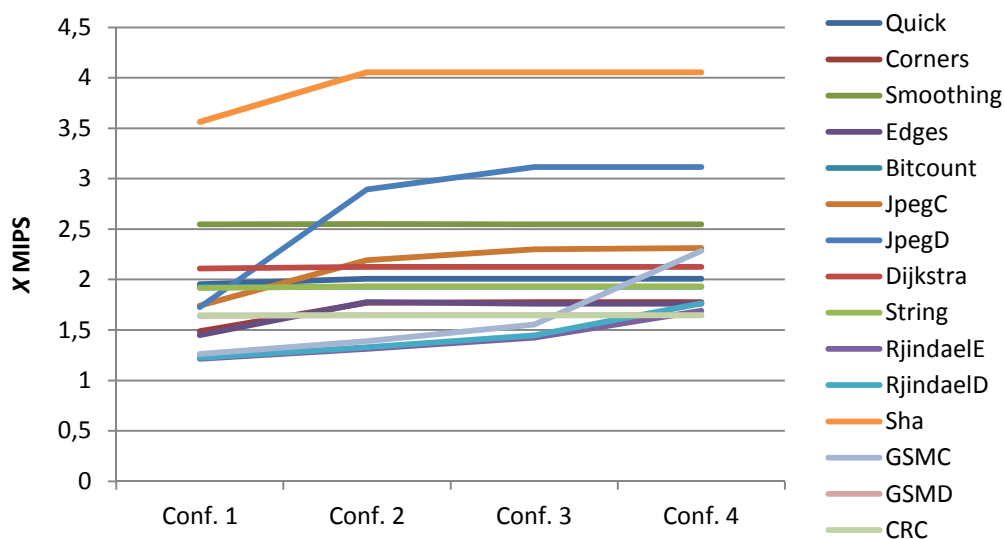


Figura 3.13: Aceleração obtida para cada configuração, fixando o tamanho da memória de contexto em 64 entradas.

A Figura 3.14 mostra a distribuição do reuso das configurações na execução de todo *benchmark*, considerando todas as configurações da matriz reconfigurável apresentada na Tabela 3.2- os dados ajudam a descobrir qual o tamanho da matriz ideal (em número de níveis), sem comprometer o desempenho. Esse é um dado importante, uma vez que o número de níveis da matriz reconfigurável afeta diretamente o tamanho da memória de contexto. O eixo X representa o número de níveis da configuração, enquanto o eixo Y representa o número de vezes que as configurações que apresentam o mesmo número de níveis foram reutilizadas. Tal gráfico demonstra que aproximadamente 99% das configurações reutilizadas usam menos de 10 níveis. A Tabela 3.3 apresenta a porcentagem do número de níveis utilizados na execução de todo *benchmark* para cada configuração.

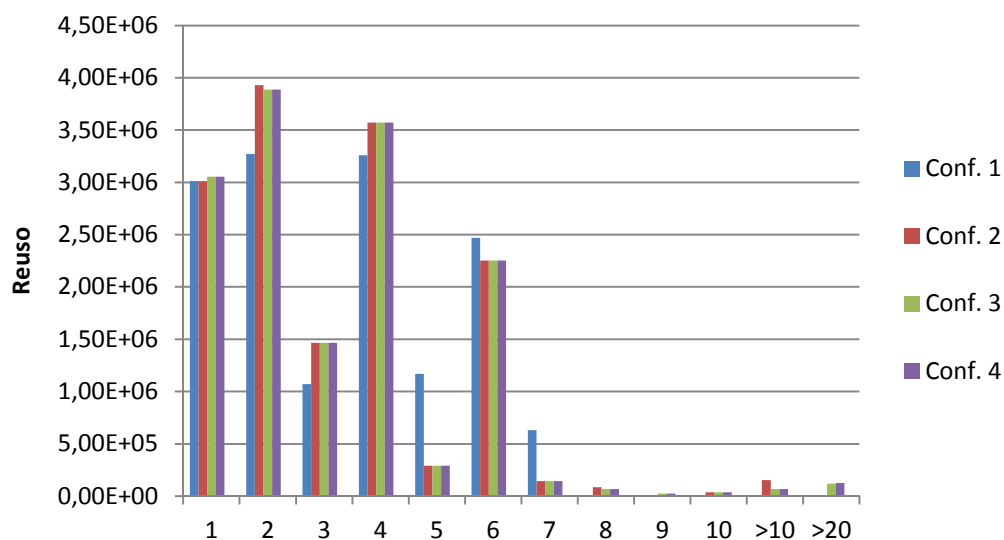
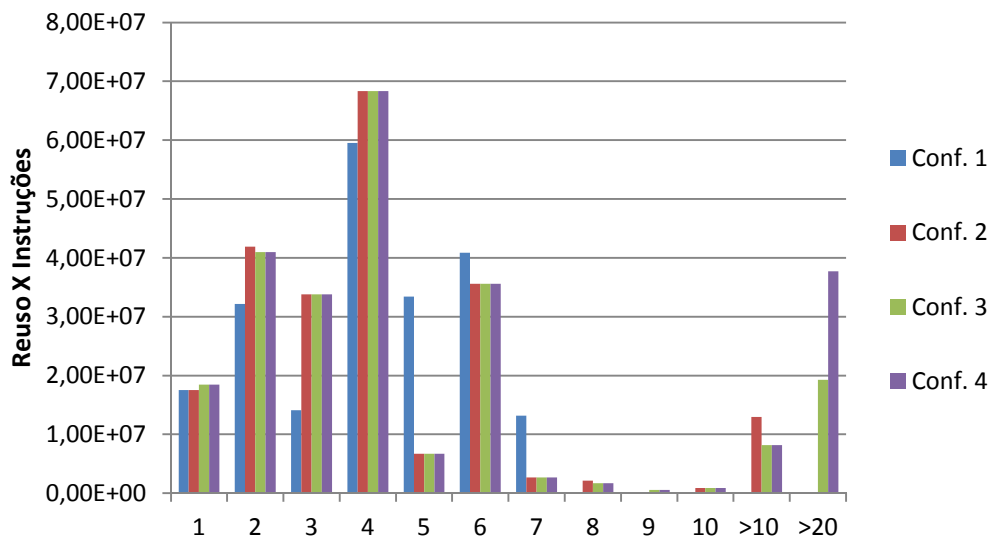


Figura 3.14: Distribuição do reuso das configurações na execução de todo *benchmark*.

Tabela 3.3: Porcentagem do número de níveis utilizados na execução de todo benchmark.

# Níveis	1 a 10	11 a 20	>20
Conf. 1	100%	-	-
Conf. 2	98,98%	1,02%	-
Conf. 3	98,77%	0,44%	0,79%
Conf. 4	98,73%	0,44%	0,83%

Aparentemente, uma matriz com poucos níveis seria o suficiente para atingir quase 100% da aceleração possível. No entanto, configurações com um grande número de níveis agrupam mais instruções e, por isso, acabam sendo responsáveis por grande parte da aceleração alcançada (Figura 3.15). O eixo Y é definido pela reutilização da configuração multiplicada pelo número de instruções que compõem cada configuração. Na Tabela 3.4 é apresentada a porcentagem das instruções que executaram na matriz reconfigurável, agrupadas pelo número de níveis da configuração gerada pelo TB.

Figura 3.15: Distribuição do reuso x número de instruções das configurações na execução de todo *benchmark*.Tabela 3.4: Distribuição instruções executadas na arquitetura reconfigurável na execução de todo *benchmark*.

# Níveis	1 a 10	11 a 20	>20
Conf. 1	100%	-	-
Conf. 2	94,19%	5,81%	-
Conf. 3	88,43%	3,45%	8,12%
Conf. 4	82,05%	3,20%	14,75%

A Tabela 3.4 mostra que as configurações com mais de 10 níveis correspondem a 5,81%, 11,57% e 17,95% de todas as instruções executadas na arquitetura reconfigurável, valores respectivos para as Configurações 2, 3, e 4 (a Configuração 1

possui somente 8 níveis). Portanto, embora haja um pequeno número de configurações com um grande número de níveis, elas representam uma parcela significativa do número total de instruções executadas, tornando-as não menos essenciais. Por sua vez, o grande tamanho de configurações resulta numa grande largura de porta de memória, o que implica mais energia de acesso, reforçando a necessidade de um mecanismo eficiente para lidar com esse problema.

Para analisar esses dados com mais detalhes, a Figura 3.16 mostra a reutilização da configuração multiplicada pelo número de instruções que compõem cada configuração, considerando apenas a aplicação *GSMC*, usando-se a Configuração 3.

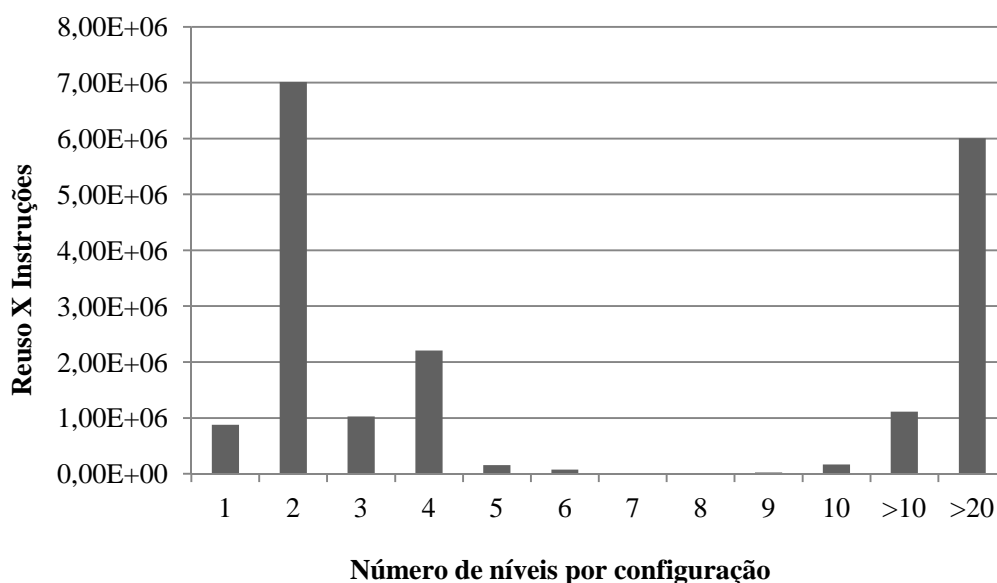


Figura 3.16: Distribuição do reuso x número de instruções das configurações na execução da aplicação *GSMC*.

A aplicação *GSMC*, apresentada na Figura 3.16, é um exemplo de aplicação que usa todos os níveis disponíveis. Como demonstradas, as configurações que possuem mais de dez níveis são responsáveis por 40% (entre 10 e 20 correspondem a 6% e mais que 20 correspondem a 34%) de todas as instruções executadas na arquitetura reconfigurável, reforçando o fato de que é necessário incluí-los caso se deseje alcançar maior aceleração.

Como observado, o sistema reconfigurável pode obter ganhos em desempenho, entretanto há um custo para isso. Conforme a Figura 3.8, quanto maior a configuração (maior número de unidades funcionais), maior é a aceleração que se pode alcançar. Contudo, ao passo que se insere mais unidades funcionais, maior é a área necessária para a matriz reconfigurável. Ainda na Figura 3.8, pode-se visualizar que o número de entradas da memória de contexto também influencia na aceleração, quanto maior o número de entradas, maior será a aceleração. Em ambos os casos existe um limite de desempenho que se pode alcançar, o qual depende da aplicação executada.

A Tabela 3.5 apresenta os valores de área relativa ocupada pela matriz reconfigurável e o processador MIPS R3000, a capacidade necessária da memória de contexto para armazenar 64 entradas e a aceleração obtida para as quatro configurações analisadas. Para a menor configuração (Conf. 1) é necessária uma matriz reconfigurável de aproximadamente 26,9 vezes o tamanho do processado e uma memória de contexto

com capacidade de 52KB para atingir aceleração de 1,82, enquanto para a maior configuração (Conf. 4) é necessária 74,2 vezes a área do processador e 534KB de memória para alcançar a aceleração 2,17.

Tabela 3.5: Valores de área relativa ocupada pela matriz reconfigurável e o processador MIPS R3000, a capacidade necessária da memória de contexto (64 entradas) e a aceleração.

	Conf. 1	Conf. 2	Conf. 3	Conf. 4
Área da matriz reconfigurável em relação ao R3000	26,9	74,2	161,2	322,4
Capacidade da memória de contexto com 64 entradas (bytes)	52.480	117.632	272.704	534.848
Aceleração (MIPS)	1,82	2,03	2,08	2,17

Na Tabela 3.5 observa-se o aumento de área que a matriz reconfigurável acarreta no sistema final para se atingir a aceleração desejada. Além da matriz reconfigurável, a memória de contexto também contribui para este aumento, pois, são necessárias memórias de elevada capacidade para armazenarem os bits de configuração. Tendo esta uma enorme largura de porta e sendo frequentemente acessada, a memória de contexto pode elevar consideravelmente o total de energia consumida pelo sistema, de tal modo que, pode-se tornar inviável a utilização de arquiteturas reconfiguráveis, devido às restrições de área e energia em dispositivos embarcados.

Como citado anteriormente, a energia consumida pela matriz reconfigurável foi abordado no trabalho (RUTZIG, 2008), e como o mesmo obteve resultados satisfatórios, consideraremos a técnica de *Sleep Transistor* apresentada para efeitos de análise de consumo de energia da matriz reconfigurável. Portanto, o foco principal de redução do consumo de energia é a memória de contexto.

Nos próximos capítulos são apresentadas técnicas para de minimizar os problemas de aumento de área e energia ocasionado pela arquitetura reconfigurável. No Capítulo 4 são apresentadas técnicas de virtualização de hardware, quais permitem reduzir a área da matriz e no Capítulo 5 são propostas abordagens que modificam a memória de contexto de forma a reduzir o consumo de energia nos acessos da mesma.

4 VIRTUALIZAÇÃO DE HARDWARE

Neste capítulo é demonstrada a virtualização de hardware da matriz reconfigurável, que visa reduzir a área ocupada pela matriz de unidades funcionais com a mínima perda de desempenho possível. Para fins didáticos, primeiramente o comportamento temporal da execução será apresentado e, em seguida as técnicas de virtualização propostas serão demonstradas.

4.1 Visão temporal da execução na arquitetura reconfigurável

As Figuras 4.1 e 4.2 ilustram uma visão temporal da execução da matriz reconfigurável. Nessas figuras é possível visualizar o processamento ao longo do tempo, iniciando em $T=0$, até o instante $T=6$ – momento em que termina o tempo de execução (considera-se que a configuração utiliza todos os 6 níveis da matriz). As etapas expressas ocorrem sempre que a matriz reconfigurável é utilizada.

As unidades funcionais, bem como os barramentos de dados, foram simplificados para possibilitar uma melhor visualização. No exemplo apresentado a matriz reconfigurável é composta por seis níveis – cabe lembrar que o mesmo ocorre no caso da matriz possuir mais níveis.

Inicialmente todas as unidades funcionais e o caminho de dados (multiplexadores de entrada e saída) são configurados. Como visto no Capítulo 3, o tempo do ciclo é equivalente a um período do relógio do processador, que é equivalente a execução de um nível de unidades funcionais. A fase de configuração é representada na Figura 4.1(a) em amarelo.

Uma vez que o circuito esteja configurado, o instante $T=0$ denota o início da computação. Logo, conforme apresentado no Capítulo 3, a execução ocorre de forma combinacional e o sinal percorre o caminho de dados definido pela configuração, até completar a execução no instante $T=6$ como apresentado na Figura 4.2(d). Vale ressaltar que a execução pode terminar antes do instante $T=6$, pois o tempo de execução é determinado pelo caminho crítico. Entretanto, é necessário aguardar o tempo determinado pelo caminho crítico para garantir que a computação foi totalmente realizada e os sinais de saída se encontrem estáveis.

Nas Figuras 4.1 e 4.2, as unidades reconfiguráveis destacadas em vermelho representam os sinais que estão sendo computados, ao passo que as unidades ressaltadas em verde corroboram a já computação e estabilidade do sinal. A cada instante T definido é possível observar que os sinais são processados ao longo do tempo.

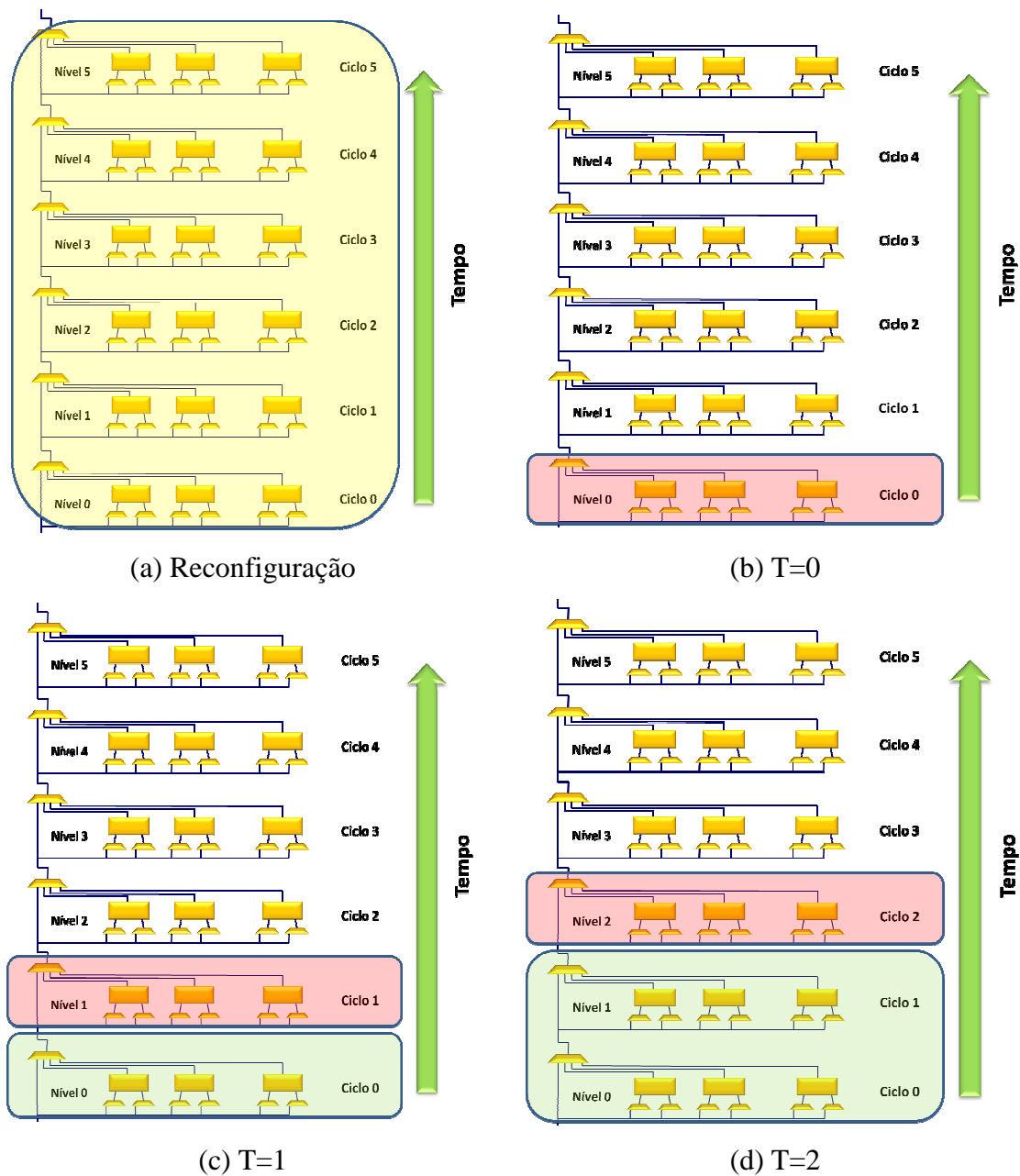


Figura 4.1: Execução de uma configuração com 6 níveis (parte inicial).

Tomando como exemplo o instante de tempo em que $T=5$ (Figura 4.2 (c)), pode-se visualizar que os cinco níveis iniciais (níveis 0-4) já realizaram as operações e seus sinais de saída se encontram estáveis, enquanto a computação do Nível 5 teve início.

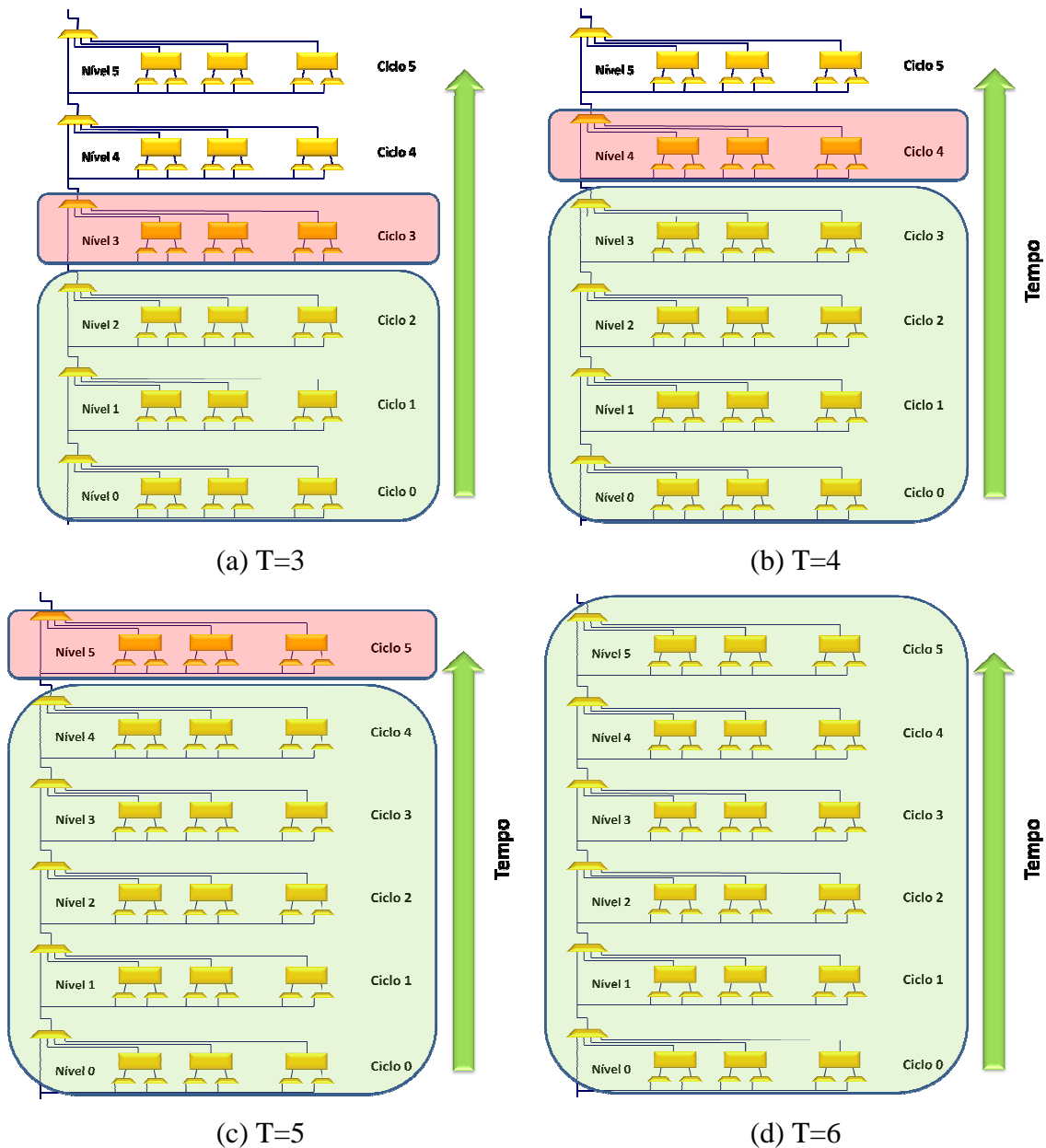


Figura 4.2: Execução de uma configuração com 6 níveis (parte final).

Ao analisar as Figuras 4.1 e 4.2 é possível constatar que as unidades funcionais de cada nível são utilizadas somente uma vez em cada execução. Assim, considerando o tempo de uso durante a execução de uma configuração, o fator de utilização das unidades funcionais é baixo.

Como cada nível é composto de um grande número de unidades funcionais, essenciais para a obtenção do potencial de aceleração desejado das aplicações, esse fator de utilização pode piorar. Isso ocorre porque muitas aplicações podem não utilizar todos os recursos disponíveis, seja por falta de paralelismo seja por forte dependência das operações/dados. Esse baixo fator de utilização das unidades impacta negativamente, pois é gasto grande quantidade de recurso de área para a fabricação da matriz reconfigurável que frequentemente são subutilizados.

Outro fator na execução diz a respeito à função dos níveis que já processaram a computação. Tomando como exemplo a Figura 4.2 (b), o Nível 4 iniciou a execução, uma vez que os níveis anteriores já completaram sua execução e estabilizaram o sinal. Nesse momento os níveis inferiores (Níveis 0, 1, 2 e 3) têm apenas a função de transmitir os sinais e mantê-los estáveis para o próximo nível. Novamente, quando se pensa em eficiência de uso, os recursos são subutilizados.

Já com relação ao consumo de energia, os níveis com a computação concluída, apesar de não consumirem mais potência dinâmica, permanecem consumindo potência estática (devido à necessidade de continuarem mantendo os sinais ativos para que os demais níveis funcionem corretamente).

Nesse exemplo apresentado, a matriz reconfigurável possui somente seis níveis; entretanto, no estudo de caso são utilizados 8, 16, 32 e 64 níveis - a matriz que atinge todo potencial da técnica demonstrada por (BECK, 2006) possui 85 níveis. Aqui os fatores descritos anteriormente podem gerar grandes benefícios para o sistema reconfigurável quando otimizados, devido principalmente ao elevado número de níveis da arquitetura reconfigurável.

Uma das formas de minimizar os problemas descritos acima, *i.e.* grandes quantidade de recursos de área necessários e subutilização de recursos, acontece por meio da utilização de técnicas de virtualização de hardware. Algumas formas de virtualização de hardware reutilizam parte dos recursos que já foram utilizados ou dividem a aplicação em blocos menores para serem executados sequencialmente, permitindo que uma aplicação de tamanho arbitrário possa ser executada em um hardware com menos recursos.

A Figura 4.3, baseada no exemplo da Figura 4.2, mostra uma forma de eliminar a necessidade de alguns níveis que já completaram a sua execução permanecerem ativos. Esse é um exemplo inicial que servirá de base para a compreensão da forma como a técnica de virtualização será aplicada neste trabalho.

No exemplo da Figura 4.3, foi realizada a inserção de um registrador na linha de contexto de saída do Nível 2. Dessa forma, quando o tempo de execução for maior que $T=4$, ou seja, após a estabilização do sinal de saída do nível que possui o registrador, o sinal é armazenado no registrador. Assim, torna-se desnecessário que os níveis inferiores permaneçam ativos, transferindo para o registrador a responsabilidade de manter e fornecer o sinal para o próximo nível.

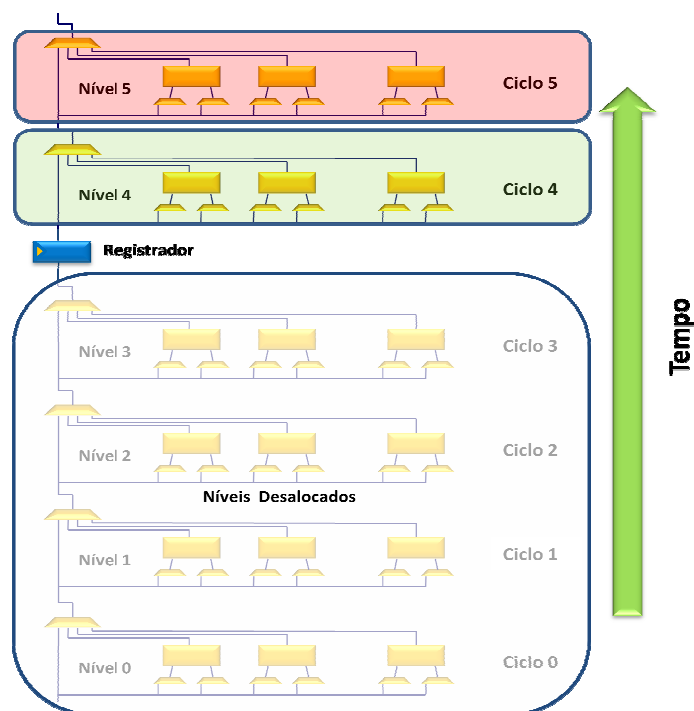


Figura 4.3: Inserção do registrador na linha de contexto para desativação dos níveis já processados.

Na próxima seção, serão explanadas as técnicas de virtualização que foram aplicadas no sistema reconfigurável utilizado no estudo de caso do Capítulo 3, objetivando a redução da área ocupada pela matriz reconfigurável e o uso mais eficiente das unidades funcionais disponíveis.

4.2 Virtualização de hardware por meio da técnica de *pipeline* de estágio reconfigurável

A virtualização de hardware proposta é aplicada por meio da modificação da arquitetura reconfigurável, criando um *pipeline* virtual dos níveis. A ideia principal é aproveitar a regularidade da arquitetura e virtualizar os demais níveis com um pequeno número de níveis físicos (denominados estágios).

Um exemplo da técnica de virtualização de hardware por meio de *pipeline* de estágio reconfigurável foi apresentado no Capítulo 2, na arquitetura reconfigurável denominada PipeRench (GOLDSTEIN, 2000).

Basicamente, a virtualização se dá pela inserção de registradores nas linhas de contexto de saída de nível, conforme a Figura 4.4. Esses registradores armazenam as saídas das linhas de contexto que serão utilizados pelo próximo nível, semelhante a um *pipeline*.

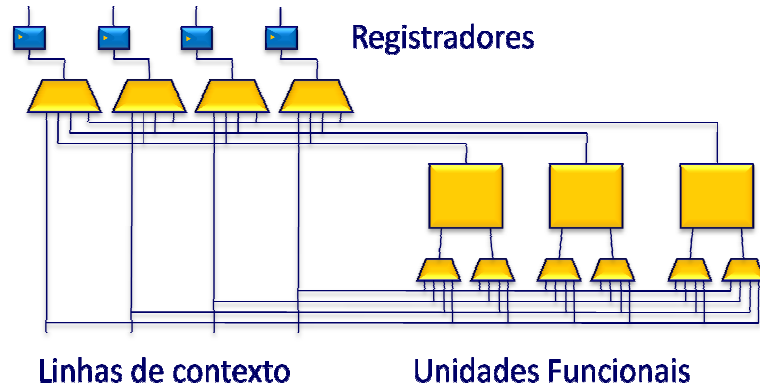


Figura 4.4: Nível com registradores nas saídas das linhas de contexto.

Duas formas de implementação da matriz reconfigurável foram modeladas a partir dessa modificação. No primeiro modelo é utilizado 1 nível físico para virtualizar os demais, ao passo que no segundo modelo é utilizado o dobro de níveis (os detalhes de cada modelo estão presentes a seguir).

4.2.1 Pipeline virtual de 1 estágio

Nessa implementação é adotado somente um estágio físico composto de um nível da matriz reconfigurável. As saídas do estágio realimentam a entrada do mesmo, resultando em um circuito do tipo *pipeline* cíclico de um estágio, conforme demonstrado na Figura 4.5.

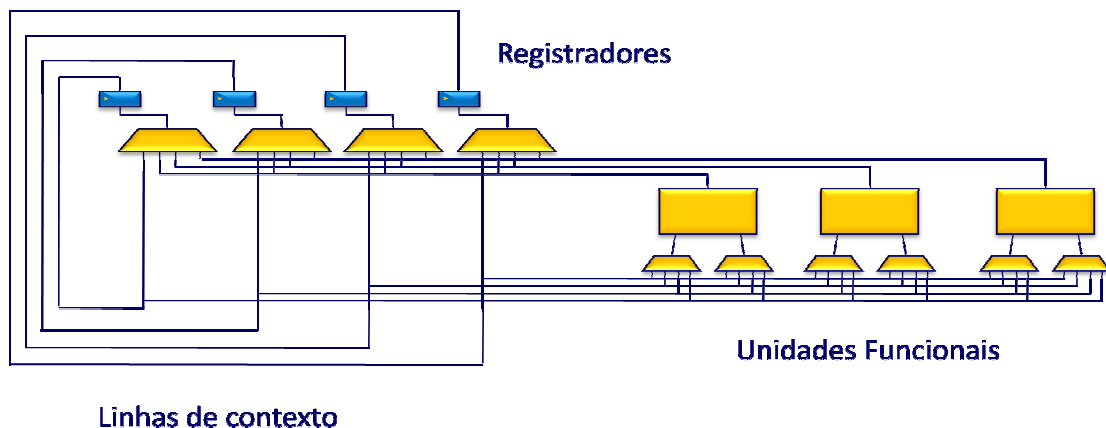


Figura 4.5: Virtualização por meio de 1 estágio físico.

A cada ciclo o estágio é reconfigurado para virtualizar o próximo nível virtual que será executado (o que torna possível a execução de tantos níveis quantos forem necessários). Dessa forma, a virtualização do circuito proporciona uma grande economia em área, pois quanto maior o número de níveis da matriz original, maior será a economia relativa.

O funcionamento desse tipo de *pipeline* procede da seguinte forma: em um mesmo ciclo, primeiramente a configuração é realizada e depois a execução do nível. Em seguida, a cada novo ciclo o estágio é reconfigurado de acordo com o nível virtual que se deseja executar. Por fim, os registradores das linhas de contexto armazenam os resultados intermediários da execução, que serão a entrada para o próximo nível virtual a ser processado.

A busca da configuração do nível é realizada no ciclo anterior à configuração do estágio. Assim, enquanto é realizado o processamento do nível atual, paralelamente é obtida a configuração do próximo nível a partir da memória de contexto.

O mecanismo de controle do circuito da Figura 4.5 é basicamente definido pela busca da configuração do próximo nível e pela configuração e execução do nível atual. Em suma, o controle corresponde às seguintes ações (em que n corresponde ao nível atual que está em execução):

- Busca da configuração do nível ($n+1$);
- Reconfiguração e execução do nível (n).

O tempo do período do ciclo e o tempo total de execução podem ser calculados pelas equações abaixo:

$$T_{ee1} = T_c + T_{en} + T_s \quad (\text{Eq. 1})$$

$$T_{te1} = N \times T_{ee1} \quad (\text{Eq. 2})$$

em que: T_{ee1} = Tempo de execução do estágio

T_c = Tempo de configuração

T_{en} = Tempo de execução do nível

T_s = Tempo de setup do registrador

T_{te1} = Tempo total de execução

N = Número de níveis de configuração que serão executados

A abordagem de um estágio tem o desempenho prejudicado, porém, é a que corresponde à menor área ocupada se comparada a matriz reconfigurável original. Como observado na Equação 1, o tempo de configuração faz parte do tempo de processamento do nível, o que não ocorria na arquitetura original, em que a configuração era realizada uma única vez previamente ao início da execução.

4.2.2 Pipeline virtual de 2 estágios

Essa seção descreve o modelo de uma matriz reconfigurável de dois estágios, proposto para contornar o acréscimo do tempo total de execução devido à configuração ser realizada no mesmo ciclo que a execução.

A implementação da matriz reconfigurável de 2 estágios pode ser observada na Figura 4.6. As saídas Nível 0 são conectadas nas entradas do Nível 1 e as saídas dos Nível 1 são conectadas na entrada do Nível 0. Como no modelo anterior, as saídas das linhas de contexto do nível também possuem registradores para armazenar o resultado do processamento do nível, o que resulta em um *pipeline* cíclico de 2 estágios.

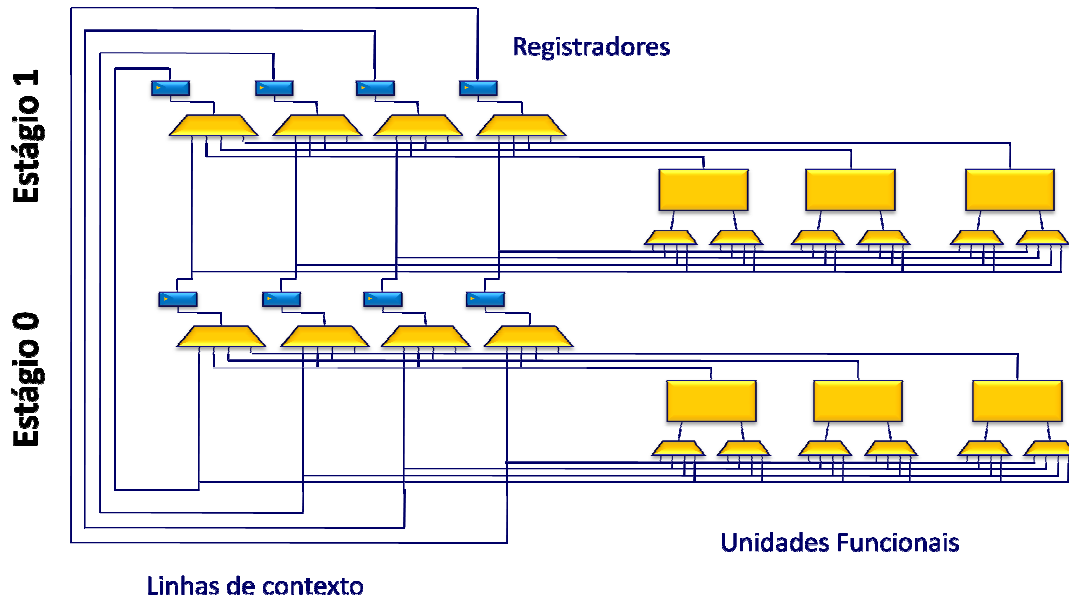


Figura 4.6: Virtualização por meio de 2 estágios físicos.

Nesse modelo com 2 estágios, a execução ocorre paralelamente à reconfiguração: enquanto um estágio está executando, o outro está sendo reconfigurado de acordo com o próximo nível virtual. Todos os níveis a serem executados existentes na configuração são virtualizados nos dois estágios disponíveis, sendo executado um estágio por ciclo, da mesma forma que ocorre na virtualização de 1 estágio.

A Figura 4.7 ilustra a virtualização de hardware por meio do mapeamento dos níveis virtuais nos níveis físicos em que uma configuração de 6 níveis é executada. No primeiro ciclo o nível virtual 0 é configurado no estágio 0, já no ciclo seguinte, o nível virtual 1 é mapeado no segundo nível físico ao mesmo tempo que o primeiro estágio é executado.

No terceiro ciclo o nível virtual 2 é configurado no estágio 0, o mesmo nível físico em que antes fora mapeado o nível virtual 0. Os resultados da execução desse nível já foram armazenados previamente nos registradores, portanto, a reconfiguração do nível não ocasiona problema na passagem do sinal para o nível seguinte.

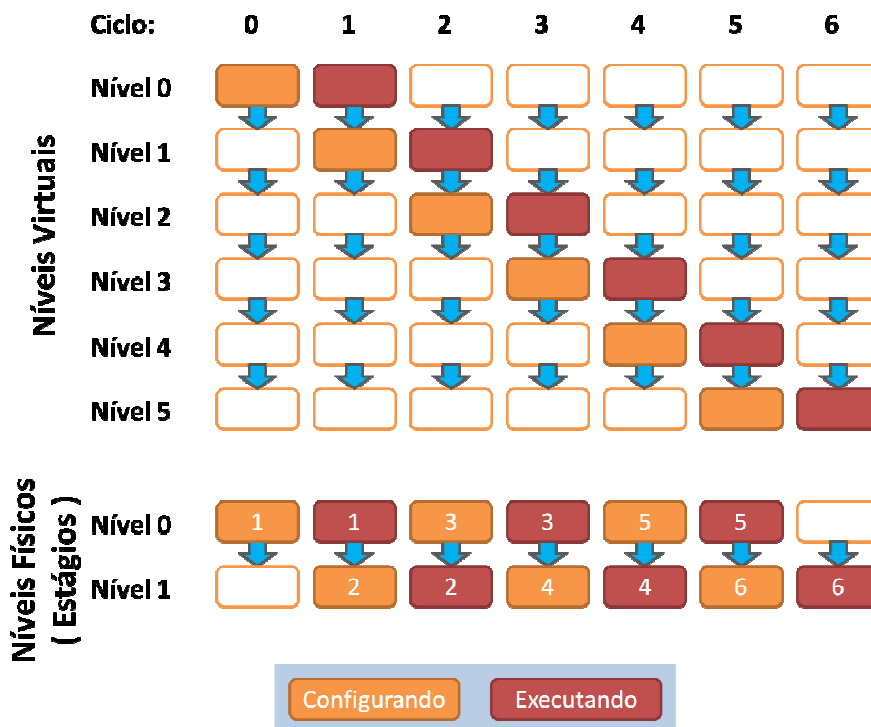


Figura 4.7: Mapeamento dos níveis virtuais nos 2 estágios.

O mecanismo de controle opera da seguinte forma: quando um nível n está sendo executado, a saída do nível anterior é registrada, o nível $n+1$ é reconfigurado e a configuração no nível $n+2$ é obtida da memória de contexto - tais etapas ocorrem paralelamente no mesmo ciclo. Resumidamente as ações são as seguintes:

- Busca da configuração do nível $(n+2)$;
- Reconfiguração do nível $(n+1)$;
- Execução do nível n ;
- Registro do nível $(n-1)$.

O tempo do período do ciclo e o tempo total de execução podem ser calculados pelas equações abaixo:

$$T_{ee2} = T_{en} + T_s \quad (\text{Eq. 3})$$

$$T_{te2} = N \times T_{ee2} \quad (\text{Eq. 4})$$

em que: T_{ee2} = Tempo de execução do estágio

T_{en} = Tempo de execução do nível

T_s = Tempo de setup do registrador

T_{te2} = Tempo total de execução

N = Número de níveis de configuração que serão executados

Nesta abordagem, como na original, o tempo de reconfiguração é menor que o tempo de processamento de 1 estágio, o que justifica a equação do tempo de processamento do nível.

O que diferencia o modelo da arquitetura de 2 estágios para a de 1 estágio é o processo de reconfiguração. No modelo de 2 estágios a reconfiguração do nível ocorre em paralelo à execução, ao passo que, no modelo de 1 estágio, a reconfiguração e execução ocorrem de forma sequencial. Essa diferença implica diretamente no tempo de processamento do estágio, já que o tempo de configuração não faz mais parte da equação. Em contrapartida, o segundo modelo necessita o dobro de área, uma vez que é composto de 2 estágios.

Uma abordagem de virtualização semelhante ocorre na arquitetura reconfigurável chamada PipeRench, como citado no Capítulo 2. Tal arquitetura é composta de 3 estágios (*strips*), em que cada estágio é configurado em 1 ciclo e executado nos próximos 2 ciclos. Da mesma forma que o PipeRench, os modelos propostos podem ser classificados como execução virtualizada, de acordo com a classificação (PLESSL, 2004) apresentada no Capítulo 2.

4.2.3 Implicações da utilização da virtualização por meio de *Pipeline*

A virtualização de *hardware* por meio de um *pipeline* virtual dos estágios implica acrescentar um registrador no final de cada estágio. A inserção do registrador acrescenta o tempo de *setup* do registrador ao tempo de execução, pois faz parte do caminho crítico. Apesar do tempo de *setup* ser pequeno comparado ao tempo de execução do estágio, esse atraso pode ocasionar perda de desempenho.

Para amenizar o atraso do tempo de *setup* um estudo de caso pode ser realizado modificando a granularidade do estágio. Nos modelos de 1 e 2 níveis abordados, cada estágio corresponde a um nível. Entretanto, se cada estágio fosse formado por dois ou quatro níveis, ou seja, um registrador a cada dois níveis ou quatro níveis, o tempo de *setup* seria reduzido proporcionalmente.

Na arquitetura do estudo de caso, o aumento do caminho crítico corresponde a cerca de 2% quando cada nível corresponde a um estágio. Caso for definido que cada estágio seja formado de 2 níveis, o caminho crítico seria acrescido de cerca de 1%. A Figura 4.8 apresenta essa exploração do espaço de projeto.

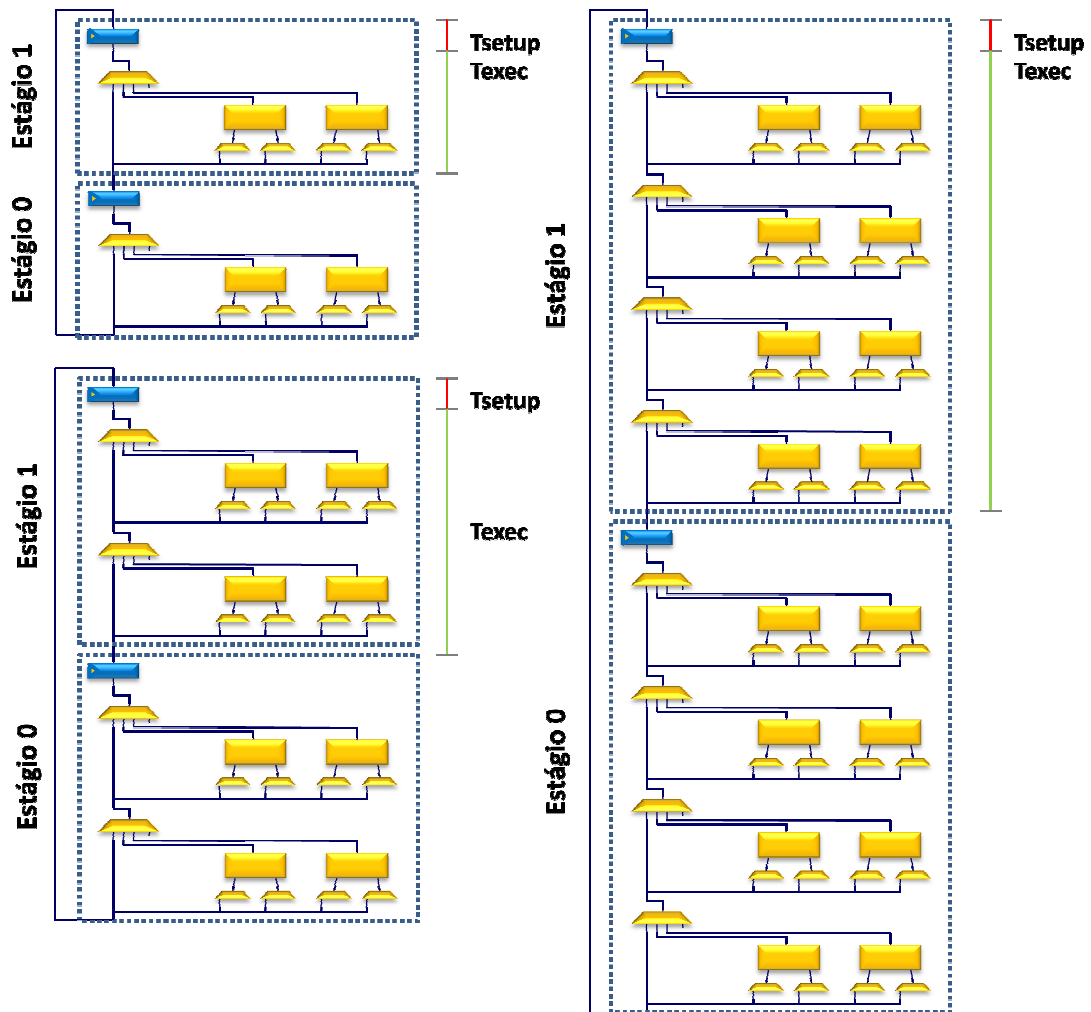


Figura 4.8: Variação da granularidade de cada estágio.

Originalmente a matriz reconfigurável era constituída de conjuntos de unidades funcionais conectadas de maneira combinacional. A inserção do registrador implica transformar a lógica anteriormente combinacional em lógica sequencial, o que pode igualmente afetar o tempo de execução. Anteriormente a lógica combinacional dos níveis poderia ser mais bem otimizada, o que possibilita a obtenção de um menor tempo de execução.

Para melhor entendimento, toma-se como exemplo um circuito em que é necessário realizar sucessivas operações de soma, sendo a expressão alvo:

$$S = A + B + C + D + E \quad (\text{Eq. 5})$$

Duas abordagens da implementação para esse circuito podem ser conferidas na Figura 4.9. Na primeira abordagem o circuito é formado por uma cascata de somadores, conforme a Figura 4.9(a); já na outra abordagem é utilizado somente um somador e o resultado intermediário é armazenado em um registrador, como apresentado na Figura 4.9(b). A primeira solução é totalmente combinacional e a segunda pode ser classificada como sequencial. No exemplo da Figura 4.9, os valores A, B, C, D e E são formados de 4 bits e são utilizados somadores *Ripple-Carry* (WESTE, 2010).

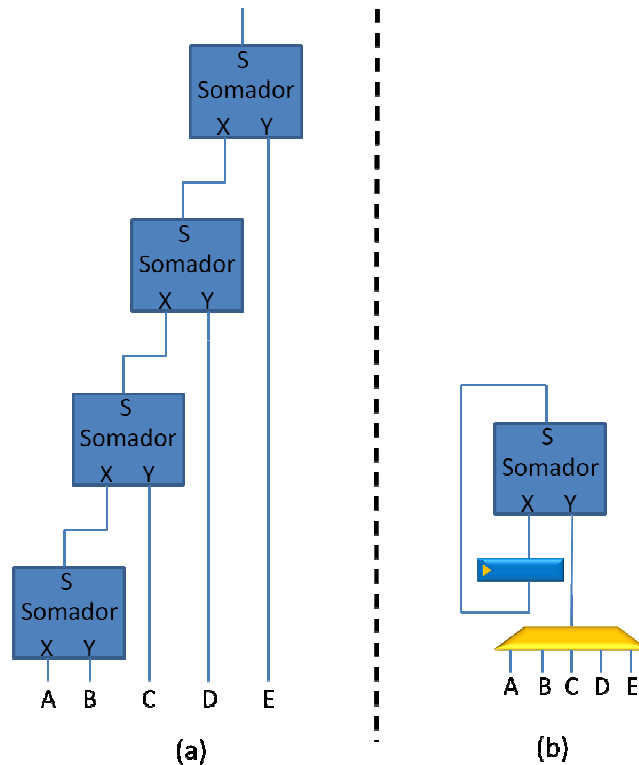


Figura 4.9: Diferentes formas de realizar somas sucessivas por solução combinacional (a) e sequencial (b).

Para calcular o tempo de computação, considerou-se o atraso do somador de quatro bits como sendo 4 unidades de tempo (t), e o tempo de *setup* do registrador (e da máquina de estados do segundo circuito) nulo.

Calculando o tempo de computação do circuito da Figura 4.9(b) obtemos a seguinte equação simplificada:

$$T_{tess} = 4 \times T_{sum} \quad (\text{Eq. 6})$$

em que: T_{tess} = Tempo total de execução do somador sequencial

T_{sum} = Tempo de execução de um somador

Se

$$T_{sum} = 4t$$

então:

$$T_{tess} = 16t$$

Para a solução apresentada na Figura 4.9(a), à primeira vista, pode parecer que o tempo de computação é idêntico ao do segundo circuito, porém, isso não é verdade. O comportamento do circuito formado pela cascata de somadores pode ser observado na Figura 4.10, em que são detalhados os blocos somadores internos que compõem os somadores de 4 bits. Nessa figura é possível analisar o comportamento dinâmico do sinal ao longo do tempo.

Portanto, substituindo os valores na Equação 7, o tempo de execução do circuito da Figura 4.9(a) é de 7 unidades de tempo, conforme pode ser observado na Figura 4.10:

$$T_{tesc} = 4 + 3 \times 1 = 7 \quad (\text{Eq. 8})$$

Para verificar essa diferença no atraso foram realizados experimentos nos quais a cascata de somadores apresentada na Figura 4.9(a) foi descrita em VHDL. Nos experimentos utilizaram-se as ferramentas ISE e Power Compiler (tecnologia 0.18 mm) no projeto e sínteses dos circuitos.

No primeiro experimento a cascata de somadores utiliza somadores *Ripple-Carry*; já no segundo utilizou-se somadores *Carry-Lookahead* (WESTE, 2010). Em ambos os experimentos foram usados somadores de 32 bits. A Tabela 4.1 mostra os valores de atraso encontrados variando o número de somadores em cascata.

Tabela 4.1: Atraso de somadores *Rippe-Carry* e *Carry-Lookahead* em cascata.

# somadores em cascata	<i>Ripple-Carry</i>		<i>Carry-Lookahead</i>	
	Atraso total (ns)	DT (ns)	Atraso total (ns)	DT (ns)
1	6,71		4,37	
2	7,02	0,31	7,21	2,84
3	7,33	0,31	10,05	2,84
4	7,65	0,32	12,88	2,83

Aqui em ambos os tipos de somadores a adição de somadores acarreta um aumento quase constante do atraso do circuito - fator menor que o atraso de um somador. Para o somador *Ripple-Carry* a diferença do atraso (DT) foi de aproximadamente 0,31 ns, ao passo que para o somador *Carry-Lookahead* foi de 2,84 ns.

No próximo experimento, em vez de se utilizarem somadores, foram adotadas uma série de unidades lógicas e aritméticas (ULAs) que compõem a matriz reconfigurável. Essas ULAs implementam as seguintes operações: Soma, Subtração, Comparação e Lógica de Bit.

Os resultados da síntese da cascata de ULAs são apresentados na Tabela 4.2. Neste experimento também é possível observar que o acréscimo de novos níveis adiciona um atraso quase constante ao tempo total, em que o valor de DT foi de aproximadamente 3,16 ns.

Tabela 4.2: Atraso das unidades lógicas e aritméticas (ULAs) em cascata.

# ULAs em cascata	Atraso total (ns)	DT (ns)
1	4,42	
2	7,58	3,16
3	10,74	3,16
4	13,9	3,16
5	17,05	3,15

Para finalizar, foi projetada uma matriz reconfigurável variando de um a quatro níveis, em que cada nível é composto de 6 ULAs em paralelo com 6 linhas de contexto. Esse modelo é baseado no modelo original da matriz reconfigurável, totalmente

combinacional, semelhante à Figura 4.1. A Tabela 4.3 apresenta os resultados de atraso obtidos.

Tabela 4.3: Atraso da matriz reconfigurável com 6 ULAs em paralelo e 6 linhas de contexto.

# Níveis	Atraso total (ns)	DT (ns)
1	5,78	
2	10,95	5,17
3	16,11	5,16
4	21,27	5,16

Nesse experimento, onde são modelados níveis semelhantes ao da arquitetura do estudo de caso, também é possível observar que o acréscimo do atraso é menor que o atraso de um nível. Um nível apresentou atraso de 5,78 ns, e o acréscimo de mais níveis acarreta o incremento de 5,17 ns. A diferença chega muito próxima ao atraso de um nível, contudo, essa diferença acumulada pode ocasionar um atraso significativo.

Os valores DT obtidos nos experimentos dependem muito do comportamento do circuito combinacional que está sendo sintetizado. No caso do último exemplo, os valores dependem também do número de linhas de contexto, do número de ULAs em paralelo, além do tamanho do conjunto de bits que representam os dados.

A modificação da arquitetura por meio da virtualização dos níveis, incluindo registradores de passagem do contexto entre os estágios, possibilita obter-se grande redução da área ocupada, porém, esta técnica pode ocasionar perda de desempenho em relação à arquitetura original. Com os experimentos apresentados, pode-se observar que a execução de um conjunto de níveis combinacionais é mais rápida do que a de forma sequencial, em que os níveis são fragmentados em estágios e separados por uma barreira temporal.

A fim de minimizar a perda de desempenho ao se aplicar a técnica de virtualização dos estágios, foi modelada uma nova forma de passagem dos sinais de contexto no qual os registradores não fazem parte do caminho crítico. Esse terceiro modelo faz com que a transformação não perca desempenho devido à barreira temporal imposta pelos registradores, a qual quebra o comportamento combinacional. Na próxima seção essa nova técnica será descrita em detalhes.

4.3 Pipeline virtual

Para minimizar os atrasos decorrentes da transformação do circuito combinacional em um *pipeline* virtualizado, foi desenvolvida uma nova técnica derivada da de *pipeline* de estágio reconfigurável de 2 estágios (item 4.2.2). Entretanto, essa abordagem traz consigo algumas modificações na estrutura e no controle.

Nos *pipelines* de 1 e 2 estágios reconfiguráveis apresentados, o tempo de ciclo é igual ao atraso de um nível, pois a cada estágio existe uma barreira temporal. A barreira temporal exige que a computação esteja completa no fim do ciclo para que os dados possam ser armazenados e posteriormente serem repassados para o estágio seguinte.

Diferentemente do *pipeline* de estágio reconfigurável, a adição de novos níveis à matriz original incrementa somente um atraso DT, como observado nos experimentos. No circuito combinacional, parte dos dados são transmitidos para o estágio seguinte, mesmo sem ter-se concluído o processamento do nível atual. Um exemplo desse comportamento foi apresentado na Figura 4.10, em que os bits menos significativos do circuito somador ficam disponíveis muito antes dos bits mais significativos.

O objetivo dessa nova abordagem é construir os estágios de tal forma que mantenha a característica combinacional, obtendo vantagem em área ocupada e minimizando tempo de ciclo, a fim de atingir um atraso semelhante ao circuito combinacional.

Nessa nova técnica o tempo do ciclo é igual à diferença de tempo ao adicionar mais um estágio (DT). Na cascata formada por ULAs, o atraso foi de 4,42 ns (Tabela 4.2); contudo, ao utilizarmos esse mesmo exemplo juntamente com o *pipeline* virtual, o tempo de ciclo é igual a 3,16 ns, aproximadamente 25% menor. A diferença de desempenho, provocada pela diferença do tempo de ciclo, depende diretamente da diferença entre o tempo de processamento de uma unidade e o DT, como será analisado a seguir.

Para obter-se um comportamento combinacional, um multiplexador 2x1 e um registrador foram inseridos em cada saída das linhas de contexto, conforme a Figura 4.11 (a). Essa modificação permite que o sinal de saída de cada estágio possa ter origem nas unidades funcionais (de forma combinacional, como na Figura 4.11 (b)), ou no registrador (Figura 4.11(c)).

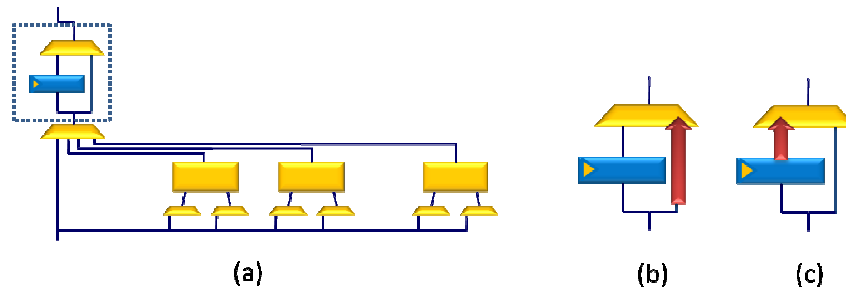


Figura 4.11: Modificação da estrutura do estágio para que o sinal de saída possa ter origem combinacional (b) ou registrada (c).

Para ilustrar o processo da dinâmica do circuito, é apresentado um exemplo de execução em que a virtualização dos níveis ainda não foi realizada. O modelo é composto de seis estágios, quais estão representados na Figura 4.12. Os atrasos são baseados nos valores da Tabela 4.3, os quais foram obtidos do experimento da matriz de ULAs. Assim, como no circuito combinacional, o primeiro estágio tem atraso de um nível e os demais estágios têm atraso igual à DT, que são respectivamente, 5,78 ns e 5,16 ns. O atraso decorrente do multiplexador 2x1 foi desprezado devido ao seu pequeno impacto no desempenho. O fluxo dos sinais para esse exemplo será apresentado na sequência de Figuras 4.13 e 4.14.

Inicialmente, a matriz de unidades é configurada e os dados iniciais são disponibilizados na entrada (como apresentado na Figura 4.13 (a)).

A Figura 4.13 (b) mostra o início da execução do Nível 0 no primeiro intervalo de tempo. Durante o intervalo, antes da execução do Nível 0 completar, parte do sinal já foi processado e é transmitido adiante, estimulando a entrada do nível seguinte. Para isso, o multiplexador desse nível deve estar com a entrada combinacional selecionada.

No intervalo seguinte (Figura 4.13 (c)), o Nível 0 já realizou o processamento, enquanto o Nível 1 está em execução. Novamente os dados parciais do nível que está em execução são transmitidos para o nível seguinte. É nesse intervalo que o registrador do Nível 0 armazena os dados resultantes da execução do primeiro nível. O armazenamento ocorre paralelamente à transmissão dos dados, pois o multiplexador permanece com a entrada combinacional selecionada.

Na execução do Nível 2, durante o terceiro intervalo de tempo, os dados parciais do nível são transmitidos para o nível seguinte e o registrador do Nível 1 armazena os dados processados no intervalo anterior, como pode ser observado na Figura 4.13 (d). Nesse momento observa-se a troca da seleção da entrada do multiplexador do Nível 0. Como o registrador está responsável por manter o sinal para o nível seguinte, as unidades do Nível 0 podem ser desativadas.

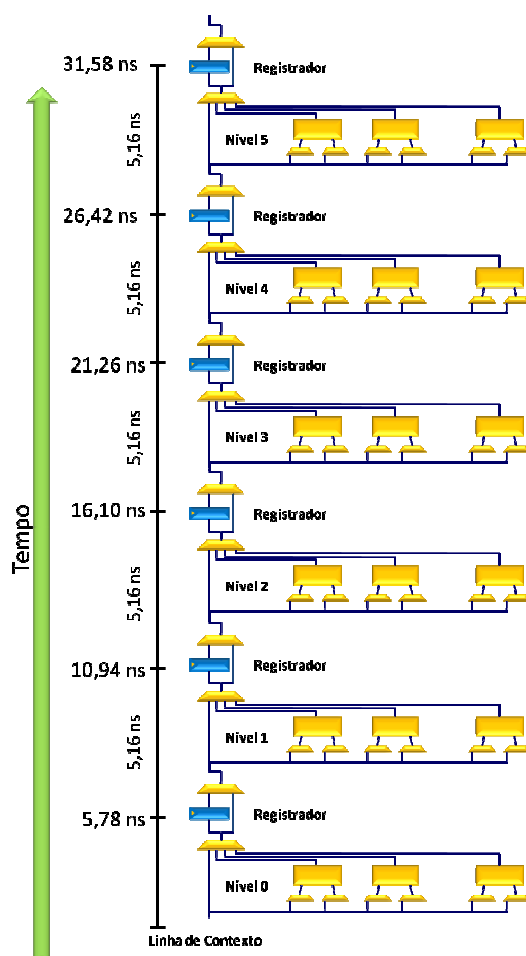


Figura 4.12: Matriz não virtualizada composta de ULAs.

No quarto intervalo observa-se a mesma dinâmica: o registrador do Nível 1 mantém o sinal estável, o registrador do Nível 2 armazena o sinal processado e o Nível 3 está processando. Ainda conforme a Figura 4.14 (a), os dados parciais são transmitidos para o Nível 4 e o Nível 1 não é mais necessário estar ativo, podendo, assim, ser desativado.

As Figuras 4.14 (b) e (c) correspondem ao quinto e sexto intervalo. Nestas figuras é possível observar o mesmo comportamento, o que possibilita que os níveis 2 e 3 sejam desativados. Na Figura 4.14 (d) a execução está completa e o sinal resultante pode ser obtido das linhas de contexto.

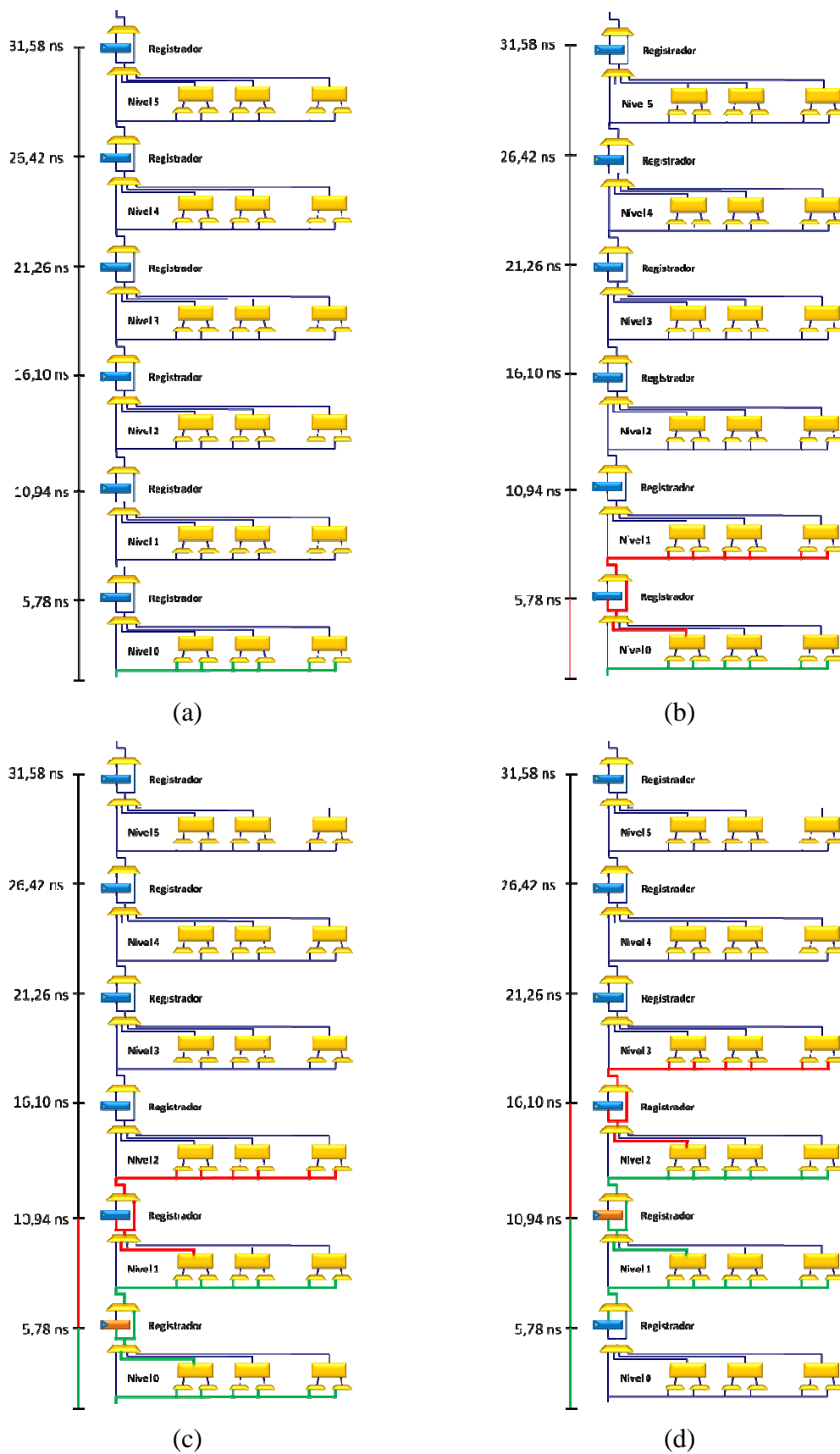


Figura 4.13: Execução de uma configuração de 6 níveis não virtualizada (parte inicial).

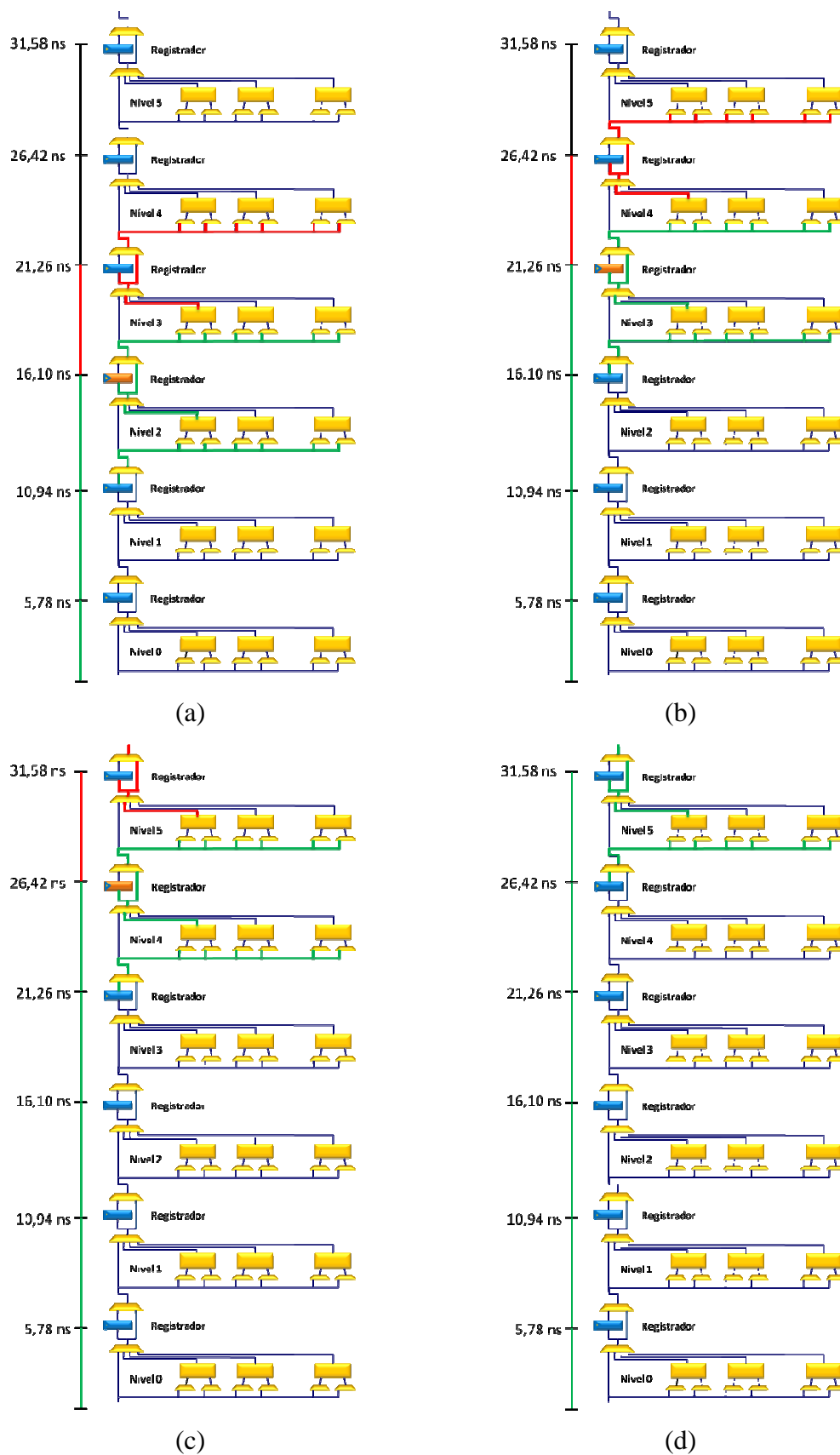


Figura 4.14: Execução de uma configuração de 6 níveis não virtualizada (parte final).

Entendido o funcionamento, pode-se aplicar a virtualização de hardware por meio do *pipeline* de estágios reconfiguráveis, agora, porém, sem a desvantagem da barreira temporal impedindo o avanço do sinal. Esse modelo foi denominado *pipeline* virtual - para exemplificar a técnica continuaremos utilizando o exemplo da matriz formada por ULAs.

Para que o funcionamento ocorra corretamente, é necessário que o circuito tenha um caminho combinacional que seja igual ou maior que o tempo de processamento de um nível. Isso garante que um nível não seja reutilizado antes que a execução termine e registre o resultado. O número de estágios necessários pode ser expresso como:

$$Nest = \lceil Test / DT \rceil \quad (\text{Eq. 9})$$

em que: Nest = Número de estágios necessários

Test = Tempo de execução de um estágio

DT = Diferença de tempo quando adicionado de um novo nível

Nesse exemplo o tempo de processamento tem atraso de 5,78 ns, enquanto DT é de 5,16 ns. Desse modo, dois níveis ativos são suficientes, pois o caminho combinacional é maior que o tempo de um nível combinacional (Figura 4.15).

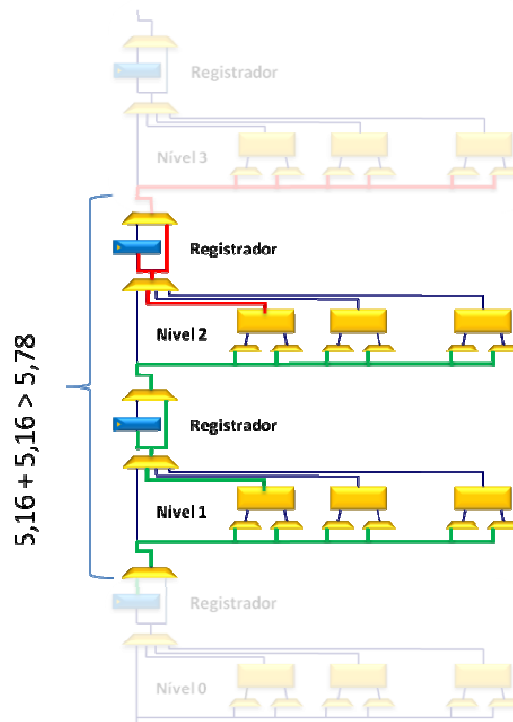


Figura 4.15: Caminho combinacional entre os estágios.

A virtualização do circuito pode ser encontrada na Figura 4.16. Para o exemplo adotado a virtualização necessita de 3 estágios. Como verificado anteriormente, são necessários que 2 estágios estejam ativos para que o caminho combinacional seja suficiente. Além disso, o terceiro nível é necessário, pois a reconfiguração é realizada em paralelo à execução, como apresentado na virtualização de estágios reconfiguráveis de 2 níveis.

O atraso de cada estágio é igual ao DT; já a virtualização de dois estágios é igual ao tempo de processamento de um nível. Como o DT determina o tempo de ciclo do

circuito, a técnica de pipeline virtual tende a ter maior desempenho em comparação à virtualização de estágios reconfiguráveis.

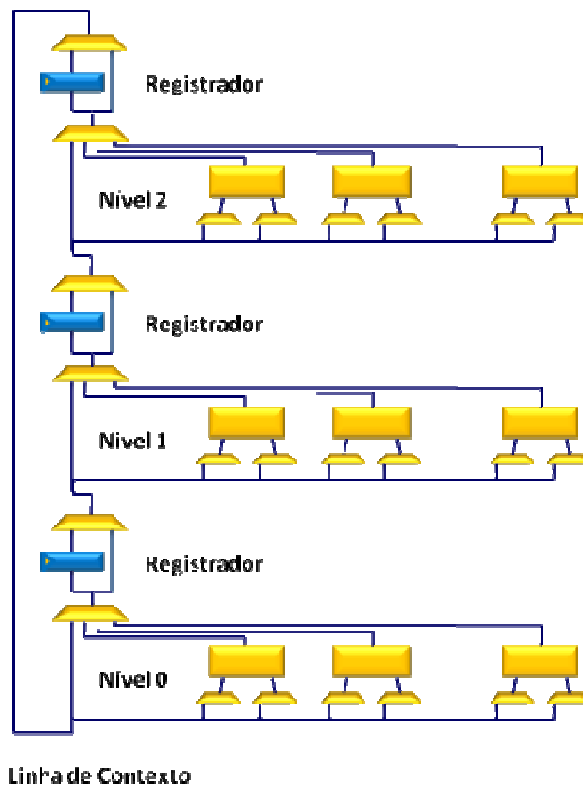


Figura 4.16: Virtualização por meio de 3 estágios físicos.

A mesma execução demonstrada nas Figuras 4.13 e 4.14 será demonstrada nas Figuras 4.17 e 4.18, contudo, de forma virtualizada.

Inicialmente é configurado o primeiro estágio (Figura 4.17 (a)); no próximo ciclo, o segundo estágio é configurado ao mesmo tempo que o primeiro estágio está em execução - Figura 4.17 (b). Nota-se que multiplexadores de saída estão com a saída combinacional selecionada durante esse momento.

No terceiro ciclo, o primeiro estágio conclui a execução e o resultado é armazenado no registrador do primeiro estágio; o terceiro estágio é configurado enquanto o segundo está em execução, como ilustrado na Figura 4.17 (c).

No quarto ciclo, o primeiro estágio é reconfigurado, mapeando a configuração do quarto nível virtual nesse estágio. Como o resultado processado anteriormente está registrado, pode-se reconfigurar o nível sem problemas. A saída do multiplexador do primeiro estágio tem como entrada o registrador - Figura 4.17 (d). Nesse mesmo ciclo é armazenado o resultado da execução do segundo estágio e o terceiro estágio permanece em execução.

No próximo ciclo, conforme a Figura 4.18 (a), o segundo estágio é reconfigurado mapeando a configuração do quinto nível virtual. A saída do multiplexador deste estágio tem como entrada ativa o registrador, enquanto o multiplexador do estágio anterior (Estágio 0) volta a ter a entrada combinacional como saída. Ainda no mesmo

ciclo, é armazenado o resultado da execução do terceiro estágio, e o primeiro estágio, que foi reconfigurado no ciclo anterior, permanece em execução.

Nos próximos ciclos a mesma sequência de ações pode ser observada- Figuras 4.18 (b) e (c). No oitavo e último ciclo, o resultado da execução está disponível (Figura 4.18 (d)).

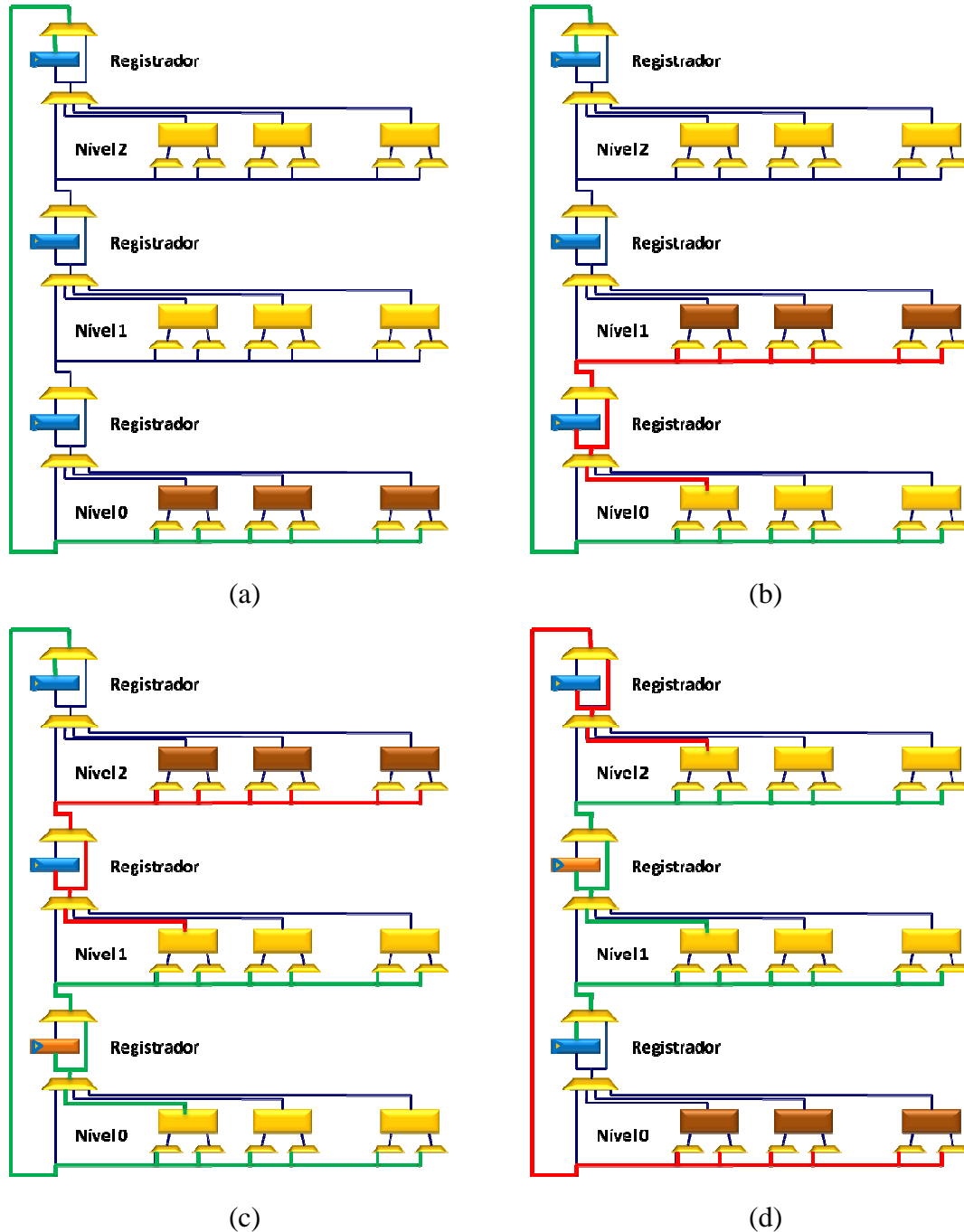


Figura 4.17: Execução de uma configuração de 6 níveis virtualizada em 3 estágios (parte inicial).

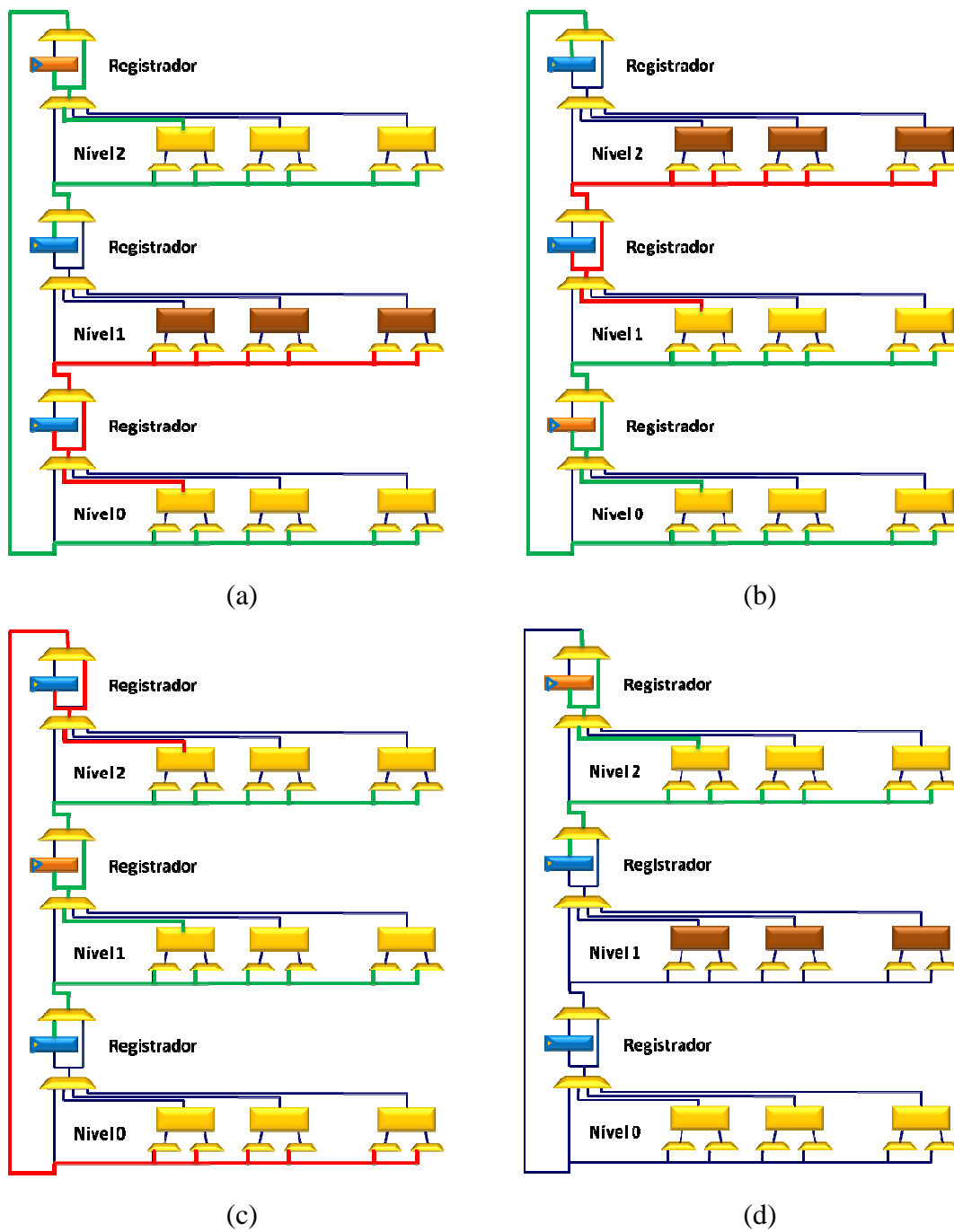


Figura 4.18: Execução de uma configuração de 6 níveis virtualizada em 3 estágios (parte final).

Na Figura 4.19 é possível visualizar o mapeamento dos níveis virtuais nos níveis físicos do exemplo apresentado nas Figuras 4.17 e 4.18. Observa-se que sempre dois níveis estão configurados e ativos, enquanto um terceiro é configurado.

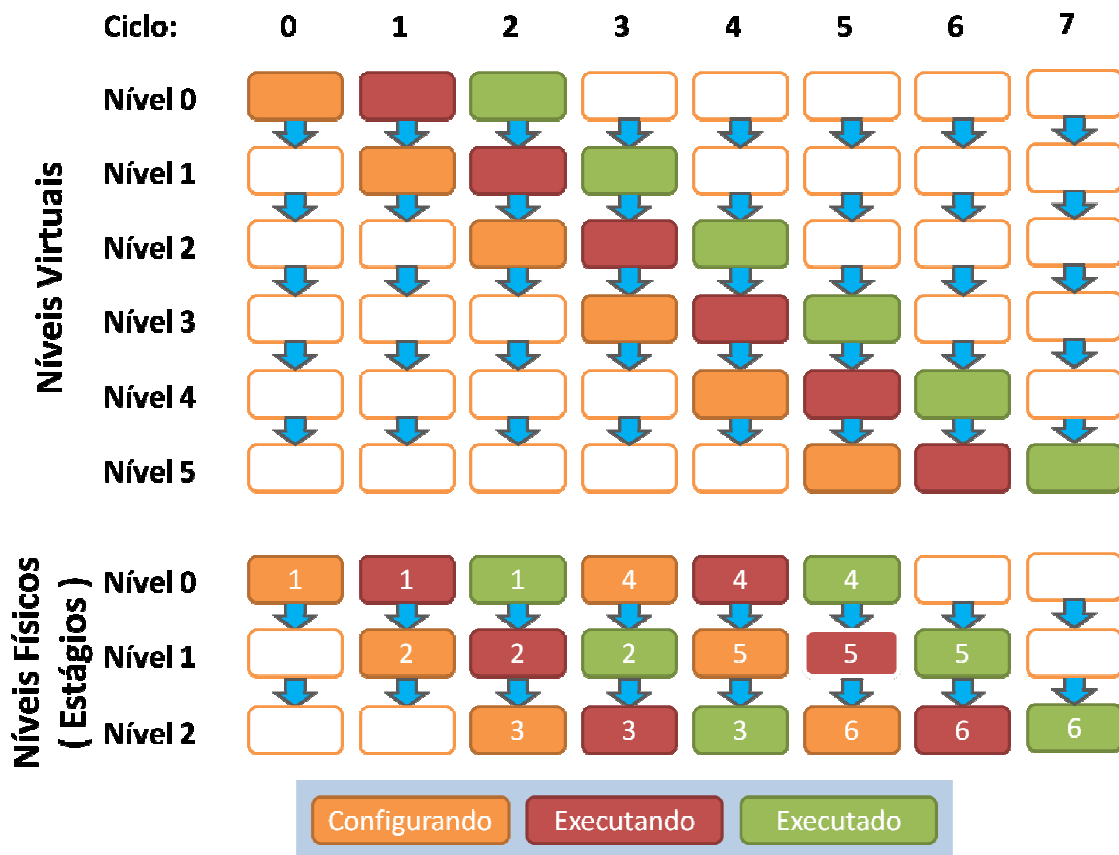


Figura 4.19: Mapeamento dos níveis virtuais nos níveis físicos (estágios).

Para o exemplo apresentado de *pipeline* virtual com 3 estágios, o controle do circuito é definido pelas seguintes operações:

- Busca da configuração do nível ($n+3$);
- Reconfiguração do nível ($n+2$);
- Execução dos níveis n e $n+1$;
- Registro do nível ($n-1$).

O controle pode variar de acordo com o número de estágios. Nos casos em que o caminho combinacional requer mais estágios ativos, mais estágios devem ser executados simultaneamente.

O tempo de execução é definido pela Equação 10, em que M corresponde ao número de níveis da configuração que será processada.

$$T_{ev} = (N_{est} + (M - 1)) \times DT \quad (\text{Eq. 10})$$

em que: T_{ev} = Tempo de execução do pipeline virtual

Nest = Número de estágios

M = Número de estágios

DT = Diferença de tempo quando adicionado de um novo nível

Tomando-se como exemplo o circuito de somadores *Ripple-Carry* em cascata, em que o atraso do somador é de 6,71 ns e DT é de 0,31 ns (Tabela 4.1), seriam requeridos 23 estágios para utilizar essa técnica de virtualização. Conforme observado anteriormente, o estágio não pode ser reconfigurado antes que sua execução seja finalizada, portanto, nesse exemplo seriam necessários 22 ciclos de execução, além de mais um de configuração. Assim, é inviável a utilização da virtualização, a não ser que o número de somadores em cascata fosse superior a 23.

Supondo que existissem 100 somadores em cascata, utilizando-se a virtualização de estágio reconfigurável, o tempo de execução seria de 671 ns, desconsiderando os atrasos do registrador - ao adotar a técnica de *pipeline* virtual, o tempo de execução seria de 37,82 ns. Contudo, a área do *pipeline* virtual seria aproximadamente 12 vezes maior, pela necessidade de apresentar 23 estágios físicos. Sendo assim, deve-se avaliar o custo-benefício que a aplicação da técnica de virtualização pode oferecer.

4.4 Resultados

Nessa seção são apresentados os resultados em relação à área ocupada e ao atraso ocasionado pela aplicação das técnicas de virtualização propostas. As técnicas de virtualização utilizadas foram correspondentes ao segundo modelo apresentado, que utiliza a técnica de *pipeline* com estágios reconfiguráveis com dois estágios, e o terceiro modelo, que utiliza a técnica de virtualização sem atraso da barreira temporal (*pipeline* virtual). O modelo do *pipeline* reconfigurável de um estágio não foi avaliado, já que esse modelo tinha o objetivo de introduzir a técnica de virtualização, que resulta na degradação do desempenho devido ao atraso adicional ocasionado pelo tempo de reconfiguração de cada ciclo.

Foram utilizadas as ferramentas da Mentor Graphics, Leonardo Spectrum para extrair a área e a frequência, e a ferramenta ModelSim para a simulação dos modelos propostos.

Inicialmente são apresentados os valores de atraso da implementação de uma matriz simples utilizada para validar a técnica. A matriz é composta de seis ULAs em paralelo por nível e seis linhas de contexto, mesmo exemplo utilizado anteriormente. A Tabela 4.4 mostra os resultados para uma matriz de seis níveis e a Tabela 4.5 apresenta os efeitos da matriz de 18 níveis.

Tabela 4.4: Atraso para a matriz de ULAs com 6 níveis.

Matriz com 6 níveis	Combinacional	Virtualização	
		Estágios reconfiguráveis	Pipeline virtual
Tempo de Execução (ns)	31,63	34,68	36,19

Tabela 4.5: Atraso para a matriz de ULAs com 18 níveis.

Matriz com 18 níveis	Combinacional	Virtualização	
		Estágios reconfiguráveis	Pipeline virtual
Tempo de Execução (ns)	93,67	104,04	98,06

Para a técnica de *pipeline* de estágios reconfiguráveis foram necessários 2 estágios; já para a técnica de *pipeline* virtual foram necessários 3 estágios.

É interessante observar que o tempo de computação da virtualização com a técnica de *pipeline* de estágios reconfiguráveis de 2 estágios na matriz com 6 níveis é menor que com 3 estágios, ao passo que na matriz com 18 níveis o tempo de computação é o inverso. Isso ocorre porque na virtualização com 3 níveis é necessário um ciclo a mais, porém o período de ciclo da virtualização é menor. Para esse exemplo, a inversão ocorre quando a matriz possuir mais de 8 níveis - Figura 4.20 - em que são apresentados os valores de tempo de computação em função do número de níveis.

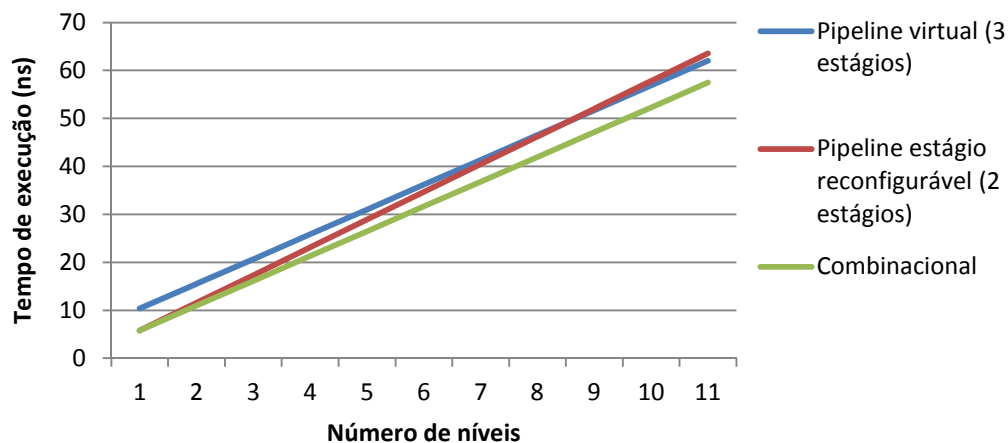


Figura 4.20: Tempo de execução em função do número de níveis.

Na Figura 4.21 é possível observar que, com o aumento de níveis, a virtualização com a técnica de *pipeline* virtual se aproxima do tempo de computação de uma matriz totalmente combinacional. Essa aproximação se deve à amortização do ciclo extra ao longo do tempo (como na Figura 4.22, em que é apresentada a relação entre o tempo de computação dos modelos virtualizados e o circuito combinacional).

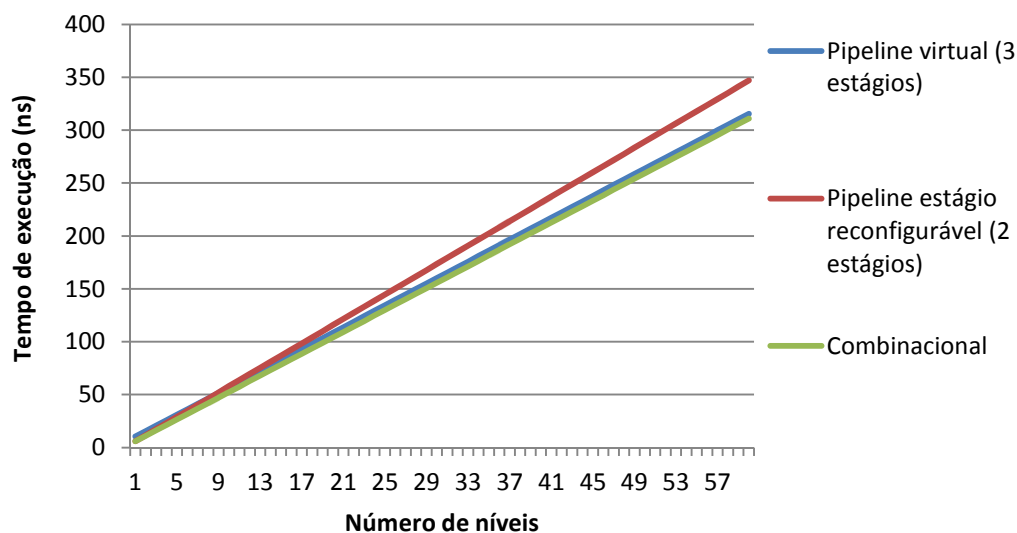


Figura 4.21: Tempo de execução em função do número de níveis.

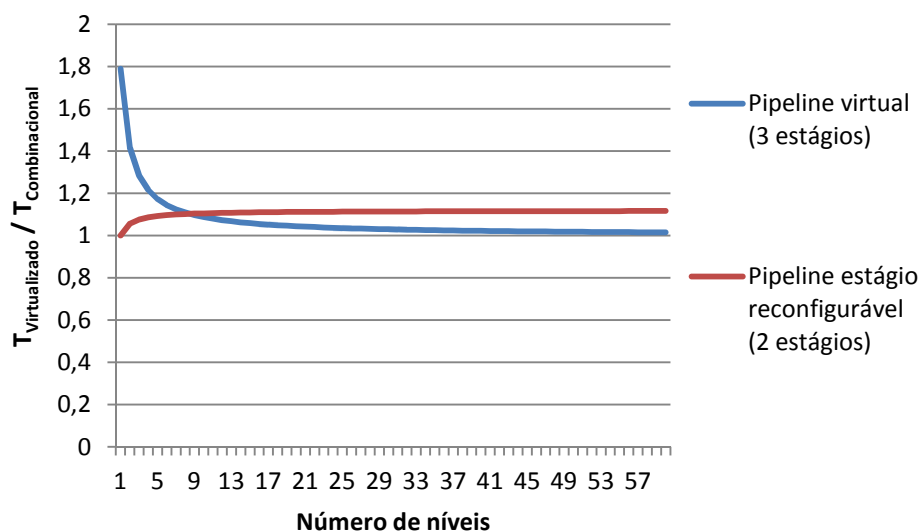


Figura 4.22: Relação entre o tempo de computação das técnicas de virtualização e o tempo de computação do circuito combinacional.

Na Figura 4.22 observa-se que o modelo do *pipeline* virtual inicialmente é mais lento que o modelo do *pipeline* de estágio reconfigurável. Quando a configuração possui 4 níveis, o tempo de computação do modelo virtual é 21,5% maior que o circuito combinacional, já o outro modelo é apenas 6,8% maior. As curvas se cruzam quando o número de níveis é igual a oito, ponto no qual o tempo de computação das técnicas de virtualização é superior em 10% ao tempo do circuito combinacional. Após esse ponto, o tempo de computação do modelo do *pipeline* virtual é inferior ao outro modelo. Conforme se incrementa o número de níveis o modelo do *pipeline* virtual, esse tende a apresentar o tempo de computação muito próximo ao do circuito combinacional, apresentando uma diferença de tempo menor que 2% quando possui 44 níveis. Portanto, para este exemplo quando é utilizado até oito níveis é melhor utilizar a abordagem de

pipeline com estágios reconfiguráveis. A transição da abordagem de *pipeline* de estágio reconfigurável para *pipeline* virtual pode ser realizada facilmente, bastando manter a seleção do multiplexador de forma que a entrada selecionada seja o registrador, como na Figura 4.11(c). Sendo assim, pode-se decidir em tempo de execução, baseando-se no número de níveis, qual melhor forma para processar uma determinada configuração.

Na Figura 4.23 é possível observar a vantagem que se tem ao aplicar a virtualização em relação à área ocupada. Se por um lado a área utilizada nos estágios é constante, a área da matriz combinacional aumenta de acordo com o incremento de níveis.

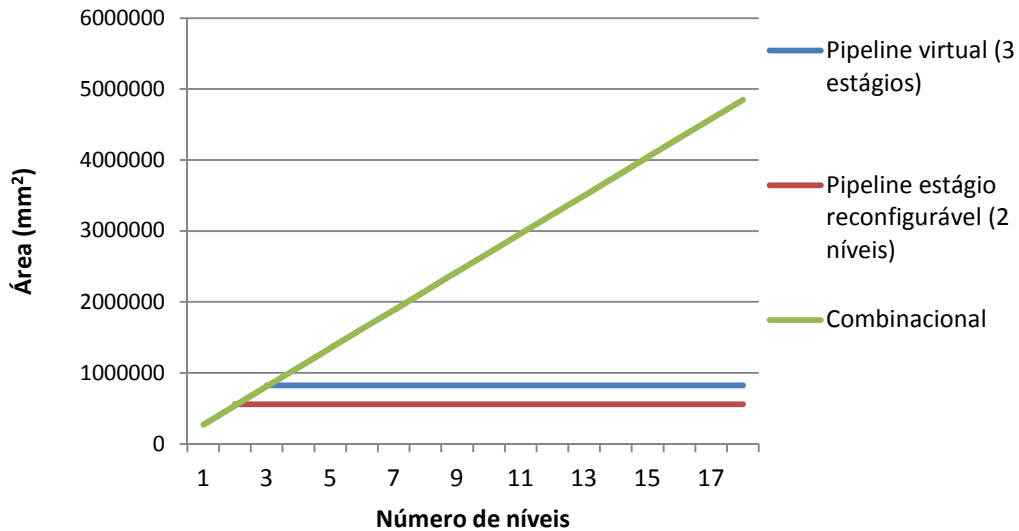


Figura 4.23: Comparativo da área ocupada entre as técnicas de virtualização em função do número de níveis.

Outro experimento realizado utilizou a arquitetura reconfigurável apresentada no Capítulo 3. Para efeito de comparação foram adotadas duas configurações da matriz reconfigurável citadas no trabalho de Rutzig (2008), cuja obra também apresenta uma técnica de redução de área.

Apesar de a matriz reconfigurável ser composta de elementos combinacionais, as unidades de *Load/Store* impõem implicitamente uma barreira temporal, pois diferentemente das ULAs e dos multiplicadores, essas unidades não podem iniciar a execução com dados parciais. (*E.g.*, o endereço e o dado a ser gravado devem estar estáveis para que a operação ocorra corretamente). Para o modelo de *pipeline* de estágio reconfigurável essa questão não é relevante, visto que o registrador entre os estágios já impõe uma barreira temporal. Todavia, para o modelo do *pipeline* virtual, que não possui uma barreira temporal, as unidades de *Load/Store* devem ser projetadas levando em consideração essa imposição.

Originalmente, para se contabilizar o tempo de computação da matriz reconfigurável considera-se que cada nível é executado em um ciclo, ou seja, existe uma barreira temporal implícita e cada nível tem determinado instante para iniciar e finalizar. Desta forma, durante a operação das unidades de *Load/Store* os dados sempre estarão estabilizados.

Da forma que a matriz original foi modelada, o ganho combinacional é aproveitado somente nos intraníveis, e não como um circuito totalmente combinacional, portanto, existe ainda um espaço de projeto a se explorar em trabalhos futuros. A técnica de

pipeline virtual pode ser utilizada para explorar novos modelos da arquitetura e possibilitar maiores ganhos de desempenho por meio do ganho combinacional, já que permite a virtualização sem a perda da propriedade combinacional do circuito.

Para que o modelo do *pipeline* virtual funcione corretamente, o tempo de uma unidade de *Load/Store* foi alterado para ter uma latência de dois ciclos (modificação necessária para que os sinais de entrada da unidade se estabilizem antes de se executar a operação). A alteração na arquitetura pode ocasionar perda de desempenho porque leva um ciclo adicional para execução. As simulações de desempenho não foram realizadas para esse modelo modificado da arquitetura.

Para o modelo de *pipeline* de estágio reconfigurável, como na matriz original, cada nível tem o tempo de computação igual a um ciclo, porém o registrador na saída de cada estágio adiciona um atraso de 1% no período do ciclo.

Na Tabela 4.6 é mostrada a primeira configuração da matriz reconfigurável que, nesse caso, é extremamente grande devido à habilidade de extrair todo potencial da técnica da arquitetura reconfigurável, como demonstrado em Beck (2005).

Na Tabela 4.7 são apresentados os valores de aceleração e área da matriz reconfigurável original, quando utilizada a técnica que obtém os melhores resultados com a ferramenta ARISE (baixo custo) (RUTZIG, 2008) e quando aplicada a virtualização. Também são mostrados valores de área relativa ao MIPS R3000 e à porcentagem de redução alcançada.

Tabela 4.6: Configuração 1 da matriz reconfigurável.

<i>Array Original</i>				
# Níveis	#Total Colunas	#ULA/Linha	#Mul/Ciclo	#Load/Ciclo
85	185	50	15	120

Tabela 4.7: Resultado para a configuração 1.

	<i>R3000</i>	<i>Array Original</i>	<i>ARISE (Baixo custo)</i>	<i>Pipeline virtual (3 estágios)</i>	<i>Estágios reconfiguráveis (2 estágios)</i>
Área (Gates)	26.886	63.544.215	6.121.960	1.495.158	996.772
Aceleração	1	2,85	2,85	*	2,82
Área em relação ao R3000	-	2.363,46	227,70	83,41	55,61
Redução (%)	-	-	0,904	0,964	0,976

(*) Não foram realizadas simulações de desempenho para esse modelo.

Analisando-se a Tabela 4.7, é possível constatar que tanto a ferramenta ARISE quanto a virtualização conseguiram ganhos extremamente significativos, todos acima de 90% de redução. Tal resultado se deve principalmente ao fato de que a configuração da matriz reconfigurável é extremamente grande. Como a técnica de virtualização diminuiu

o número de níveis em poucos estágios, os 85 níveis foram reduzidos a 2 ou 3 estágios, representando aproximadamente 2,4% e 3,5% da área do circuito original. A aceleração obtida no modelo de *pipeline* de estágios reconfiguráveis foi de aproximadamente 99% da aceleração atingida com a matriz original.

Na Tabela 4.8 é apresentada uma configuração da matriz reconfigurável mais factível de fabricação. Os resultados correspondentes dessa configuração são apresentados na Tabela 4.9.

Já na Tabela 4.9 observa-se que, quando o tamanho da matriz reconfigurável diminui, a eficiência da ferramenta ARISE também cai, pois essa ferramenta tem como base a retirada de unidades funcionais pouco utilizadas, avaliando-se o conjunto de aplicações definido. Entretanto, com a utilização das técnicas de virtualização, a redução foi superior a 94% para ambas as configurações, bem superior ao ARISE que apresentou 41%. A aceleração obtida no modelo de *pipeline* de estágios reconfiguráveis foi de aproximadamente 99% da aceleração atingida com a matriz original.

Tabela 4.8: Configuração 2 da matriz reconfigurável.

<i>Array Original</i>				
# Níveis	#Total Colunas	#ULA/Linha	#Mul/Ciclo	#Load/Ciclo
50	20	12	2	6

Tabela 4.9: Resultados para configuração 2.

	<i>R3000</i>	<i>Array Original</i>	<i>ARISE (Baixo custo)</i>	<i>Pipeline virtual (3 níveis)</i>	<i>Estágios reconfiguráveis (2 níveis)</i>
Área (Gates)	26.886	6.774.500	2.777.545	270.980	180.653
Aceleração	1	2,56	2,56	*	2,53
Área em relação ao R3000	-	251,97	103,30	15,1	10,8
Redução (%)	-	-	41	94	96

(*) Não foram realizadas simulações de desempenho para esse modelo.

Como ilustrado anteriormente, a técnica de virtualização permite virtualizar quantos níveis forem necessários, e o número máximo de níveis que podem ser executados é limitado apenas pela memória que armazena as configurações, sendo assim, foi simulado uma matriz reconfigurável com o número de níveis infinito. Para avaliar o ganho em aceleração dessa matriz, considerou-se que o conjunto de unidades funcionais de cada nível é o mesmo das Configurações 3 e 4. O mesmo conjunto de *benchmarks* foi executado para avaliar os resultados.

A Figura 4.24 apresenta a aceleração obtida para cada aplicação do *Mibench* com número de níveis ilimitados. O resultado pode ser comparado com os gráficos das Figuras 3.11 e 3.12. As principais aplicações que se beneficiaram foram a *RjindaelD* e *RjindaelE*, com ganho de aproximadamente de 130% e 100% na aceleração, quando comparado com a Configuração 4, a qual obtém o melhor resultado de aceleração.

Na Figura 4.25 é demonstrada a aceleração média para o conjunto de aplicações do *Mibench* para as Configurações 1, 2, 3, 4 e a com níveis infinitos. A aceleração média para o conjunto de aplicações foi de aproximadamente 13% a mais que Configuração 4, tendo como principal influência os ganhos das aplicações *RjindaelD* e *RjindaelE*. Com o número de níveis ilimitados, maiores trechos das aplicações podem ser executados na matriz e, conseqüentemente, um maior desempenho é alcançado.

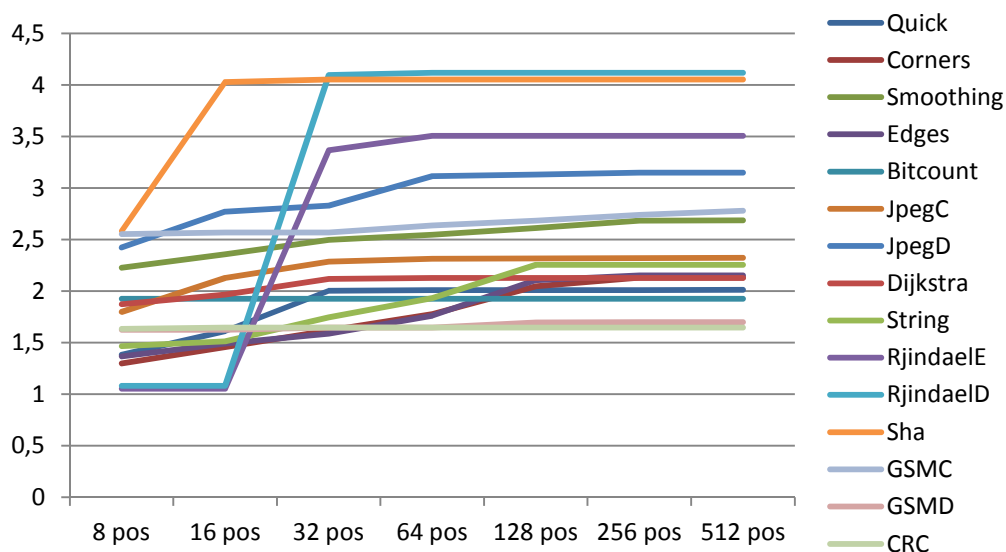


Figura 4.24: Aceleração para a matriz com número de níveis ilimitados para cada aplicação do *Mibench*.

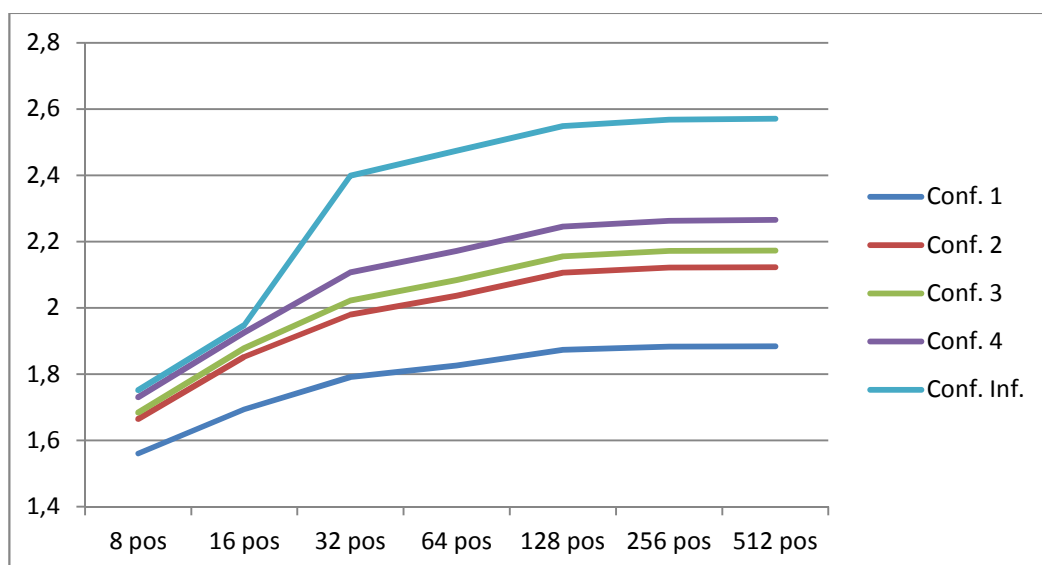


Figura 4.25: Aceleração média para cada configuração da matriz reconfigurável.

Outro ganho potencial que explora a ideia de níveis ilimitados por meio da virtualização está relacionado à inserção de novas aplicações que antes não foram previstas. Se elas utilizarem uma maior quantidade de níveis inicialmente projetado, espera-se um ganho na aceleração comparado a uma matriz de altura limitada.

Outro benefício da aplicação dessa técnica está associado à eficiência na utilização das unidades funcionais. No melhor caso, em que a ferramenta ARISE encontra a forma da matriz ideal para uma aplicação específica, a eficiência de utilização ainda é baixa. Supondo-se que a matriz possua 50 níveis e levando-se em consideração que uma unidade funcional precisa de uma unidade de tempo para executar, as unidades ficam ociosas por 49 unidades de tempo, logo, a eficiência é de 2% mesmo quando todas as unidades são utilizadas. Isso ocorre porque as unidades funcionais são utilizadas somente uma vez a cada configuração. Aplicando a técnica de virtualização, o reuso das unidades será mais bem explorado; assim, a eficiência no uso das unidades funcionais será maior.

As técnicas de virtualização apresentadas requerem uma forma fragmentada de busca dos bits de configuração, em que a memória de contexto deve ter a capacidade de disponibilizar, de alguma forma, os bits de configuração correspondentes ao nível desejado. Caso a memória não possua essa capacidade, é possível implementar essa funcionalidade por meio de registradores de deslocamento ou pelo acesso a conjuntos de bits do registrador por meio de multiplexadores.

No Capítulo 5 será demonstrada uma forma de busca parcial da configuração. Além disso, será realizado um estudo a fim de minimizar a energia consumida para essa importante operação em arquiteturas reconfiguráveis.

5 MECANISMOS DE GERENCIAMENTO DA MEMÓRIA DE CONTEXTO

As arquiteturas reconfiguráveis são compostas de unidades reconfiguráveis, que podem implementar operações ou funções lógicas e possuir suas interconexões programadas. As configurações são armazenadas na memória de contexto (composta por uma sequência definida de bits a qual determinará a funcionalidade do circuito). A memória de contexto é um das maiores responsáveis pela flexibilidade das arquiteturas reconfiguráveis.

Mesmo em uma arquitetura de grão grosso, o número de bits necessários para armazenar a configuração é muito grande. Geralmente, os bits de configuração são armazenados em uma memória intrachip e, apesar de local, essa memória é responsável por grande parte do consumo de energia total do sistema. Para reduzir este consumo deve-se minimizar o consumo de todos os componentes do sistema que impactam de alguma forma no tamanho da memória de contexto. Outra forma de minimizar o consumo de energia é tornar a busca das configurações o mais eficiente possível modificando a maneira como as configurações são acessadas na memória de contexto.

Com este objetivo, serão apresentadas as técnicas desenvolvidas que exploram o espaço de projeto da memória de contexto. Ao final será analisado o impacto da memória de contexto no sistema reconfigurável.

5.1 Explorando grão de acesso a memória

Um aspecto importante de toda memória é que a energia dissipada é uma função de seu tamanho e do número de bits de entrada/saída. Na verdade, o número de bits de entrada/saída tem grande relevância, pois cada bit de saída necessita de circuitos adicionais de buffer e de um amplificador de sinal, mesmo quando as memórias são incorporadas dentro do chip. Assim, quanto maior for o número de bits de entrada/saída, ou seja, a largura da porta da memória, maior será a energia consumida durante o acesso aos dados.

Uma maneira de reduzir a energia durante um acesso é reduzir a largura da porta da memória. No entanto, é preciso aumentar o número de acessos à memória para carregar as mesmas informações. A estratégia é equilibrar corretamente a quantidade de acessos à memória com o número correto de bits de saída de memória com o intuito de obter o menor consumo de energia. Para isso, o trabalho apresenta um modo de como encontrar

a granularidade ideal para acessar a memória de contexto, variando a quantidade de bits de configuração por acesso e alterando a largura da porta de memória.

A Figura 5.1 demonstra a técnica de divisão da memória. Nesse exemplo considera-se uma memória de contexto com capacidade de armazenar quatro configurações, sendo que cada configuração é composta de quatro níveis. Para avaliar a energia total consumida na busca da configuração, divide-se a largura (número de bits) da porta memória pela metade (Figura 5.1(b)) e por quatro (Figura 5.1(c)). Como o tamanho total da memória é mantido constante, o número de entradas da memória escala de acordo com a redução da largura da porta da memória. Assim, o objetivo é encontrar a melhor forma de acesso à memória do contexto, a fim de minimizar o consumo de energia.

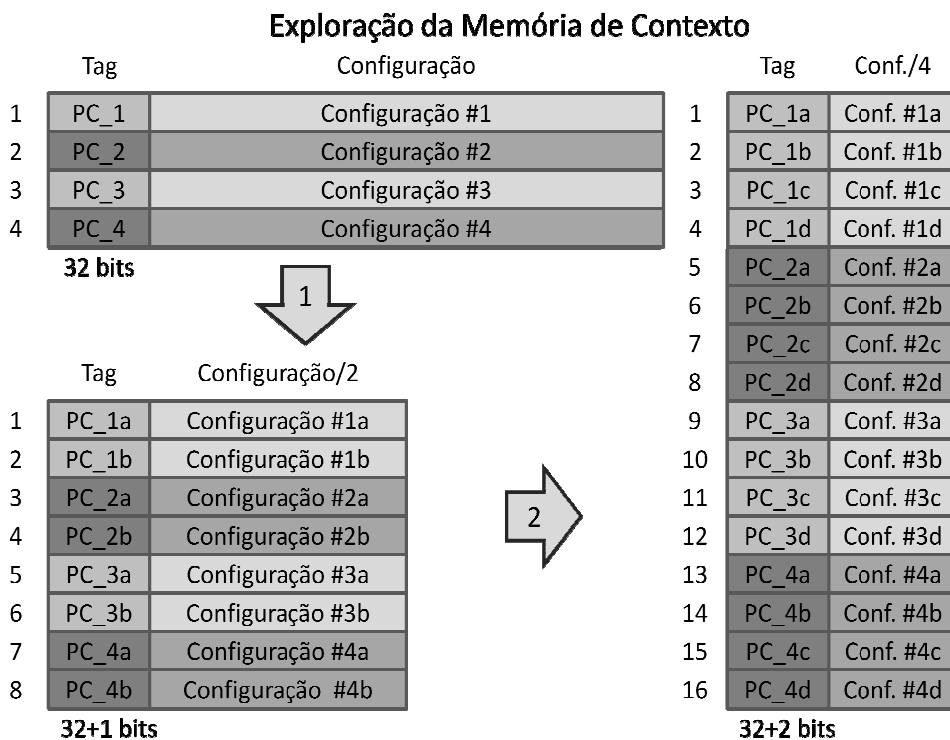


Figura 5.1: Exploração da memória de contexto dividindo a configuração em partes.

Aplicando a estratégia descrita acima, dois extremos podem ser observados. O primeiro caso ocorre quando a configuração é carregada em apenas um ciclo e todos os bits de configuração são acessados em paralelo uma única vez (Figura 5.1(a)). Por conseguinte, uma grande porta de memória é necessária e grande quantidade de energia é consumida por acesso. Em outro extremo, apenas um nível da matriz reconfigurável é buscado por ciclo, necessitando de vários ciclos. Nesse último caso, a energia consumida é a soma das energias consumidas a cada ciclo (Figura 5.1(c)).

A largura mínima é determinada pela quantidade de bits para configurar um nível - caso contrário o desempenho teria impacto negativo. Como cada nível é executado em um ciclo, é necessário que o tempo de na configuração dos níveis tenha de ser menor ou igual da execução, assim, garante-se que o nível esteja configurado antes da execução.

Explorando o espaço de projeto pode-se encontrar a melhor divisão para diferentes tamanhos da matriz reconfigurável. Neste trabalho foram consideradas quatro diferentes configurações para a matriz reconfigurável, variando o número de níveis e de unidades funcionais.

Para a exploração do espaço de projeto de memória de reconfiguração levou-se em conta as quatro configurações anteriormente apresentadas na Tabela 3.2. O tamanho escolhido para a memória de contexto é de 64 entradas, pois o acréscimo de mais entradas não apresenta ganhos relevantes para as aplicações *Mibench*. Os dados de energia consumida pelas memórias foram extraídos com a ferramenta Cacti (MURALIMANO HAR, 2007). Para que o sistema funcione corretamente, o tempo de acesso à memória de contexto deve ser menor que o período do ciclo do processador, portanto, uma restrição de tempo de acesso foi adicionada como prioridade na ferramenta para todas as modelagens de memória exploradas. A tecnologia escolhida foi de 90 nm.

Ao aplicar o processo de divisão da largura da porta de memória obtiveram-se quantidades diferentes de energia por acesso. A Figura 5.2 apresenta a energia total consumida no acesso a uma configuração completa com diferentes larguras de porta. O grão de acesso varia da seguinte maneira: na primeira coluna o acesso à memória corresponde à quantidade de bytes necessários para reconfigurar um nível por acesso; na segunda coluna corresponde ao acesso à memória com largura de dois níveis, na terceira, quatro níveis, e assim por diante. O número de bytes acessados define a largura da porta da memória. Como apresentado na Figura 5.2, em alguns casos a energia de acesso é maior para memórias com largura de porta menores, fato que ocorre quando a energia consumida com o aumento da profundidade supera a energia conservada com a redução da largura da porta. Cabe lembrar que a capacidade de armazenamento da memória é mantida constante para cada configuração avaliada.

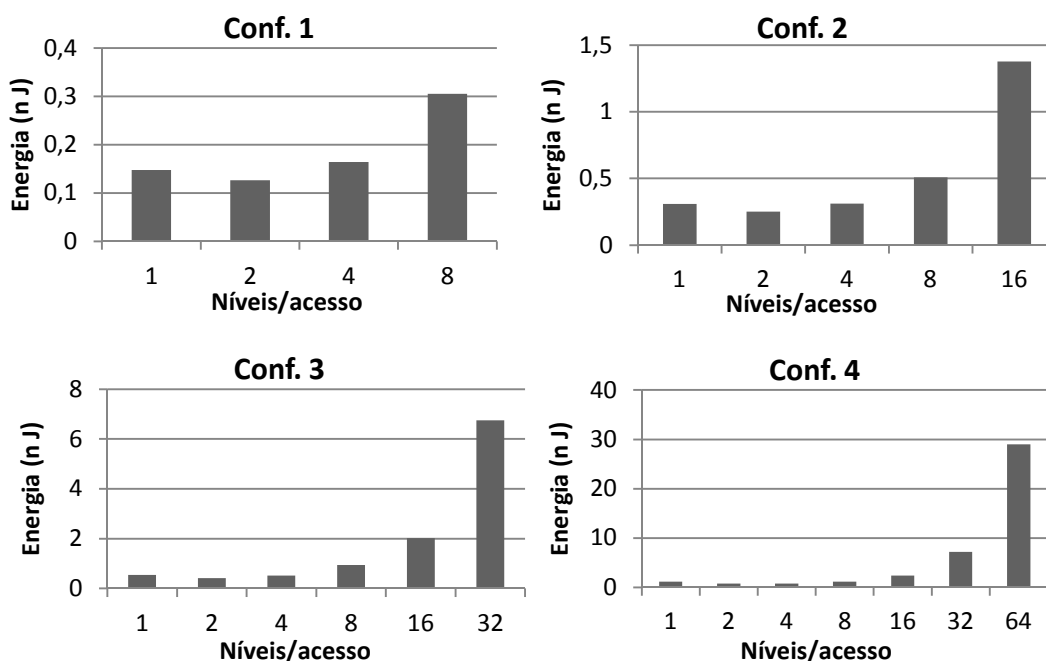


Figura 5.2: Energia total de acesso a uma configuração com diferentes larguras de porta.

Como apresentado na Figura 3.14, as configurações geradas para o conjunto de aplicações analisadas variam muito. Muitas configurações possuem poucos níveis, ao passo que outras utilizam todos os níveis disponíveis na matriz reconfigurável. Logo, será analisado o consumo de energia na busca de configurações de duas formas: Configuração total e Configuração parcial.

5.1.1 Configuração total

Na configuração total assume-se que a configuração deve ser completamente carregada, pois muitas arquiteturas reconfiguráveis podem não suportar reconfiguração parcial. Independentemente do número de níveis da configuração, a busca total dos bits de configuração é realizada. O número de bits carregados varia de acordo com a estrutura da matriz reconfigurável, apresentada na Tabela 3.2.

A Figura 5.3 mostra a quantidade de energia consumida para carregar uma configuração com diferentes grãos de acesso. A energia de acesso é obtida por meio da soma dos acessos para carregar toda a configuração (cada segmento da coluna corresponde a um acesso à memória).

Ainda na Figura 5.3 pode-se observar que, para cada configuração, existe uma largura de porta ótima que minimiza a energia consumida nos acessos à configuração. Para a Configuração 1 não houve economia de energia, visto que a melhor forma de acesso ocorreu quando a configuração foi carregada em uma só vez, incluindo os oito níveis.

Por outro lado, para as Configurações 2, 3 e 4, a melhor opção é realizar o acesso com a largura de oito níveis. No caso da Configuração 2, por exemplo, dois acessos de oito níveis seriam necessários para carregar uma configuração inteira, consistindo na melhor opção em relação ao consumo de energia. Na Configuração 4, embora haja uma diferença mínima entre acessar oito ou dezesseis níveis por ciclo, a primeira ainda pode ser considerada a melhor escolha em termos de energia, embora leve mais ciclos para buscar toda a configuração. Para as Configurações 2, 3 e 4, a energia economizada foi respectivamente de 26%, 44% e 67%, quando comparada à energia consumida pela busca de toda a configuração em um acesso à memória de contexto.

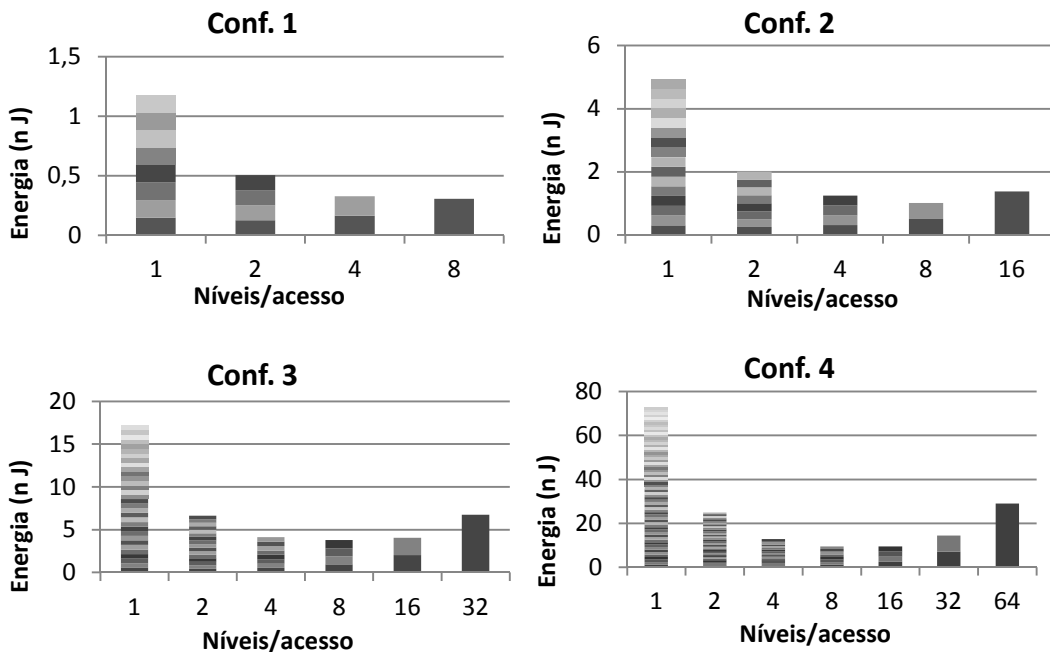


Figura 5.3: Consumo de energia para carregar uma configuração com diferentes grãos de acesso.

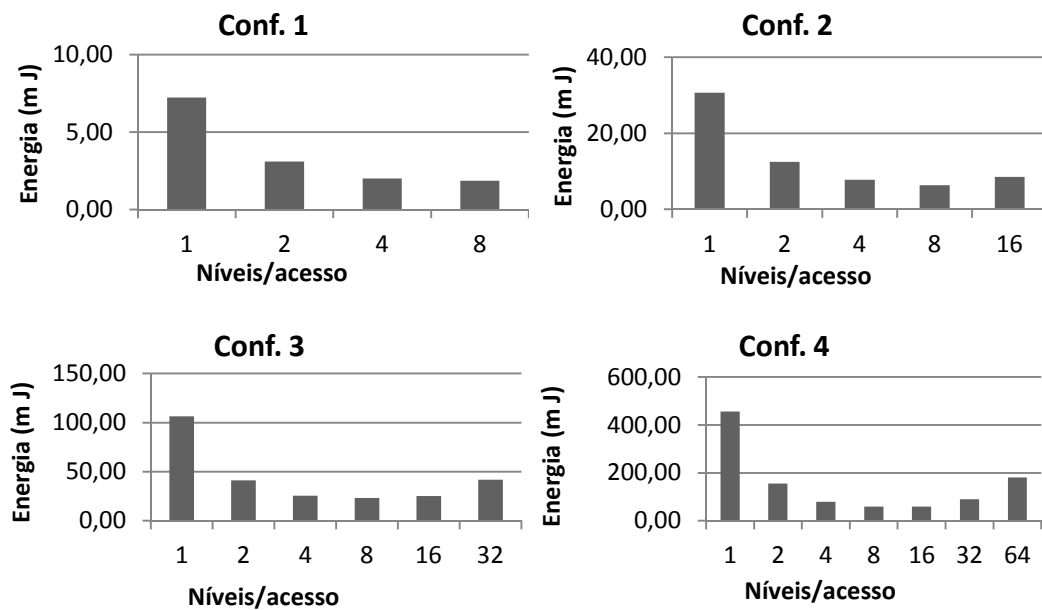


Figura 5.4: Energia consumida com diferentes grãos de acesso utilizando reconfiguração total.

A Figura 5.4 mostra o consumo total de energia da memória na execução de todo o *Mibench*, quando utilizada a reconfiguração total com diferentes grãos de acesso. Como mostrado na Figura 5.4, na maioria dos casos, exceto para a Configuração 1, os resultados sugerem que se deve fazer mais acessos à memória de contexto com uma memória de largura de porta menor. Contudo, os sistemas reconfiguráveis que precisam da configuração completa para estarem aptos a iniciar execução poderiam apresentar perda de desempenho ao se utilizar essa abordagem, uma vez que mais acessos à memória devem ser realizados para carregar a configuração completa. Este fato não ocorre com a arquitetura reconfigurável utilizada nesses experimentos.

A arquitetura utilizada pode ser configurada parcialmente porque, enquanto um nível está sendo executado, a próxima configuração é carregada. Assim sendo, nenhum atraso adicional é inserido e, portanto, o desempenho é mantido. Essa abordagem é muito semelhante ao processo de virtualização utilizado no sistema reconfigurável PipeRench e pode ser aplicado a outros sistemas reconfiguráveis.

5.1.2 Configuração parcial

Na configuração parcial somente os segmentos da configuração que serão utilizados efetivamente serão carregados; mesmo assim, em alguns casos mais bits de configuração podem ser buscados. Assim, se o TB gerar certa configuração de sete níveis e a memória de reconfiguração tem uma largura de porta (que corresponde ao tamanho de cada segmento) capaz de carregar quatro níveis por acesso, então dois acessos serão necessários, buscando oito níveis.

Dessa forma, para analisar o consumo de energia é necessário inspecionar a execução de todos os *benchmarks*. Na Figura 5.5 é apresentada a energia total consumida utilizando a reconfiguração parcial com diferentes grãos de acesso.

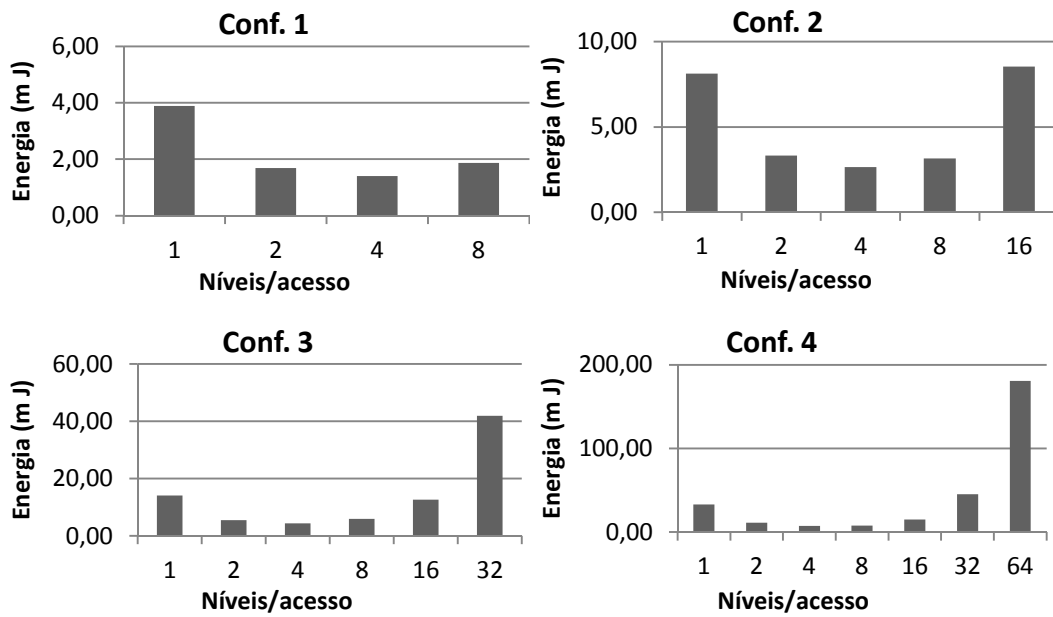


Figura 5.5: Energia consumida com diferentes grãos de acesso utilizando a reconfiguração parcial.

A Figura 5.5 apresenta uma redução de energia de 25%, 69%, 90% e 96%, respectivamente para as Configurações de 1 a 4, quando comparado à memória de contexto original. O resultado pode ser explicado pelo fato de que a maior parte das configurações executadas não utiliza muitos níveis. Por conseguinte, com a reconfiguração parcial menos dados inúteis são carregados da memória, economizando energia no carregamento da configuração.

A Tabela 5.1 mostra a área (em mm^2) ocupada pela memória de contexto para as quatro matrizes reconfiguráveis propostas na Tabela 3.2, variando a largura da porta de acordo com o número de bits necessários para configurar 1, 2, 4 níveis e assim por diante. Nesta tabela pode ser observado que a largura da porta impacta diretamente na área ocupada, pois quanto maior o número de bits de saída maior é o número de amplificadores de sinal e buffers de saída. Caso a área seja um fator limitante, memórias com porta de saída com largura menor necessitam de menor área. Porém, como analisado anteriormente, a energia consumida pode não ser minimizada. Essa é mais uma decisão de projeto que deve ser verificada.

Tabela 5.1: Área da memória de contexto (em mm^2) para as diferentes configurações com distintas larguras de porta.

Largura da porta	1	2	4	8	16	32	64
------------------	---	---	---	---	----	----	----

(níveis)								
Conf. 1	2,012	4,3161	17,226	65,801	-	-	-	
Conf. 2	3,6379	7,0421	26,205	98,107	384,92	-		
Conf. 3	6,8901	14,507	39,705	144,59	562,35	2231,4	-	
Conf. 4	11,642	18,61	43,815	159,99	566,85	2235,5	8906,1	

Uma desvantagem dessa abordagem é a fragmentação externa. Como uma configuração é quebrada em partes e somente os níveis que serão utilizados são armazenados, endereços da memória reservados para aquela configuração não são utilizados.

A Figura 5.6 mostra um exemplo de uma memória de contexto capaz de armazenar quatro configurações, em que cada configuração armazenada é dividida em quatro segmentos. Se a arquitetura possui dezesseis níveis, cada segmento armazena a configuração de quatro níveis.

Nesse exemplo é ilustrada a alocação da memória da seguinte forma: a primeira configuração armazenada possui até quatro níveis, já que ocupa somente um segmento. A segunda possui até oito níveis, a terceira até quatro níveis e a quarta até doze níveis. Tal alocação é ordinária, porque, como apresentado anteriormente, grande parte das configurações geradas pelo TB utilizam poucos níveis.

Por meio desse exemplo podemos observar a fragmentação externa da memória. Mais de 50% dela não é utilizada e, mesmo que se existisse mais uma configuração para ser inserida, esses segmentos livres não poderiam ser utilizados, pois cada segmento é reservado para uma configuração específica.

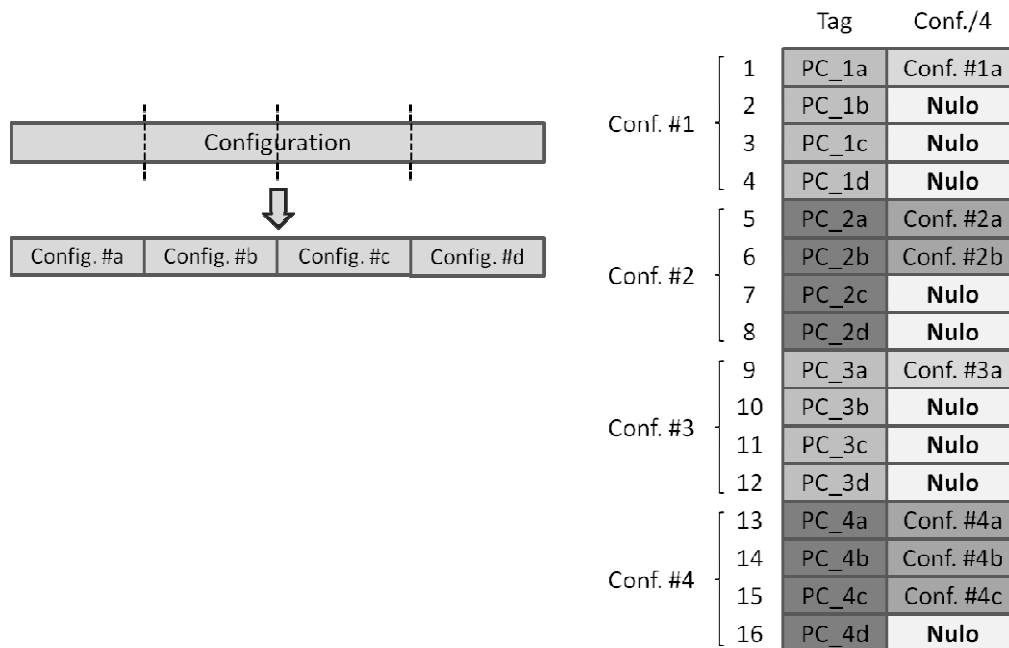


Figura 5.6: Fragmentação externa da memória de contexto.

5.2 Busca sob demanda

A abordagem anterior mostra que a busca parcial da reconfiguração pode obter grande economia de energia. A técnica apresentada é simples e eficaz, podendo ser avaliada em outras arquiteturas reconfiguráveis que suportam essa forma de busca de configurações.

Em arquiteturas em que o tamanho das configurações varia muito, a fragmentação externa pode ser muito elevada, e por consequência, apresentar um uso pouco eficiente da memória de contexto, desperdiçando grande quantidade de recursos que muitas vezes são limitados.

Na arquitetura do estudo de caso, a fragmentação externa é um problema - como observado na Figura 3.13, aproximadamente 99% das configurações não utilizam mais do que dez níveis. Quando a arquitetura possui grande número de níveis, como na Configuração 4 (Tabela 3.2), a fragmentação externa é muito grande, muitas vezes ocupando somente um segmento do conjunto de segmentos destinados àquela entrada. Portanto, a fim de minimizar o problema de fragmentação é proposta uma nova abordagem para a memória de contexto.

A solução proposta divide também a memória de contexto em partes, entretanto, usando um nível da matriz reconfigurável como unidade básica de segmento de reconfiguração. Além disso, divide a memória de contexto em duas partes, uma que armazena o contexto de entrada (Figura 5.7(a)), e outra que armazena os bits de configuração de cada nível (Figura 5.7(b)).

Cada entrada da Tabela de Configuração mantém os bits responsáveis pela configuração de um único nível e o endereço que indica o próximo nível, se ele existir. Dessa forma, o tamanho da palavra da memória de contexto diminui se comparada à memória de contexto original. Por outro lado, como uma configuração normalmente é composta por vários níveis, o número de entradas na tabela aumenta, uma vez que mais entradas são necessárias para representar um contexto.

O Contexto de Entrada contém dados referentes ao endereço de retorno após a execução, ao número de níveis utilizados, aos operandos imediatos e ao mapa de registradores, que indica quais registradores serão mapeados em cada linha de contexto.

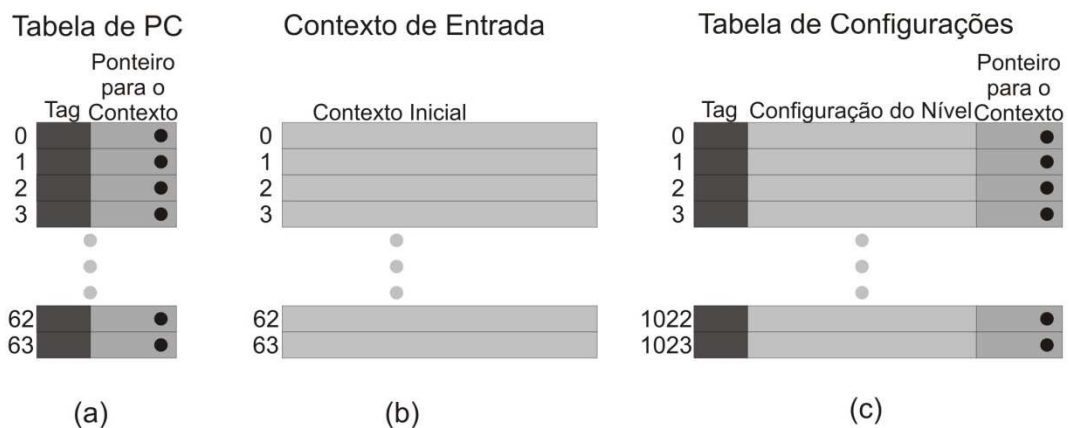


Figura 5.7: Memória de contexto proposta.

O processo de busca funciona da seguinte forma: quando uma configuração é encontrada na Tabela de PC, dois ponteiros são buscados. O primeiro indica o endereço do contexto inicial para essa configuração na Tabela de Contexto de Entrada; já o

segundo ponteiro indica os bits de configuração do primeiro nível na Tabela de Configuração, então as duas memórias são acessadas simultaneamente. O contexto de entrada é obtido apenas uma vez independentemente do número de níveis, entretanto, o número de vezes que a Tabela de Configuração é acessada depende de quantos níveis a configuração atual é composta. Cada entrada adicional (nível) obtida a partir da Tabela de Configuração leva um ciclo de relógio adicional.

Com essa técnica, chamada de busca sob demanda, apenas os níveis que são realmente utilizados são carregados. Embora mais acessos à memória sejam necessários, a energia consumida no acesso de uma configuração é significativamente menor por duas razões: a largura da porta de memória é reduzida ao mesmo tamanho que o número de bits necessários para configurar apenas um nível; e somente os níveis que serão realmente utilizados são buscados. A operação é muito semelhante aos sistemas de arquivos que usam alocação não contígua e utiliza uma lista encadeada como estrutura de dados primária, como o sistema de arquivos FAT (*File Allocation Table*) (MICROSOFT, 2000).

Essa abordagem, além de não apresentar fragmentação externa, tem a vantagem de melhor utilizar a memória de contexto. Considerando que a maior parte das configurações que são mantidas na memória de contexto não use o número total de níveis disponíveis na matriz, as entradas da memória não utilizadas agora podem armazenar outras configurações. Cada segmento da memória não pertence a uma determinada configuração, o segmento é compartilhado, podendo ser alocado por qualquer configuração, se disponível.

Por exemplo, em um sistema que tem uma memória de contexto de cinquenta entradas e uma matriz reconfigurável de trinta níveis, no sistema original seria necessário uma memória capaz de armazenar mil e quinhentos níveis de configuração. No entanto, assumindo que, em média, apenas quinze níveis da matriz são realmente utilizados em cada configuração, uma memória com metade do tamanho do original seria suficiente considerando a abordagem proposta.

Outra vantagem da busca sob demanda é observada quando a matriz reconfigurável é substituída por outra com maior número de níveis. Utilizando essa técnica a modificação da memória de contexto não é necessária, visto que o número de níveis de uma configuração está limitado somente pelo número de blocos disponíveis.

5.2.1 Operações básicas

Existe um mecanismo especial para controlar tanto as entradas da Tabela de Configuração que estão livres como as que estão alocadas (as entradas livres são mantidas em uma lista ligada). Um registrador especial armazena o endereço da primeira entrada livre (Figura 5.8(c)). Como já exposto, cada entrada é dividida em duas partes: a configuração para um nível da matriz e um ponteiro que indica qual o endereço da configuração do próximo nível a ser buscado, caso seja necessário. Quando a entrada estiver vazia, o mesmo campo que indicaria a configuração do próximo nível é usado para indicar a próxima entrada livre.

A Figura 5.8 apresenta a alocação de uma configuração composta de 6 níveis, representada na Tabela de Configuração pelas setas contínuas. As setas tracejadas representam a lista de blocos livres.

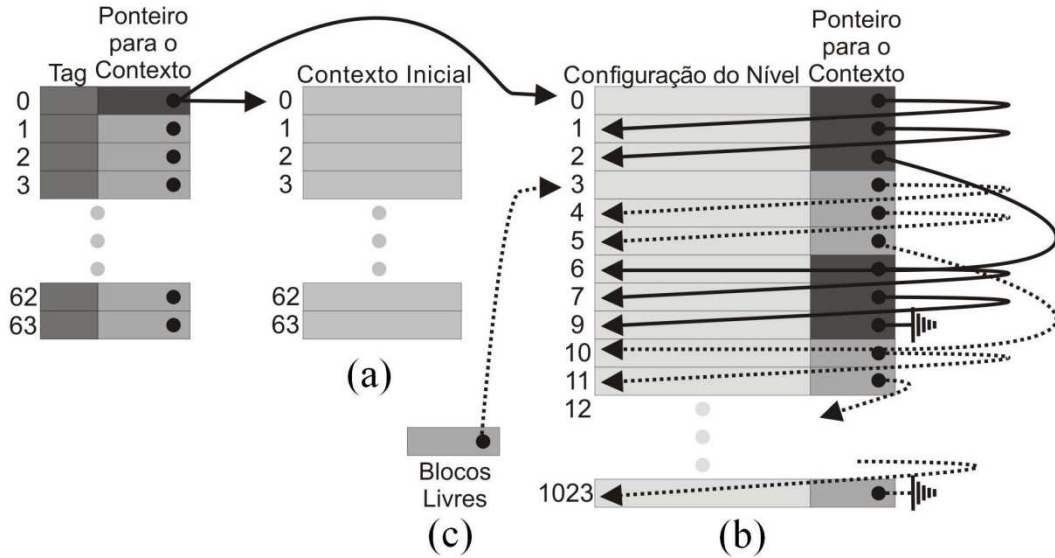


Figura 5.8: Mecanismo da memória de contexto proposta.

Quando uma nova configuração é inserida, o registrador que indica a primeira entrada livre é consultado para indicar qual entrada pode ser alocada e posteriormente atualizado. Para a remoção de uma configuração, liga-se a lista de configuração no fim da lista de entradas livres - basta utilizar o registrador que armazena endereço da última entrada livre. A parte do endereço dessa entrada é atualizada com o endereço da primeira entrada da configuração que se deseja excluir e, posteriormente, o registrador é atualizado. Para auxiliar o controle, existe outro registrador que contabiliza o número de entradas livres, que é atualizado nas operações de inserção e deleção.

5.2.2 Consumo de energia

Como já discutido, ao adotar o método original a busca de uma configuração na memória de contexto consome grande quantidade energia, porque todos os bits da configuração são sempre carregados. Utilizando o mecanismo de busca sob demanda, somente os níveis realmente utilizados serão carregados, assim economizando grande parcela da energia.

A Tabela 5.2 mostra o consumo total de energia (em mJoules) para cada configuração determinada na Tabela 3.2, considerando a execução de todas as aplicações e comparando com o mecanismo original. A redução de consumo de energia na memória de contexto é de 95%, 98%, 99% e 99% para as Configurações 1, 2, 3 e 4, respectivamente.

Tabela 5.2: Energia consumida (mJ) na execução de todas as aplicações do *benchmark*.

Conf. 1		Conf. 2		Conf. 3		Conf. 4	
Orig.	Sob Demanda	Orig.	Sob Demanda	Orig.	Sob Demanda	Orig.	Sob Demanda
147,07	7,08	828,69	9,06	4910	11,23	19632	11,25

O resultado acima, como na abordagem da busca parcial, reflete o fato de que 99% das configurações executadas são compostas de menos de 10 níveis. Assim, quanto maior o número de níveis da matriz reconfigurável, mais energia era desperdiçada no carregamento de bits de configuração não utilizados.

Analisando o consumo global de energia do sistema reconfigurável, pode-se avaliar o impacto da técnica apresentada. A Figura 5.9 mostra que, usando o método de acesso original, a memória de contexto foi responsável por uma parte relevante da energia total do sistema: para as Configurações 1, 2, 3 e 4, a memória de contexto consome 22%, 63%, 91 % e 98% da energia de todo sistema reconfigurável, respectivamente. Nas três últimas configurações, a memória de contexto consome mais do que a matriz reconfigurável, as memórias RAM e ROM, e o processador juntos.

Por meio do mecanismo de busca sob demanda, a memória contexto corresponde a apenas 1,4, 1,9, 2,4 e 2,7% da energia total do sistema para as quatro matrizes reconfiguráveis avaliadas. Com essa abordagem, a energia consumida pelo sistema além de ser menor que o sistema original, é menor se comparada à energia consumida quando se executam as aplicações do *Mibench* somente no processador MIPS R3000 (primeira coluna da Figura 5.9). A economia de energia obtida se deve à diminuição dos acessos à memória de programa, pois grande quantidade de instruções é compactada em configurações que serão executadas pela matriz reconfigurável.

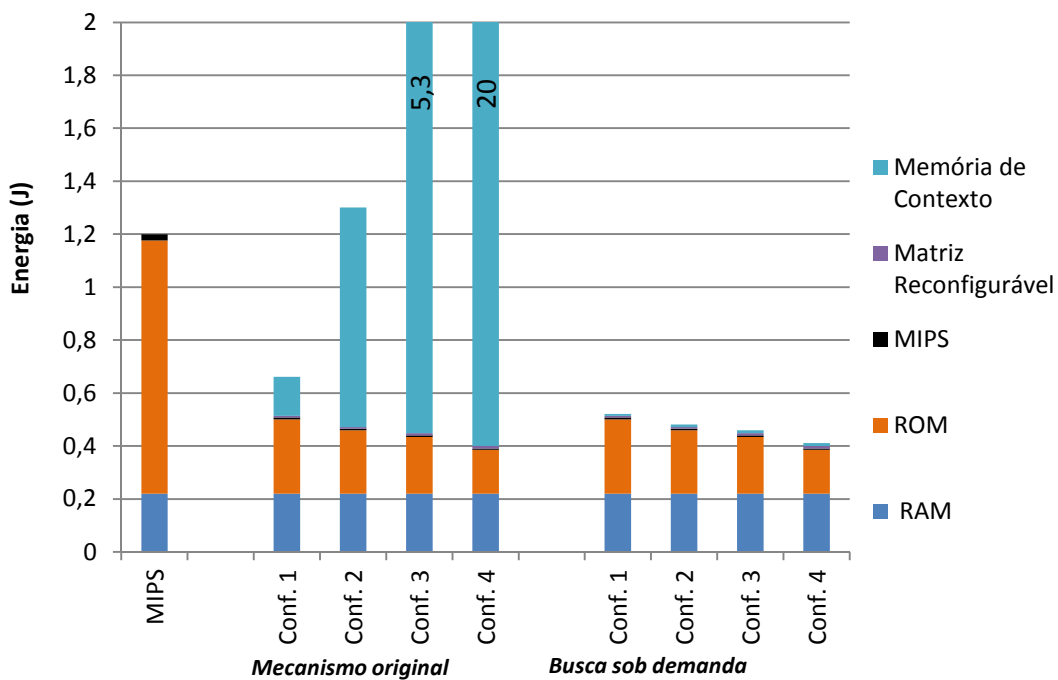


Figura 5.9: Consumo de energia pelo sistema reconfigurável.

5.2.3 Área

A área tem demonstrado redução significativa devido principalmente à diminuição do tamanho da porta da memória de contexto (fator de 8, 16, 32 e 64 vezes para configurações 1, 2, 3 e 4, respectivamente) e a economia do número de bits total de memória. A nova técnica permite uma redução de 15%, 55%, 80% e 90% do número total de bits necessários para manter o mesmo número de configurações para as quatro diferentes configurações da matriz, como o mecanismo original. Os resultados apresentados consideram que o mecanismo original e o proposto são capazes de manter 64 configurações. A Tabela 5.3 mostra os valores de área (em mm^2).

Tabela 5.3: Comparativo da área (em mm²) da memória de contexto do mecanismo original e o proposto.

<i>Conf. 1</i>		<i>Conf. 2</i>		<i>Conf. 3</i>		<i>Conf. 4</i>	
Orig.	Sob Demanda	Orig.	Sob Demanda	Orig.	Sob Demanda	Orig.	Sob Demanda
65,8	1,9	384,9	2,66	2231,4	3,47	8906,1	3,47

A área resultante é semelhante à obtida na técnica de segmentação da configuração quando o segmento é equivalente a um nível, como pode ser visualizado na primeira coluna da Tabela 5.1. Porém, como tal técnica permite a redução total do número de bits da memória, a área é menor que a técnica anteriormente apresentada, principalmente na matriz reconfigurável que possui maior quantidade de níveis.

Um dado interessante que pode ser observado na Tabela 5.3 é a igualdade da área resultante das Configurações 3 e 4. Como apresentada na Tabela 3.2, a única diferença entre as Configurações 3 e 4 é o número de níveis; portanto, cada entrada da memória de contexto possui o mesmo tamanho. Como o número de configurações que utilizam mais de dez níveis é muito baixo, as configurações podem ser mantidas mesmo com o número de entradas reduzidas, logo, as memórias são idênticas para esse caso.

Os dados são baseados na execução do *Mibench*, entretanto, caso sejam geradas configurações com muitos níveis em maior frequência necessitando mantê-las simultaneamente na memória de contexto, algumas configurações precisarão ser substituídas e, por conseguinte, podem ocorrer *Cache Misses* e degradação do desempenho. O caso relatado não foi simulado nos experimentos realizados, mas será avaliado em trabalhos futuros.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foram explorados dois aspectos importantes na implementação de uma arquitetura reconfigurável: o grande impacto de área que as unidades reconfiguráveis produzem na área total do sistema; e a importância que a memória de contexto tem no consumo de energia em sistemas reconfiguráveis, sendo a mesma responsável por grande parte do consumo de energia do sistema.

Como estudo de caso, a arquitetura reconfigurável proposta por (BECK, 2006a), acoplada a um processador MIPS R3000, foi utilizada a fim de explorar as técnicas desenvolvidas neste trabalho para redução da área e do consumo de energia em sistemas reconfiguráveis.

No Capítulo 4 foram apresentadas duas técnicas de virtualização de hardware. A técnica denominada *pipeline* de estágio reconfigurável divide o circuito original em blocos lógicos, em que ao se utilizar apenas alguns deles, é possível virtualizar os demais blocos. Para tanto foram inseridos registradores entre os blocos, que são reconfigurados ao longo da execução, semelhante a um *pipeline*. A outra abordagem é chamada de *pipeline* virtual. Essa técnica é semelhante à primeira abordagem, porém permite empregar a virtualização de hardware sem perder a propriedade combinacional do circuito – o que mantém o desempenho muito próximo do circuito original. Ambas as técnicas de virtualização atingiram uma redução significativa da área. Ao se compará-las à arquitetura do estudo de caso original foi alcançada uma redução de área superior a 94%, e menos de 1% de perda de desempenho utilizando-se a primeira abordagem. As técnicas demonstradas também podem ser adotadas em outras arquiteturas reconfiguráveis, não se limitando a arquitetura abordada.

No Capítulo 5 foram expostas duas técnicas para a redução do consumo de energia por meio da modificação da memória de contexto. A primeira abordagem baseia-se na exploração da largura ideal da porta da memória combinado com o número de acessos, para que se minimize a energia consumida na busca dos bytes de configuração. Essa técnica foi demonstrada tanto para reconfiguração parcial quanto para a reconfiguração total. Foram analisadas quatro configurações da matriz reconfigurável de diferentes tamanhos, as quais apresentaram redução de 67% a 26% da energia quando a reconfiguração é total e redução de 96% a 25% da energia quando a arquitetura permite reconfiguração parcial.

A outra abordagem apresentada utiliza um mecanismo chamado busca sob demanda. Tal abordagem, além de permitir que as configurações sejam acessadas parcialmente,

possui um mecanismo de gerenciamento por meio de listas ligadas, em que cada segmento da memória não pertence a uma determinada configuração, assim, possibilitando um uso mais eficiente da memória contexto. A abordagem apresentou redução de 99% a 95% da energia consumida nos acessos às configurações. A capacidade da memória em número de bytes também foi reduzida em 90% na maior matriz utilizada e em 15% na menor, sem nenhuma penalidade no desempenho.

Na arquitetura do estudo de caso, a memória de contexto é responsável por 98%, 91%, 63% e 22%, na menor matriz, de toda a energia consumida no sistema reconfigurável. Por meio da técnica de busca sob demanda foi possível reduzir essas proporções para 2,7%, 2,4%, 1,9% e 1,4% da energia do sistema. Para todas as configurações, utilizando-se a matriz reconfigurável, a energia consumida na execução de todas as aplicações do *benchmark* foi menor ao se comparar com a energia consumida executando-se as aplicações somente pelo processador, ou seja, foi possível obter ganho desempenho com menor consumo de energia, e com a área da matriz reconfigurável menor que sete vezes a do processador, utilizando-se a técnica de virtualização.

As técnicas de exploração da memória de contexto apresentadas podem ser adaptadas a outras arquiteturas reconfiguráveis. Tanto arquiteturas de grão fino quanto de grão grosso necessitam de enorme quantidade de bytes para cada configuração, logo, é possível se obter grande redução de energia ao se aperfeiçoar os acessos à memória para se obter menor consumo.

6.1 Trabalhos Futuros

6.1.1 Virtualização do tradutor binário

A técnica de virtualização permite que sejam executados infinitos níveis, limitando-se apenas ao tamanho das configurações que a memória de contexto pode armazenar. Com o método de busca sob demanda, o tamanho de uma configuração é limitada pelo número de entradas da memória, podendo ser expandida facilmente em projetos que tenham essa necessidade. A união dessas duas técnicas traz para o sistema uma grande flexibilidade, porém, em sistemas que possuem um tradutor binário, responsável pela geração das configurações, esse componente pode vir a ser um limitante para geração de configurações de grande tamanho, como é o caso da arquitetura do estudo de caso.

O TB, responsável pelo mapeamento e verificação das dependências das instruções, utiliza tabelas que armazenam esses dados de controle. Atualmente, essas tabelas têm o tamanho proporcional ao número de níveis definido pela matriz sem virtualização, ou seja, possuem tamanho fixo. Portanto, se aplicações anteriormente não previstas em tempo de projeto migrarem para o sistema reconfigurável, não será extraído todo o potencial da matriz reconfigurável pelo fato das tabelas imporem um limite do tamanho de uma configuração.

Para deixar todo o sistema mais flexível, um estudo sobre virtualização do TB mostra-se interessante para tornar a arquitetura totalmente flexível.

6.1.2 Interconexões das unidades funcionais com Rede Omega

Juntamente com a virtualização hardware, pode-se utilizar a técnica apresentada em Ferreira (2008), pois tal estudo realiza a redução da área por meio da otimização das conexões entre as unidades funcionais do nível, diferentemente da técnica de

virtualização, que reduz o número de níveis necessários. Logo, a união das técnicas irá possibilitar maior redução da área ocupada pela matriz reconfigurável.

6.1.3 Adição de unidades reconfiguráveis mais complexas

Como foi possível reduzir o número de níveis por meio da virtualização de hardware, torna-se viável a adição de unidades reconfiguráveis mais complexas. A unidade de ponto flutuante é um exemplo atraente, já que atualmente essas instruções não são suportadas na matriz reconfigurável, ocasionando uma quebra da configuração. Portanto, se adicionadas, aplicações com ponto flutuante poderão se beneficiar da aceleração proporcionada pelo sistema reconfigurável.

6.1.4 Política de troca das configurações na memória de contexto

O mecanismo de busca sob demanda, apresentado na Seção 5.2, permite explorar mais variáveis, como o tamanho da configuração, para se adotar uma política de substituição de configurações da memória de contexto. Tal ação permite realizar uma exploração do espaço de projeto, contrabalançando a redução da capacidade de armazenamento e o número de *caches misses* da memória de contexto. Com isso, é possível obter uma relação entre desempenho e área, possibilitando que a técnica seja adaptada em projetos com diferentes restrições.

6.1.5 Explorar as técnicas de virtualização em outras arquiteturas reconfiguráveis

Já que em muitas arquiteturas a área ocupada pela matriz reconfigurável é grande, seria essencial realizar um estudo para reduzir a área dessas arquiteturas por meio das técnicas de virtualização de hardware apresentadas nesse trabalho.

Pode-se, igualmente, comparar algumas arquiteturas que já se aproveitam da virtualização, como exemplo a PipeRench, a qual utiliza uma técnica semelhante a de pipeline de estágio reconfigurável, e transformá-la de tal forma que se possa aplicar a técnica de pipeline virtual. Desse modo, seria possível verificar os possíveis ganhos em desempenho ao se manter as características de um circuito combinacional e os impactos na área e potência.

6.1.6 Impacto da memória de contexto em outras arquiteturas reconfiguráveis

Neste trabalho foi apresentada a grande significância que a memória de contexto possui no sistema reconfigurável. Para reforçar a importância pode-se avaliar outras arquiteturas reconfiguráveis, extraindo-se o consumo de energia da memória em relação ao sistema. Posteriormente, pode-se aplicar as técnicas abordadas a fim de avaliar a redução do consumo de energia.

REFERÊNCIAS

- BECK FILHO, A.C.S.; CARRO, L. **Automatic Dataflow Execution with Reconfiguration and Dynamic Instruction Merging**. In: VERY LARGE SCALE INTEGRATION, VLSI-SOC, 2006, Perth. Proceedings... New York: IEEE Computer Society, 2007. p. 30–35.
- BECK FILHO, A.C.S.; CARRO, L. **Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility**. In Proceedings of Design Automation Conference, DAC 42. Anaheim. 2005. p. 732 – 737.
- BECK FILHO, A.C.S.; RUTZIG M. B.; GAYDADJIEV, G.; L. CARRO. **Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications**. In: Design, Automation and Test in Europe, 2008.
- BENINI, L.; MACII, A.; MACII, E.; PONCINO, M. **Selective instruction compression for memory energy reduction in embedded systems**. In Proceedings of the 1999 International Symposium on Low Power Electronics and Design (San Diego, California, United States, August 16 - 17, 1999). ISLPED '99. ACM, New York, NY, 206-211.
- CARRILLO, J.E.; CHOW, P. **The effect of reconfigurable units in superscalar processors**. In: FPGA '01: Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays, pp. 141–150. ACM, New York (2001). doi:10.1145/360276.360328.
- CASPI E.; CHU M.; HUANG R.; YE H J.; WAWRZYNEK J.; DEHON A. **Stream computations organized for reconfigurable execution (SCORE)**. In Proc. 10th Int. Conf. on Field Programmable Logic and Applications (FPL), pages 605–614, 2000.
- CLARK, N., et al.: **An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors**. In: ISCA, pp. 272–283, 2005.
- CRONQUIST, D.C.; FISHER, C.; FIGUEROA, M.; FRANKLIN, P.; EBELING, C. **Architecture design of reconfigurable pipelined datapaths**. In: ARVLSI '99: Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI, p. 23. IEEE Computer Society, Los Alamitos (1999).
- DEHON, A. **DPGA utilization and application**. In Proc. 4th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA), pages 115–121, 1996.
- ENZLER, R.; PLESSL, C.; PLATZNER, M. **Virtualizing hardware with multi-context reconfigurable arrays**. In Proc. 13th Int. Conf. on Field Programmable Logic and Applications (FPL), pages 151–160, 2003.

- FERREIRA, R.S.; LAURE, M.; RUTZIG, M. B.; BECK FILHO, A.C.S.; CARRO, L. **Reducing Interconnection Cost in Coarse-Grained Dynamic Computing through Multistage Network**. In: International Conference on Field Programmable Logic and Applications, 2008, Heidelberg. Proceedings of., 2008.
- FLYNN, M. J.; HUNG, P. **Microprocessor Design Issues: Thoughts on the Road Ahead**. IEEE Micro, Los Alamitos, v.25, n.3, p. 16-31, May 2005.
- FUJI, T.; FURUTA, K.; MOTOMURA, M.; NOMURA, M.; MIZUNO, M.; ANJO, K.; WAKABAYASHI, K.; HIROTA, Y.; NAKAZAWA, Y.; ITOH, H.; YAMASHINA, M. **A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture**. In 46th IEEE Int. Solid-State Circuits Conf. (ISSCC), Dig. Tech. Papers, pages 364–365, 1999.
- GOLDSTEIN, S. C.; SCHMIT, H.; BUDIU, M.; CADAMBI, S.; MOE, M.; TAYLOR, R. R. **PipeRench: A reconfigurable architecture and compiler**. IEEE Computer, 33(4):70–77, Apr. 2000.
- GUPTA, R. K.; MICHELI, G. D. **Hardware-software co-synthesis for digital systems**. In: IEEE Design and Test of Computers. vol. 10, 1993, pp. 29-41.
- HA, Y.; VERNALDE, S.; SCHAUMONT, P.; ENGELS, M.; LAUWEREINS, R.; MAN, H. D. **Building a Virtual Framework for Networked Reconfigurable Hardware and Software Objects**. Journal of Supercomputing, 21(2):131–144, February 2002.
- HA, Y.; SCHAUMONT, P.; ENGELS, M.; VERNALDE, S.; POTARGENT, F.; RIJNDERS, L.; MAN H. D. **A hardware virtual machine for the networked reconfiguration**. In IEEE International Workshop on Rapid System Prototyping, pages 194–199, 2000.
- HAUCK, S.; FRY, T.W.; HOSLER, M. M.; KAO, J.P. **The chimaera reconfigurable functional unit**. In: FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines, p. 87. IEEE Computer Society, Los Alamitos (1997).
- HAUSER, J. R.; WAWRZYNEK, J. **Garp: A MIPS Processor with a Reconfigurable Coprocessor**. In: FPGA-BASED CUSTOM COMPUTING MACHINES, 1997, Napa Valley. Proceedings... Washington: IEEE Computer Society, 1997. 12-21.
- HUTCHINGS, B. L. **Exploiting reconfigurability through domain-specific systems**. In: INTERNATIONAL WORKSHOP ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 1997, London. Proceedings... Berlin: Springer, 1997. p.193 – 202. (Lecture Notes in Computer Science, v.1304).
- IENNE, P.; LEUPERS, R. **Customizable Embedded Processors: Design Technologies and Applications**. San Mateo : Morgan Kaufmann, 2006.
- JAIN, M. K.; BALAKRISHNAN, M.; KUMAR, A. **ASIP design methodologies: Survey and issues**. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, VLSID, 2001, Bangalore. Proceedings... Washington: IEEE Computer Society, 2001. p. 76 – 81.
- KIM, Y.; MAHAPATRA, R. N. **Dynamically Compressible Context Architecture for Low Power Coarse-Grained Reconfigurable Array**. In Proc. IEEE International Conference on Computer Design (ICCD), pp. 395-400, October 2007.

KIM, Y.; PARK, I.; CHOI, K.; PAEK, Y. **Power-Conscious Configuration Cache Structure and Code Mapping for Coarse-Grained Reconfigurable Architecture**. In Proc. of Int. Symp. on Low Power Electronics and Design, Oct. 2006.

KUCUKCAKAR, K. **An ASIP design methodology for Embedded Systems**, International Workshop on Hardware/Software Codesign, pages 17-21, 1999.

LAWREI, D. H. **Access and alignment of data in an array processor**. In IEEE Trans. Comput., vol. C-24, pp. 1145-1155, Dec. 1975.

LYSECKY, R.; VAHID, F. **A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning**. Design Automation and Test in Europe Conference, 2004.

MAHESWARAN, K.; AKELLA, V. **Hazard-free implementation of the self-timed cell set in a Xilinx fpga**. Tech. Rep., University of California (1994).

MARSHALL, A.; STANSFIELD, T.; KOSTARNOV, I.; VUILLEMIN, J.; HUTCHINGS, B. **A reconfigurable arithmetic array for multimedia applications**. In: FPGA '99: Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, pp.135–143. ACM, New York (1999). doi:10.1145/296399.296444.

MICROSOFT. Microsoft Extensible Firmware Initiative FAT32 File System Specification, 6 Dec. 2000; www.microsoft.com/hwdev/download/hardware/fatgen103.pdf.

MIYAMORI, T.; OLUKOTUN, K. **Remarc: Reconfigurable multimedia array coprocessor**. In: IEICE Transactions on Information and Systems E82-D, pp. 389–397 (1998).

PLESS, C.; PLATZNER, M. **Virtualization of hardware – introduction and survey**. In Proc. 4rd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA), pages 63– 69. CSREA Press, 2004.

RUTZIG, M. B.; BECK FILHO, A. C. S.; CARRO, L.. **Balancing Rconfigurable Data Path Resources According to Applications Requirements**. In: 15th Reconfigurable Architecture Workshop, 2008, Miami. Proceedings of 15th Reconfigurable Architecture Workshop, 2008.

RUTZIG, M. B. **Gerenciamento Automático de Recursos Reconfiguráveis Visando a Redução de Área e do Consumo de Potência em Dispositivos Embarcados**. 2008 Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SCHMIT, H.; WHELIHAN, D.; MOE, M.; LEVINE, B.; TAYLOR, R. **PipeRech: A virtualized programmable datapath in 0.18 micron technology**. In Proc. 24th IEEE Custom IntegratedCircuits Conf. (CICC), pp. 63–66.

SIMA, D.; FALK, H. **Decisive aspects in the evolution of microprocessors**. In Proceedings of the IEEE, 2004, pp. 1896-1926.

STITT, G.; VAHID, F. **Hardware/Software Partitioning of Software Binaries**. In IEEE/ACM International Conference on Computer Aided Design, 2002, pp. 164-170.

STITT, G.; VAHID, F. **The Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic**. In IEEE Design and Test of Computers, 2002.

- TANG, X.; AALSMA, M.; JOU, R. **A compiler directed approach to hiding configuration latency in Chameleon processors.** In Proc. 10th Int. Conf. on Field Programmable Logic and Applications (FPL), pages 29–38, 2000.
- TRIMBERGER, S.; CARBERRY, D.; JOHNSON, A.; WONG, J. **A time-multiplexed FPGA.** In Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), pages 22–28, 1997.
- TSCHANZ, J. et al. **Dynamic-sleep transistor and body bias for active leakage power control of microprocessors.** IEEE Journal of Solid-State Circuits, San Francisco, v. 38, n. 11, p. 1838 – 1845, November 2003.
- VENKATARAMANI, G.; NAJJAR, W.; KURDAHI, F.; BAGHERZADEH, N.; BOHM, W. **A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture.** In Conf. on Compiler, Architecture and Synthesis for Embedded Systems, 2001.
- VERMA, M.; WEHMEYER, L.; MARWEDEL, P. **Dynamic overlay of scratch-pad memory for energy minimization.** In International Conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS) (Stockholm, Sweden). ACM Press, New York, 2004.
- WALL, D .W., **Limits of instruction-level parallelism,** In Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, p.176-188, April 08-11, 1991, Santa Clara, California, United States
- WESTE, N.; HARRIS D. **CMOS VLSI Design: A Circuits and Systems Perspective,** 4/E/; ISBN-10: 0136076939 ISBN-13: 9780136076933; Publisher: Addison-Wesley, 2011.
- WIRTHLIN, M. J.; HUTCHINGS, B. L.; GILSON, K. L.: **The nano processor: A low resource reconfigurable processor.** In: Buell, D.A., Pocek, K.L. (eds.) IEEE Workshop on FPGAs for Custom Computing Machines, pp. 23–30. IEEE Computer Society, Los Alamitos (1994). citeseer.ist.psu.edu/wirthlin94nano.html
- ZHANG, C.; VAHID, F.; NAJJAR, W. **Energy Benefits of a Configurable Line Size Cache for Embedded Systems.** In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (Isvlsi'03) (February 20 - 21, 2003). ISVLSI. IEEE Computer Society, Washington, DC, 87.
- ZHANG, M.; CHANG, X.; ZHANG, G. **Reducing cache energy consumption by tag encoding in embedded processors.** In Proceedings of the 2007 international Symposium on Low Power Electronics and Design (Portland, OR, USA, August 27 - 29, 2007). ISLPED '07. ACM, New York, NY, 367-370.