VINÍCIUS DUARTE LOPES

UMA HEURÍSTICA POLINOMIAL PARA ESCALONAMENTO DE LOOPS EM ARQUITETURAS RECONFIGURÁVEIS DE GRÃO GROSSO

Dissertação apresentada a Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2013

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
L864h
2013

Lopes, Vinícius Duarte, 1989-
Uma heurística polinomial para escalonamento de loops em arquiteturas reconfiguráveis de grão grosso / Vinícius Duarte Lopes. – Viçosa, MG, 2013.
ix, 78 f. : il. ; 29 cm.

Orientador: Ricardo dos Santos Ferreira
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 73-78.

1. Circuitos gate arrays. 2. Dispositivo de lógica programável. 3. Heurística. 4. Arquitetura de computador. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Informação. II. Título.

CDD 22. ed. 004.22

VINÍCIUS DUARTE LOPES

UMA HEURÍSTICA POLINOMIAL PARA ESCALONAMENTO DE LOOPS EM ARQUITETURAS RECONFIGURÁVEIS DE GRÃO GROSSO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae.*

APROVADA: 24 de junho de 2013.

Carlos de Castro Goulart

José Augusto Miranda Nacif

Ricardo dos Santos Ferreira
(Orientador)

AGRADECIMENTOS

Agradeço a todos os meus familiares, minha namorada e meus amigos pelo apoio constante na realização deste trabalho. Em especial agradeço ao professor Ricardo Santos Ferreira pela orientação e sempre estar disposto a ajudar, aos amigos Lucas Mucida, Waldir Denver e Julio Vendramini por participarem diretamente. Também gostaria de agradecer à Universidade Federal de Viçosa e a todos os professores e funcionários do Departamento de Informática, pela oportunidade. Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa, sem ela provavelmente eu não teria chegado aqui. O agradecimento mais importante de todos vai para Deus, se é que ele realmente existe, como eu fortemente suspeito.

Sumário

Lista de Figuras **v**

Lista de Tabelas **viii**

Resumo **ix**

Abstract **x**

1 Introdução **1**

1.1 Motivação 2

1.2 Objetivos 4

1.3 Organização do texto 5

2 Referencial Teórico **7**

2.1 Arquiteturas Reconfiguráveis de Grão-Grosso (CGRAs) 7

2.2 Configuração de Arquiteturas Reconfiguráveis 10

2.3 *Modulo Scheduling* 11

2.3.1 *Modulo Scheduling* para CGRAs 12

2.3.2 Trabalhos relacionados 16

2.4 NP-Completude 19

2.5 Estrutura Geral 19

3 Heurística Polinomial para Escalonamento de *Loops* **22**

3.1 Arquitetura Base 22

3.2 Algoritmo MSG Base 23

3.2.1 Exemplo: Heurística MSG Base 26

3.3 Compartilhamento de Registros 29

3.4 Registros Locais 31

3.4.1 Exemplo: Comparação entre estratégias e arquiteturas 32

3.5 Tolerância a falhas no circuito 34
3.5.1 Exemplo: Heurística MSG tolerante a falhas 34
3.6 Mapeamento de *multikernels* 37

4 Modelo de Implementação em *Hardware* 39
4.1 Estrutura da Memória e MEF 39
4.2 Execução da MEF 42

5 Resultados 47
5.1 *Benchmarks* Utilizados 47
5.2 Economia de Registros 49
5.3 Intervalo de Iniciação 51
5.4 Tamanho da Arquitetura 54
5.5 Tempo de Compilação 57
5.6 Falhas no CGRA 60
5.7 *Multikernels* 62
5.8 Algoritmo em *Hardware* 64

6 Conclusões 66
6.1 Trabalhos Futuros 68
6.1.1 Rebalanceamento 68
6.1.2 Banco de registradores 70

Referências Bibliográficas 73

Lista de Figuras

2.1 Exemplos de redes de interconexão em CGRAs: (a)Malha Simples (b)Malha *Plus* (c)Agrupamentos (d)*Crossbar*. 8
2.2 Exemplos de redes de interconexão em CGRAs: (a) Linhas (b) Multiestágios. 9
2.3 Exemplo de algoritmo *modulo scheduling*: (a) Grafo de dependências (b) Arquitetura CGRA (c) Resultado do mapeamento (d) Execução detalhada. Adaptado de [Mei et al., 2005]. 13
2.4 Exemplo de algoritmo *modulo scheduling*: (a)Partição 1 (b)Partição 2 (c)Paralelismo espacial com sobreposição temporal e particionamento. . . 14
2.5 Exemplo de falha no roteamento (a)Grafo dataflow dividido em partições temporais (b)Arquitetura em malha 2x2 (c)Posicionamento Inicial (d)Falha no roteamento de f para h (e)Falha solucionada por reposicionamento de operações. 16
2.6 Ambiente de aceleração de laços com computação intensiva, em duas abordagens diferentes: à esquerda com o algoritmo de mapeamento MSG implementado para executar sequencialmente no processador *softcore* (MSG) e à direita com o algoritmo EPR implementado em *hardware* (EPR). 21

3.1 Arquitetura base, por [Ferreira et al., 2011a] 23
3.2 Grafo de fluxo de dados não balanceado (esquerda) e balanceado (direita). 26
3.3 Arquitetura com 4 PEs (à direita), sendo os dois primeiros com capacidade de entrada externa de dados, e grafo de fluxo de dados a ser mapeado (à esquerda). 26
3.4 Posicionamento das três primeiras operações (a) com entrada externa de dados (D e E) e (b) adicionando a operação F e fazendo o roteamento de $D \rightarrow F$. 27

3.5 Estado final do mapeamento para $II = 1$. Falta posicionar G, porém não há mais PEs disponíveis. É necessário recomeçar com $II = 2$. 28
3.6 Mapeamento das três primeiras operações. 28
3.7 Escalonamento final, $II = 2$. 29
3.8 (a) Balanceamento normal (b) Compartilhamento de Registros. 30
3.9 Exemplo real: parte extraída do escalonamento do *benchmark ewf* (a) sem compartilhamento de registros (b) com compartilhamento de registros. 31
3.10 (a) Arquitetura base: ignora B, somente A é utilizado (b) Arquitetura para registros locais: pode usar os dois registros. 32
3.11 (a) Grafo não balanceado (b) Grafo balanceado (c) Mapeamento na arquitetura base (d) Mapeamento na arquitetura para RL: usa os dois registros. 33
3.12 Grafo de fluxo da dados a ser mapeado no CGRA com 4 PEs, sendo que o PE 1 apresenta falha. 35
3.13 Estado do mapeamento em que estão posicionadas as operações com entrada externa de dados. 36
3.14 Estado final do mapeamento. 36
3.15 Exemplo de compressão de mapeamento (a) Grafo a ser mapeado (b) mapeamento normal (b) mapeamento comprimido. 37
3.16 Dois grafos a serem mapeados ao mesmo tempo (*multikernel*) são tratados como apenas um. 38

4.1 Sistema Reconfigurável: (a) estrutura interna da heurística MSG em *hardware* (b) *benchmark* de entrada (c) modelo de implementação em *hardware* (d) o modelo de implementação em *software* usando *softcore* para executar o código C. 40
4.2 Memórias de Escalonamento, Posicionamento e Roteamento para a heurística MSG base. 41
4.3 Máquina de estados finitos. 41
4.4 Exemplo de grafo de fluxo de dados. 42
4.5 (a) Mapeamento parcial (b) Estados percorridos para os nós A, B, C e D (c) Código do estado POSICIONA_ENTRADA: todas as operações do código são executadas paralelamente. 43
4.6 (a) Mapeamento parcial. (b) Estados percorridos para os nós E e F. . . 44
4.7 (a) Mapeamento final. (b) Estados percorridos para o nó H. 45

4.8 Descrição dos estados INSERE_ROTEIA_REG e VERIFICA_N_REGS. Todas as operações de cada estado são executadas em paralelo. 46

5.1 (a) Grafo de Fluxo de dados (b) Impossibilidade de roteamento: impossível rotear $A \rightarrow D$. 49
5.2 CGRA 4x4, PEs são interconectados em malha 2-D. Cada PE é uma ALU somado a um banco de registros locais. Figura extraída de [Hamzeh et al., 2013]. 53
5.3 Intervalo de Iniciação para três arquiteturas: 16, 20 e 24 PEs. 55
5.4 Paralelismo em nível de instrução para três arquiteturas: 16, 20 e 24 PEs. 55
5.5 Ocupação para três arquiteturas: 16, 20 e 24 PEs. 56
5.6 Impacto da inserção de falhas nos PEs da arquitetura de 16 PEs. Algoritmo MSG e arquitetura básicos. 61
5.7 Impacto da inserção de falhas nos PEs da arquitetura de 16 PEs. Algoritmo MSG e arquitetura com registros locais. 61

6.1 Exemplo de aumento desnecessário do II: (a) Grafo de fluxo de dados (b) Tentativa de mapear com o $II = MinII$ (c) Mapeamento final com II=3. 68
6.2 Deslocamento de operações (a) Grafo de fluxo de dados com operações deslocadas (b) Mapeamento realizado com $II = MinII$. 69
6.4 Espalhamento do posicionamento do *benchmark cosine1* nas partições temporais. 70
6.3 Exemplo de escalonamento do *benchmark* cosine1. 71
6.5 Estrutura da arquitetura com banco de registros locais (BR). 72
6.6 Grafo de fluxo de dados de exemplo com mapeamento com II = 1. 72

Lista de Tabelas

1.1 Total de laços extraídos dos algoritmos H.264, 3D, AAC e MP3 e a relação de quantos têm até 96 operações. Resultados extraídos de [Park et al., 2008]. 2

1.2 Porcentagem de recursos de FPGA para arquiteturas *crossbar* com 16, 20 e 24 elementos de processamento e um CGRA baseado no ADRES, em um FPGA XILINX xc6vlx75t. Total de LUTs: 46558. Total de RAMs: 156. 4

5.1 Conjunto de *benchmarks* selecionados. 48

5.2 Número de PEs usados como registros para três estratégias: base, registros compartilhados e registros locais. 50

5.3 Intervalo de Iniciação para três estratégias: base (BA), registros compartilhados (RC), registros locais (RL). Em negrito os resultados ótimos. . . 51

5.4 Comparação entre a heurística MSG com registros locais e REGIMap. Os tempos estão em milissegundos (10^{-3}s). 53

5.5 Comparação das arquituras de 24 PEs com 4 e 6 entradas externas, para os 7 *benchmarks* limitados por entradas. 57

5.6 Tempo de compilação em μ segundos para três processadores diferentes: Intel Core i7, MicroBlaze e ρ-Vex. 58

5.7 Tempo de compilação e ciclos por nó. 59

5.8 *Multikernels*. 63

5.9 Clocks e tempos (em μ segundos) dos *benchmarks* para o algoritmo RLMEF. Tmp100 e Tmp50 representam os tempos para RLMEF, quando o ciclo do *hardware* é 100 e 50Mhz, respectivamente. MicroBlaze é o tempo de compilação no *softcore* Microblaze. 65

Resumo

LOPES, Vinícius Duarte, M.Sc., Universidade Federal de Viçosa, Junho de 2013. **Uma Heurística Polinomial para Escalonamento de Loops em Arquiteturas Reconfiguráveis de Grão Grosso** Orientador: Ricardo Santos Ferreira.

Atualmente as arquiteturas reconfiguráveis são atrativas em desempenho e baixo consumo de energia para aplicações com laços de computação intensiva. FPGAs são arquiteturas de grão fino que oferecem possibilidade de aceleração para essas aplicações, porém, o processo de mapeamento geralmente é demorado e complexo. Como alternativa, surgem as arquiteturas reconfiguráveis de grão grosso, que provêem menor flexibilidade que os FPGAs, porém menor complexidade de mapeamento. O objetivo deste trabalho é o mapeamento em tempo de execução do grafo de fluxo de dados, que representa um laço, em uma arquitetura reconfigurável grão grosso. O problema é NP-Completo e as diversas heurísticas encontradas na literatura não são viáveis para uma implementação dinâmica. Nesta dissertação propomos uma nova heurística capaz de realizar o mapeamento em tempo de execução da aplicação. Enquanto soluções anteriores necessitam de segundos para mapear aplicações, resultados experimentais mostraram que a solução proposta requer em média apenas 390 microssegundos para gerar mapeamentos próximos do ótimo na arquitetura utilizada, para 15 *benchmarks* extraídos de aplicações multimídia. Assim, a solução apresentada pode ser implementada em um ambiente de compilação *Just-In-Time*, podendo ser utilizada em contextos dinâmicos onde várias aplicações compartilham a arquitetura reconfigurável com possibilidade de mudança na composição dos elementos de processamento ou em cenários com presença de falhas no *hardware*. Além disso, apresentamos um modelo de implementação da heurística em *hardware*, com potencial redução do tempo de mapeamento em até 90% em relação à execução em *software* em um ambiente *Just-In-Time*.

Abstract

LOPES, Vinícius Duarte, M.Sc., Universidade Federal de Viçosa, June, 2013. **A Polinomial Heuristic Scheduling of Loops on Coarse Grained Reconfigurable Architectures.** Adviser: Ricardo Santos Ferreira.

Nowadays reconfigurable architectures are attractive both in performance and low power consumption for applications with computing intensive loops. FPGAs are fine-grained architectures that offer the possibility of acceleration for these applications, however, the mapping process is typically time consuming and complex. Although Coarse Grained Reconfigurable Architectures (CGRAs) provide less flexibility than FPGAs, they has less complex mapping. The goal of this work is the runtime mapping of the dataflow graph, which represents a loop, on a reconfigurable architecture. The problem is NP-Complete and the heuristics found in the literature are not feasible for a dynamic implementation. We propose a heuristic solution that utilizes two distinct mechanisms: an architecture that simplifies the most of the mapping process, with minimal degradation in performance, and an algorithm to perform the mapping at run-time application. While previous solutions require seconds to mapping applications, the proposed solution requires only microseconds to generate near-optimal mappings. Accordingly, the presented solution enables the use of the Just-In-Time compilation, which can be used in dynamic contexts where various applications share a reconfigurable architecture with a possibility of change in composition of processing elements or scenarios with presence of hardware failures. Furthermore, we present an implementation model of the algorithm in hardware, which aims to be 90% faster when compared to the software Just-In-Time solution.

1. Introdução

Atualmente muitas aplicações exigem alto desempenho para processamento de fluxo de dados (por exemplo, aplicações que envolvem geração de imagens em tempo real) e usar um processador de propósito geral pode não satisfazer as restrições de tempo, energia e memória. Uma solução é o desenvolvimento de um *hardware* específico para acelerar a execução, usando a computação temporal e espacial, na qual vários elementos de processamento simples operam concorrentemente.

Nessa direção, uma alternativa são os circuitos integrados de aplicação específica (ASICs), que são fabricados para executar uma tarefa específica, satisfazendo restrições específicas. No entanto, demandam um longo ciclo de tempo para desenvolvimento, gerando um alto custo. Além disso, não podem ser reprogramados, o que é incompatível com o mercado dinâmico onde as aplicações estão em constante adaptação.

Outra alternativa são as arquiteturas reconfiguráveis. Estas surgiram como uma solução intermediária entre a flexibilidade da programação em *software* dos microprocessadores e a eficiência dos ASICs, além de prover rápida prototipação. *Field-Programmable Gate Arrays* (FPGAs) oferecem essa flexibilidade por meio de um grande número de unidades reconfiguráveis de grão-fino e elementos de interconexão.

Entretanto, mapear aplicações genéricas para FPGAs é um desafio, sendo um problema NP-Completo [Yoon et al., 2009] que exige tempo de processamento da ordem de minutos, horas ou até dias, nos piores casos. Problemas adicionais são o tamanho da memória de configuração do dispositivo, causado pela granularidade em nível de bit e a sobrecarga da área de roteamento causada pela necessidade de um grande número de fios de interconexão, cerca de 90% do circuito [Hauck & DeHon, 2007].

Com o principal objetivo de reduzir complexidade e tempo de configuração, surgiram os *Coarse-Grained Reconfigurable Arrays* (CGRAs), que se reconfiguram em nível de palavra, não em nível de bit, como os FPGAs. Essa diferença na granularidade reduz o número de bits usados na configuração, reduzindo o tempo de configuração e a complexidade do posicionamento e roteamento. Muitos CGRAs foram propostos com objetivo de reduzir também área, atraso, tamanho da rede de interconexão e consumo de energia, em comparação com FPGAs [Bouwens et al.,

2007; Brant & Lemieux, 2012; Hoo & Kumar, 2012; Coole & Stitt, 2010; Friedman et al., 2009].

Apesar da redução da complexidade e sobrecarga de configuração provida pelas arquiteturas de grão-grosso, os algoritmos de compilação para CGRAs continuam tendo tempo de execução alto, impossibilitando a compilação em tempo de execução. Os melhores resultados da literatura, em tempo de compilação, gastam em média de 1 a 100 segundos [Yoon et al., 2008; Chen & Mitra, 2012; Hamzeh et al., 2012]. Além disso, para laços com número considerável de operações, com cerca de 60 operações, as soluções propostas na literatura geralmente gastam tempos superiores a 30 minutos. Sendo assim, o desafio é propor uma arquitetura e uma heurística que sejam simples e rápidas o bastante para a compilação em tempo de execução.

1.1 Motivação

O trabalho desenvolvido por [Park et al., 2009] mostra que, para três aplicações multimídia importantes (*ACC decoder*, *H.264 decoder* e *3D graphics rendering*), 71% do tempo de execução é gasto em laços que são paralelizáveis com *software pipelining* [Allan et al., 1995]. Além disso, cerca de 90% desses laços contém menos de 100 operações, como mostra a Tabela 1.1, extraída de [Park et al., 2008]. Se a execução fosse otimizada, por exemplo executando as operações paralelamente em um CGRA, o tempo total de execução poderia ser reduzido a um terço no melhor caso.

Algoritmo	Nº de Laços	Até 96 operações
H.264	81	64
3D	84	81
AAC	36	34
MP3	13	9
Total	**214**	**188**
Relação	-	**88,26%**

Tabela 1.1. Total de laços extraídos dos algoritmos H.264, 3D, AAC e MP3 e a relação de quantos têm até 96 operações. Resultados extraídos de [Park et al., 2008].

A técnica de *software pipelining* consiste em sobrepor a execução de instruções de múltiplas iterações de um laço, onde o objetivo é alcançar uma taxa mínima para o intervalo entre o início das diferentes iterações. Assim, uma iteração pode iniciar

antes mesmo do fim da iteração anterior. Uma abordagem para a realização do *software pipelining* é o *modulo scheduling* [Rau, 1994], que primeiramente encontra um intervalo de iniciação mínimo para o início de iterações subsequentes, considerando os recursos da arquitetura e a estrutura do laço e, em seguida, faz sucessivas tentativas de escalonar o laço na arquitetura. Se a tentativa falhar, o *modulo scheduling* reinicia com um intervalo de iniciação maior, até que encontre uma solução ou desista de tentar.

Muitos algoritmos que usam *modulo scheduling* foram propostos para escalonar esses laços em CGRAs [Hatanaka & Bagherzadeh, 2007; Mei et al., 2003; Park et al., 2006; Oh et al., 2009; Park et al., 2009], no entanto, os tempos de compilação são elevados para a compilação em tempo de execução (compiladores *Just-In-Time* - JIT), onde o tempo gasto na tarefa de compilar somado ao tempo de execução no *hardware* paralelo deve ser consideravelmente menor que o tempo sequencial. O objetivo geral deste trabalho é paralelizar trechos de laços para um CGRA em tempo de execução, de forma a minimizar o tempo total de execução.

Recentemente, uma heurística, chamada MSPR, juntamente com uma arquitetura CGRA, propostas por [Ferreira et al., 2011a], consegue alcançar um tempo de compilação compatível com a compilação JIT, da ordem de milissegundos. Porém, o MSPR tem alguns problemas. Primeiramente, a arquitetura usa rede de interconexão multiestágio. Essa rede possibilita o roteamento de um elemento de processamento para qualquer outro, porém pode haver conflito, podendo impossibilitar a realização do roteamento, causando aumento do tempo de compilação na busca de uma alternativa de roteamento. Além disso, devido aos conflitos da rede, o MSPR exige uma arquitetura com mais elementos de processamento que o mínimo necessário para o posicionamento. Isso torna a arquitetura mais cara e mais lenta (tempo de ciclo maior). Uma alternativa à rede de interconexão multiestágio é a rede *crossbar*, pois, também provê roteamento de um elemento de processamento para qualquer outro, porém, não há conflitos, o que simplifica o roteamento. No entanto, tem um custo de implementação quadrático em relação ao número de elementos de processamento da arquitetura, restringindo o número máximo de elementos de processamento.

Nos estudos de [Ferreira et al., 2013a] e [Filho & Ferreira, 2012], foi realizada uma avaliação do custo de implementação em FPGA de uma rede *crossbar* para 16 a 24 elementos de processamento, comparada a uma arquitetura ADRES [Mei et al., 2005], que é uma arquitetura em malha. O resultado desta avaliação está mostrado na Tabela 1.2. O custo de área em LUTs [Hauck & DeHon, 2007] para a arquitetura *crossbar* e ADRES são similares. A arquitetura de 16 PEs gera uma

Arquitetura	LUT	RAM	Frequência (Mhz)
Crossbar 16 PEs	8194	7	90
Crossbar 20 PEs	14014	10	72
Crossbar 24 PEs	18391	12	57
ADRES 4x4	6844	25	110

Tabela 1.2. Porcentagem de recursos de FPGA para arquiteturas *crossbar* com 16, 20 e 24 elementos de processamento e um CGRA baseado no ADRES, em um FPGA XILINX xc6vlx75t. Total de LUTs: 46558. Total de RAMs: 156.

pequena sobrecarga em área, mesmo com a complexidade $O(n^2)$. Entretanto, o número de LUTs necessárias para as arquiteturas 20 e 24 PEs cresce de forma significativa assim como diminui a frequência de relógio. Ainda assim, é possível implementá-las em FPGAs de tamanho médio. Em relação ao número de memórias RAM usadas para armazenar os bits de configuração para todas as arquiteturas e o registro global do ADRES, a arquitetura de 16 PEs usa pouco recurso, apenas 7 RAMs comparado ao ADRES, que usa 9 RAMs para para armazenar os bits de configuração somado a 16 RAMs para o banco de registros global.

Essa avaliação mostra que uma arquitetura com rede *crossbar* pode ser implementada em FPGA, com custo viável, o que possibilita a exploração de heurísticas para a compilação JIT, pois simplifica ao máximo o posicionamento e o roteamento. A heurística pode ser executada por um processador *softcore*, que também é implementado em FPGA.

1.2 Objetivos

Esse trabalho tem o objetivo de propor uma solução para acelerar o mapeamento de laços em uma arquitetura CGRA, possibilitando a compilação *Just-In-Time*. A solução é uma combinação de três mecanismos distintos:

- Uma heurística *modulo scheduling* gulosa, simples e eficiente.
- Uma nova arquitetura CGRA, baseada em [Ferreira et al., 2011a], que otimiza o uso de registradores e utiliza redes *crossbar* com roteamento sem conflitos.
- Propor um modelo para mostrar a viabilidade para a implementação do algoritmo em *hardware*.

Objetivos secundários visam explorar o potencial da implementação dinâmica através de:

- Avaliação do comportamento da heurística na presença falhas nos elementos de processamento do CGRA.
- Avaliação do comportamento da heurística na execução de múltiplas aplicações concorrentemente.

1.3 Organização do texto

O trabalho está organizado em capítulos, como descrito a seguir:

- Capítulo 1: contém a introdução para contextualização do estudo, sua motivação e os objetivos.
- Capítulo 2: apresenta o referencial teórico sobre as arquiteturas reconfiguráveis de grão-gosso, abordando as questões estruturais e uma seção sobre os diferentes tipos de configurações para CGRAs. É apresentada também a técnica de Modulo Scheduling e uma revisão de trabalhos importantes que a utilizaram.
- Capítulo 3: apresenta a proposta de solução para mapeamento de laços em CGRAs, na qual é apresentada a arquitetura CGRA utilizada no trabalho, juntamente com a heurística e suas estratégias, além de soluções para suporte a falhas no *hardware* e mapeamento de *multikernels*. São mostrados exemplos da solução proposta.
- Capítulo 4: apresenta um modelo em *hardware* para a heurística proposta, com o objetivo de mostrar a possibilidade de futuras implementações em *hardware*. É proposto um modelo em máquina de estados que pode ser implementado em FPGA.
- Capítulo 5: apresenta a avaliação da solução proposta para laços paralelizáveis de aplicações multimídia. Foram realizados experimentos que avaliaram a heurística, considerando suas estratégias, a qualidade do escalonamento obtido, o tempo de execução, o tamanho da arquitetura, o gerenciamento de falhas na arquitetura, a possibilidade de mapeamento de mais de um laço ao mesmo tempo (*multikernel*) e a implementação do algoritimo de mapeamento em *hardware*.

- Capítulo 6: apresenta a conclusão do estudo, avaliando os resultados obtidos e propondo algumas sugestões para trabalhos futuros para melhoria e ampliação da solução proposta.

2. Referencial Teórico

Este capítulo apresenta o referencial teórico desta dissertação. Em primeiro lugar há uma descrição das arquiteturas de grão-grosso com relação à estrutura e são listadas as principais existentes na literatura. Em seguida foi feito um resumo sobre os modos de configuração dessas arquiteturas. A técnica de *software pipelining* conhecida como *modulo scheduling* é apresentada, com exemplos de trabalhos relacionados. Por fim, há uma seção sobre a complexidade do problema de mapeamento de laços em CGRAs.

2.1 Arquiteturas Reconfiguráveis de Grão-Grosso (CGRAs)

As arquiteturas de grão-grosso, comumente denominadas *Coarse-Grained Reconfigurable Arrays* (CGRAs), surgiram com o principal objetivo de reduzir a complexidade e o tempo de configuração, posicionamento e roteamento das arquiteturas de grão-fino (FPGAs). Todos os CGRAs têm elementos de processamento (PEs[1]) onde são realizadas as operações (soma, multiplicação, etc), uma rede de interconexão que provê ligação entre os PEs e uma memória para a configuração da arquitetura. Os CGRAs variam basicamente com relação à granularidade, tipo das PEs e tipo de rede, como o mostrado nos itens a seguir. Cada característica da arquitetura tem impacto direto no custo de implementação do circuito, no algoritmo de mapeamento e na capacidade de processamento.

- Tipos de elementos de processamento: podem ser homogêneos ou heterogêneos. Quando são homogêneos, todos os PEs realizam qualquer tipo de instrução de um determinado conjunto. Quando são heterogêneos, os PEs realizam operações específicas, por exemplo, alguns PEs realizam soma de dois números, outros realizam *load/store* ou quaisquer outras.

- Rede de Interconexão: os PEs podem ser interconectados de várias formas, sendo as mais comuns mostradas nas Figuras 2.1 e 2.2. Vários trabalhos foram desenvolvidos usando arquitetura em malha (Figura 2.1(a,b)), dentre

[1] do inglês, *processing elements*

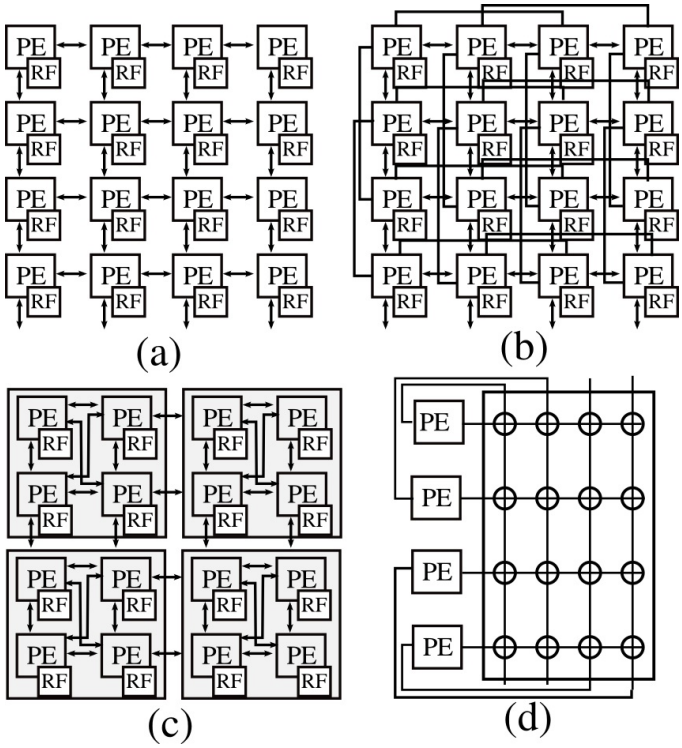


Figura 2.1. Exemplos de redes de interconexão em CGRAs: (a)Malha Simples (b)Malha *Plus* (c)Agrupamentos (d)*Crossbar*.

eles pode-se citar: ADRES [Mei et al., 2005], KressArray [Hartenstein et al., 1994] e RAW [Babb et al., 1997]. Com a topologia baseada em linhas (Figura 2.2(a)) o trabalho mais conhecido é o PipeRench [Goldstein et al., 2000]. As arquiteturas baseadas em agrupamentos, também chamadas de arquiteturas em *clusters*, são na verdade arquiteturas híbridas de malhas divididas em *clusters*, foram utilizadas no WaveScalar [Swanson et al., 2003] e MorphoSys [Singh et al., 2000]. Arquiteturas com redes *crossbar* (Figura 2.1(d)) podemos citar PADDI [Chen, 1992] e, mais recentemente, [Filho & Ferreira, 2012]. Com rede multiestágios (Figura 2.2(b)), [Ferreira et al., 2011a] e [Mucida et al., 2012].

- Granularidade da arquitetura: os PEs e a rede operam com palavras de 8 a 32 bits, como mostra [Hartenstein, 2001]. A granularidade influencia diretamente o tamanho da memória de configuração, pois nos CGRAs a reconfiguração ocorre em nível de palavra e não de bits, como nos FPGAs.

Quando consideram-se os passos de escalonamento, posicionamento e roteamento nos CGRAs, a estrutura da rede de interconexão é crucial. Adicionar recursos de roteamento na arquitetura pode reduzir a complexidade do mapeamento, no en-

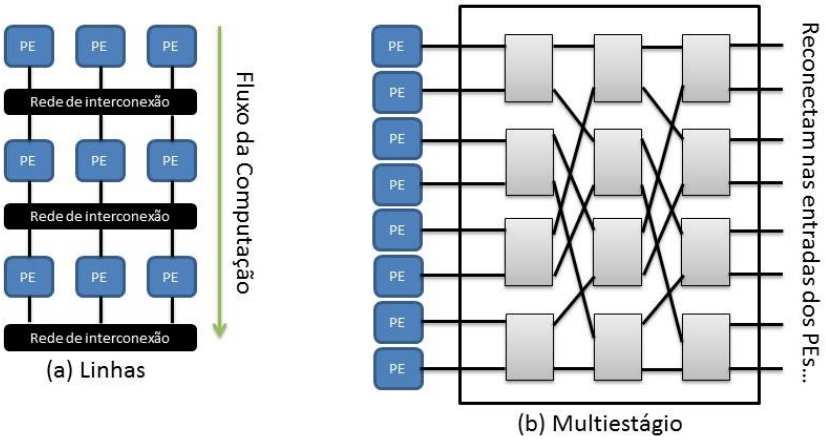


Figura 2.2. Exemplos de redes de interconexão em CGRAs: (a) Linhas (b) Multiestágios.

tanto, a rede de interconexão terá seu custo de implementação aumentado. Assim, há uma relação entre custo da rede de interconexão e tempo de mapeamento.

Por exemplo, a Figura 2.1(a) mostra um CGRA 4x4 onde cada PE tem ligação direta com seus 4 vizinhos (norte, sul, leste, oeste). Além disso, cada PE tem um banco de registros para armazenar valores temporários. Na Figura 2.1(b) é mostrada a interconexão *mesh plus* ou malha melhorada, que adiciona algumas ligações de PEs pulando uma ou mais linhas ou colunas, isso melhora a capacidade de roteamento e simplifica o algoritmo de posicionamento e roteamento. A Figura 2.1(c) mostra uma arquitetura em agrupamentos, onde em cada um é usada a conexão completa (*crossbar*) entre os PEs, e cada PE de fronteira conecta-se ao seu vizinho de agrupamento. A Figura 2.1(d) mostra um CGRA com conexão completa entre PEs ou rede *crossbar*. Esta interconexão simplifica o roteamento, pois todos os PEs podem conectar diretamente e simultaneamente a qualquer outro, e o posicionamento não tem restrição. Porém, a rede *crossbar* tem custo quadrático em relação ao número de PEs, podendo ser usada apenas em casos em que há um número reduzido de PEs, até aproximadamente 32, como mostra [Filho & Ferreira, 2012].

A rede de interconexão multiestágio (Figura 2.2(b)), assim como a *crossbar*, também possibilita conexão entre qualquer PE. Porém, podem acontecer conflitos na rede, quando já existem ligações feitas e é solicitada uma ligação nova, que precisa passar por um caminho que já está ocupado na rede. Ainda assim, a rede multiestágio provê boa solução [Ferreira et al., 2011a], principalmente quando comparadas à *crossbar*, para um número maior de PEs, acima de 32 PEs. Na arquitetura em

linhas (Figura 2.2(a)) a computação flui linha por linha, tendo as conexões de PEs feitas nas redes entre as linhas.

2.2 Configuração de Arquiteturas Reconfiguráveis

A computação reconfigurável, que segundo [Hauck & DeHon, 2007] é o processo de explorar da melhor maneira o potencial do *hardware* reconfigurável, pode ser dividida em dois modos de configuração: estática e dinâmica [Sanchez et al., 1999]. Na configuração estática, pressupõe-se que a configuração seja carregada uma vez e não é alterada durante a execução da tarefa programada. Este modo não provê grande flexibilidade, mas permite atingir alto desempenho para uma determinada aplicação.

Porém muitos ambientes necessitam de maior flexibilidade e para suprir esse requisito, mais de uma configuração podem ser utilizadas ao longo da execução da aplicação. Isso possibilita que vários segmentos da aplicação sejam mapeados na arquitetura especializada e, portanto, mais recursos poderão ser utlizados, resultando em maior flexibilidade. Este modo de configuração é chamado de dinâmico.

As configurações estáticas ou dinâmicas podem ser geradas por um compilador de maneira estática ou durante a execução, conhecido na literatura como *Just-In-Time* - JIT - ou *on-the-fly compilation*. O fato de uma arquitetura ser dinamicamente reconfigurável não significa que sua configuração será gerada durante a execução, mas que a arquitetura poderá mudar de configuração em tempo de execução, porém a compilação de cada configuração pode ter sido gerada estática ou dinamicamente.

A geração estática de configuração supre muito bem os requisitos de um sistema reconfigurável quando os trechos de códigos a serem acelerados têm uma característica determinística de computação, ou seja, o que deve ser feito é bem definido e não surgem possibilidades novas ao longo da execução da aplicação. É necessário conhecer o momento em que a aceleração será realizada, não abrindo espaço para nenhuma capacidade à adaptação num ambiente dinâmico.

A necessidade de maior eficiência no consumo de energia dos sistemas computacionais atuais motivou os pesquisadores a conceber o uso do CGRA como um processador de propósito mais geral, sendo utilizado como co-processador fortemente acoplado, e não somente como um processador embarcado, acelerando um conjunto muito maior de aplicações [Shrivastava et al., 2011]. Isso motiva a compilação JIT, pois há uma demanda dinâmica de aceleração, tanto para a compilação quanto para

a execução.

Compilar dinamicamente proporciona a possibilidade de usar informação que só está disponível em tempo de execução. Cria até mesmo a possibilidade de compilar mais de um *thread* ao mesmo tempo (*multithreading*), como mostra [Shrivastava et al., 2011]. Outra vantagem da configuração JIT é a possibilidade de gerenciar falhas permanentes geradas na fabricação do circuito CGRA. O algoritmo de mapeamento pode ser capaz de identificar falhas e restringir a compilação somente às partes úteis da arquitetura.

Em resumo, o processo de gerar a configuração de forma dinâmica para um CGRA pode ser dividido em duas partes. A primeira é identificar, em tempo de execução, um trecho da aplicação que pode ter sua execução acelerada no CGRA. Compilar diretamente do código binário é uma opção, conhecida como tradução binária, e é usado para executar programas que não foram originalmente compilados para a plataforma. A segunda parte é mapear o trecho para o CGRA, no menor tempo possível. O escopo desta dissertação limitou-se apenas à segunda parte, ou seja, mapear um laço na arquitetura em tempo compatível com o processo de compilação dinâmica.

2.3 *Modulo Scheduling*

Modulo Scheduling (MS) é uma técnica de *Software Pipelining* que foi introduzida por [Rau, 1994] e busca sobrepôr diferentes iterações da execução de um laço, sendo que uma iteração pode começar a sua execução antes do término das iterações anteriores, tentando alcançar um alto nível de paralelismo em nível de instrução. O *Modulo Scheduling* tenta obter o escalonamento mínimo para uma iteração do laço, tal que, quando o escalonamento é repetido, nenhuma restrição de dependência ou recurso é violada.

O intervalo constante entre o início de iterações sucessivas é chamado de Intervalo de Iniciação (II). Um escalonamento válido é aquele em que não há nenhum conflito no uso de recursos entre as operações e nenhuma dependência interna ou externa da iteração é violada. Dependência interna é aquela que ocorre dentro de uma iteração, entre as operações. Dependência externa é quando uma iteração depende de algum resultado de outra.

A vazão do laço é determinada quase integralmente pelo intervalo de iniciação, quando a execução encontra-se no estado em que há iterações iniciando e outras terminando, chamado estado estável. Somado a este também há os estados de

prólogo e epílogo do escalonamento na arquitetura. O prólogo se define como o período entre o início da primeira iteração e o ciclo anterior ao fim dessa iteração, período no qual ainda não há resultados. O epílogo é o período final da execução do laço, onde não há mais iterações começando e somente é necessário continuar executando para terminar a última iteração. Assim, para um número de iterações considerável, a vazão será igual ao intervalo de iniciação.

Um algoritmo MS sempre inicia sua tarefa de escalonamento calculando o II inicial. Geralmente, o II inicial é o mínimo II (MinII), calculado com base na restrição de recursos da arquitetura e na dependência (interna e externa) de operações do laço. Se um escalonamento válido não é encontrado com o II inicial, este valor é incrementado e um novo escalonamento é realizado, com o novo II. A execução do MS pára quando atingir um resultado satisfatório ou quando atinge um valor de limite para o II.

2.3.1 *Modulo Scheduling* para CGRAs

Trabalhos anteriores usaram *modulo scheduling* para resolver o problema do escalonamento em processadores VLIW [Rau, 1994; Sánchez & González, 2000; Yun & Kim, 2001; Zalamea et al., 2001; Guo, 2005]. O escalonamento MS para CGRAs possui alguns desafios adicionais, quando comparado ao escalonamento MS para arquiteturas VLIW, como mostra [Park et al., 2008]. Essas dificuldades adicionais se devem às diferenças de *hardware* e podem ser listadas como:

- roteamento explícito e inteligente: assim como nos FPGAs, os CGRAs precisam especificar explicitamente como os valores serão roteados até seus destinos. O roteamento pode ser feito através de redes de interconexão, registros locais ou elementos de processamento. Numa arquitetura VLIW, por exemplo, o roteamento é garantido implicitamente por registros centralizados;

- operações heterogêneas: todos os PEs podem realizar operações lógicas e adição, porém operações com maior custo de implementação em *hardware*, como multiplicação, *load* e *store*, podem estar disponíveis apenas em um subconjunto de PEs. Em arquiteturas deste tipo (com PEs heterogêneos), é importante evitar o posicionamento de operações simples, como adição, em PEs mais complexos, porque isso limita a flexibilidade das operações que necessitam de PEs mais complexos;

- restrição modular: os recursos são utilizados de maneira periódica, uma vez que um trecho de código repete a cada ciclo do II. Assim, ao contrário do

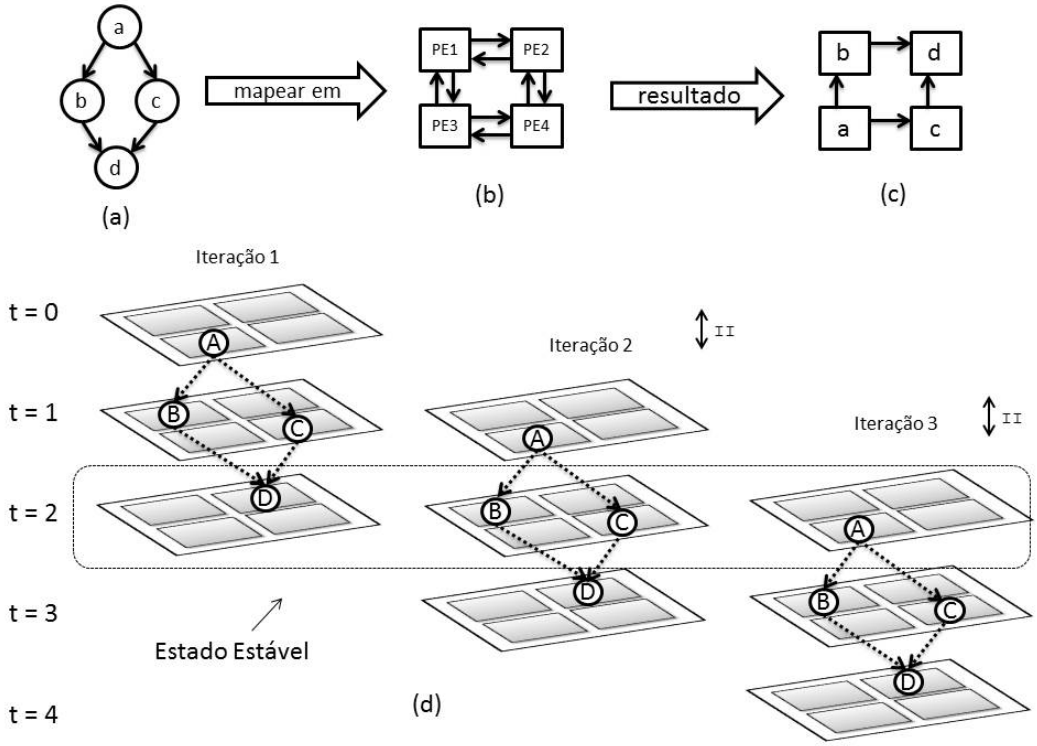


Figura 2.3. Exemplo de algoritmo *modulo scheduling*: (a) Grafo de dependências (b) Arquitetura CGRA (c) Resultado do mapeamento (d) Execução detalhada. Adaptado de [Mei et al., 2005].

escalonamento não periódico, não é possível garantir capacidade de roteamento da arquitetura apenas estendendo o escalonamento e este poderá falhar por causa de operações anteriormente escalonadas.

Para ilustrar o problema, pode-se considerar um grafo de dependências simples que representa o código de um laço (Figura 2.3(a)) e um CGRA em malha 2x2 (Figura 2.3(b)). O resultado do algoritmo MS é mostrado na Figura 2.3(c). A Figura 2.3(d) é uma representação espaço-temporal do mapeamento obtido.

O *modulo scheduling* para CGRAs é uma combinação de três subproblemas: escalonamento, posicionamento e roteamento. O escalonamento determina em que ciclo será executada cada operação. O posicionamento determina em qual elemento de processamento estará posicionada. O roteamento conecta as operações mapeadas. A seguir um exemplo de algoritmo MS será apresentado, para explicar o resultado da Figura 2.3(c).

Como dito anteriormente, o algoritmo MS começa sua execução calculando o intervalo de iniciação mínimo (MinII). Neste exemplo, suponhamos que o MinII é

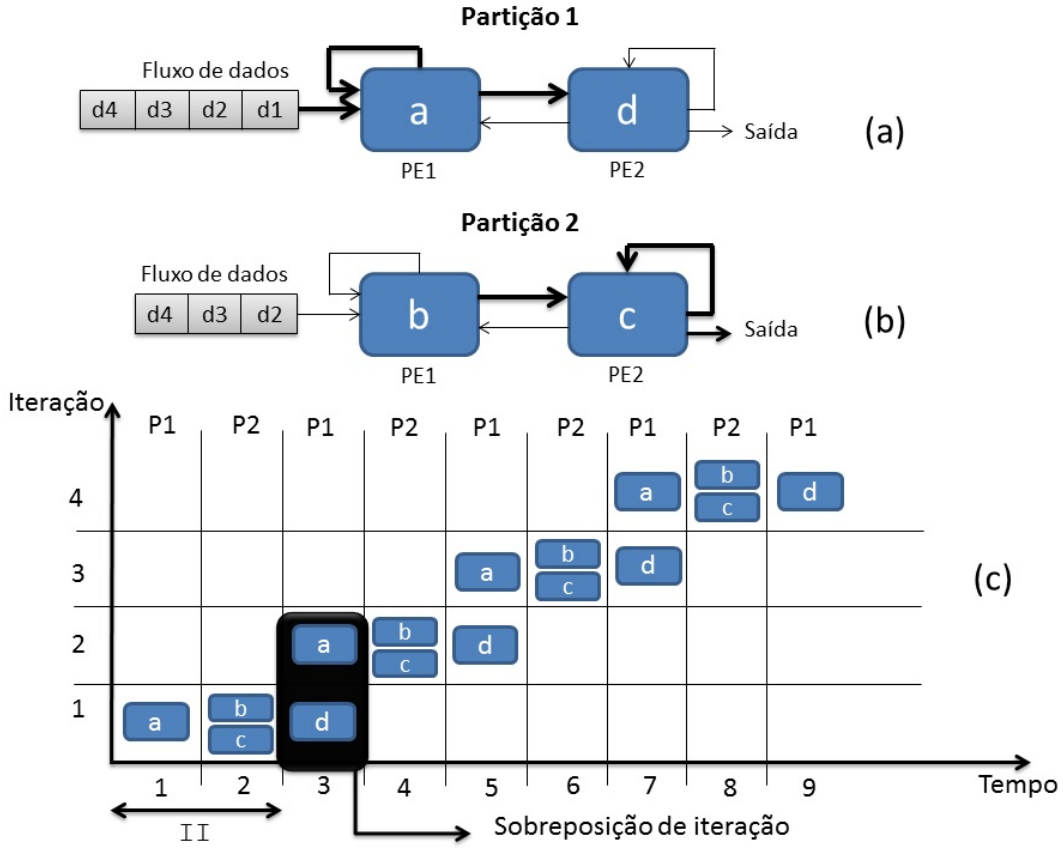


Figura 2.4. Exemplo de algoritmo *modulo scheduling*: (a)Partição 1 (b)Partição 2 (c)Paralelismo espacial com sobreposição temporal e particionamento.

calculado apenas considerando o número de operações a serem mapeadas dividido pelo número de PEs disponíveis, o resultado seria 1 (4 operações em 4 PEs). Assim o algoritmo seguiria para seu núcleo, começando pelo escalonamento das operações do grafo. A operação a seria escalonada no tempo 0, as operações b e c são dependentes de a e somente dela, sendo escalonados no tempo 1. A operação d depende de b e c então é escalonada no tempo seguinte, 2.

Feito o escalonamento, vem a etapa de posicionamento, onde usualmente começa-se posicionando a operação de menor tempo. Assim a pode ser mapeada aleatoriamente em qualquer PE, suponha o PE_3. Agora, b e c podem ser posicionadas. O algoritmo tenta posicioná-las o mais próximo possível de a, ficando nos PEs 1 e 4 respectivamente. O posicionamento de d segue a mesma lógica. Por fim, o roteamento conecta as operações dependentes, usando os caminhos disponíveis. Neste ponto já temos um *modulo scheduling* válido para o grafo. O algoritmo conseguiu escalonar com II igual 1, assim, a cada novo ciclo, uma nova iteração será iniciada.

Na Figura 2.3(d) podemos ver que a execução se inicia somente com a operação a sendo executada (t=0) e, em seguida, (t=1) há também b e c. No tempo 2, todas as operações estão executando, caracterizando o estado estável, com três iterações em pipeline: todas os elementos de processamento estão ocupadas e em iterações diferentes.

Sobrepor diferentes iterações de um laço nos torna capazes de alcançar um nível mais alto de ILP (*instruction level paralelism*). Neste exemplo, o ILP é 4, maior do que o 1,33 alcançado pela mesma arquitetura sem sobreposição de iterações.

No exemplo da Figura 2.3, o número de PEs disponíveis é suficiente para posicionar todas as operações do grafo. No entanto, na maioria dos casos, o número de PEs é menor que o número de operações a serem mapeadas, sendo necessário o uso de partições temporais, ou seja, intervalo de iniciação maior que 1.

Outro exemplo está na Figura 2.4, onde a arquitetura tem apenas dois elementos de processamento, mas é necessário o cálculo de 4 operações, mesmo grafo da Figura 2.3. Assim, o particionamento temporal é usado de modo que a cada ciclo de tempo é trocada a partição. No exemplo, o PE1 recebe o dado $d1$ na partição 1, e passa para o próprio PE1 e para o PE2, na partição 2. As PE1 e PE2 calculam e repassam o valor, sendo que o PE1 manda para PE2 e este para si mesmo, na partição 1. O gráfico 2.4(c) mostra o cálculo do fluxo de dados através do tempo e das partições. Tem-se uma latência igual a 3 e uma vazão igual a 0,5, que corresponde ao $II = 2$.

Um problema comum é a impossibilidade de roteamento. Para ilustrar, consideremos o grafo de fluxo de dados de um laço mostrado na Figura 2.5(a). Neste exemplo, a arquitetura usada é novamente malha com 4 PEs e o grafo tem 12 nós, logo, o escalonamento precisará ser feito em pelo menos 3 partições temporais, denominadas $pt0$, $pt1$, e $pt2$ na Figura 2.5. Os nós a e b são posicionados nos PE_0 e PE_1 na partição $pt0$. Os nós c e d devem ser posicionados em $pt1$ devido às suas dependências de a e b. A Figura 2.5(c) mostra que c e d foram posicionados nos mesmos PEs que a e b e o roteamento é possível. Para e, f e g, um posicionamento e roteamento possível na partição $pt2$ é mostrado na Figura 2.5(c). Entretanto, quando o nó h é posicionado na PE_2 na $pt0$, não é possível rotear de f na PE_1 para h no PE_0, pois não há conexões diagonais na arquitetura da Figura 2.5(b). O roteamento $f \rightarrow h$ falha, como mostrado na Figura 2.5(d).

Uma solução é mudar o posicionamento de e, f e g na $pt2$. Assim, e e f podem rotear para h, da partição $pt2$ para $pt0$, assim como o grafo todo: $g \rightarrow i$, $h \rightarrow j$, $i \rightarrow k$, e $k \rightarrow l$, como mostra a Figura 2.5(e). Portanto, várias alternativas para posicionamento e roteamento podem ser avaliadas durante o processo de

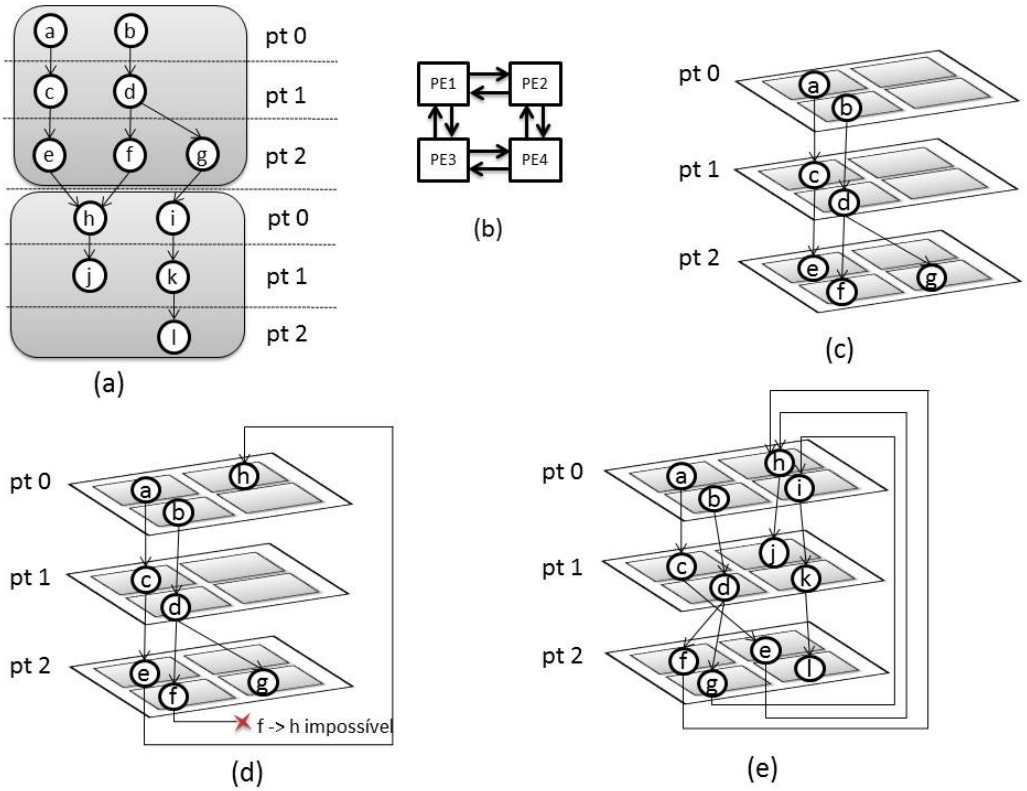


Figura 2.5. Exemplo de falha no roteamento (a)Grafo dataflow dividido em partições temporais (b)Arquitetura em malha 2x2 (c)Posicionamento Inicial (d)Falha no roteamento de f para h (e)Falha solucionada por reposicionamento de operações.

escalonamento e a rede de interconexão influencia diretamente.

2.3.2 Trabalhos relacionados

O primeiro trabalho a adaptar o MS para CGRAs foi realizado por [Callahan & Wawrzynek, 2000], aplicando o algoritmo a uma arquitetura reconfigurável 1D. Porém, o compilador DRESC, descrito por [Mei et al., 2002], obteve maior impacto na área, sendo frequentemente citado como pioneiro.

DRESC baseia sua solução na técnica *simulated annealing*. Para cada intervalo de iniciação, o algoritmo gera um escalonamento e posicionamento inicial das operações, satisfazendo as restrições de dependência, mas podendo sobrecarregar a utilização dos PEs. Por exemplo, duas operações podem estar posicionadas num PE no mesmo ciclo. Usando o *simulated annealing*, o algoritmo iterativamente tenta reduzir a sobrecarga de recursos, posicionando aleatoriamente uma operação num novo PE, refazendo o roteamento das unidades. Neste momento o algoritmo usa

uma função que avalia o custo do novo posicionamento e roteamento e o compara com o custo encontrado na iteração anterior, se a iteração corrente não for a primeira. Se o custo for menor, o novo posicionamento e roteamento poderá ou não ser aceito, dependendo da "temperatura", que é uma heurística que evita mínimos locais. A temperatura fica cada vez mais baixa com o passar do tempo, o que significa que ficará cada vez mais difícil mover uma operação. Se o tempo atingir o limite, o algoritmo reinicia com novo II.

O compilador DRESC frequentemente alcança bons resultados em termos de escalonamento, obtendo frequentemente o mínimo II [Mei et al., 2003]. Porém, o tempo é um fator limitante do DRESC, chegando a precisar de 15 minutos para escalonar um grafo de 80 operações. Isso impossibilitaria uso desse compilador para mapear um trecho de código em tempo de execução, que é um dos objetivos desta dissertação.

A solução proposta por [Park et al., 2006] apresenta uma heurística que alcança um resultado de escalonamento pouco inferior ao DRESC em termos de II, porém a heurística pode ser utilizada para uma variedade de CGRAs em malha de custo menor que DRESC. Além disso, o tempo de compilação é reduzido à ordem de poucos segundos, de 1 a 5. A heurística SPKM, desenvolvida por [Yoon et al., 2008], em média, consegue um tempo de execução em torno de 1 segundo, alcançando resultado ótimo em 72% dos casos. SPKM é comparado com uma solução de programação linear, que encontra uma solução ótima, gastando mais de 24 horas em média, mas não é comparado com soluções anteriores da literatura. Outro trabalho foi proposto por [De Sutter et al., 2008], utilizando uma arquitetura 4x4 com um banco de registradores compartilhados por todos os PEs e 12 registradores locais. O algoritmo MS proposto foi voltado para otimização da utilização desses registradores. O tempo do escalonamento é da ordem de dezenas a centenas de segundos, dependendo do número de operações. O trabalho também serviu para mostrar o impacto dos registradores no desempenho de um CGRA.

Com o objetivo de reduzir mais o tempo de compilação foi proposto o *Edge-Centric Modulo Scheduling* (EMS) [Park et al., 2008]. Enquanto DRESC e outros algoritmos MS [Park et al., 2006; Hatanaka & Bagherzadeh, 2007; Yoon et al., 2008] propunham primeiro posicionar a operação para depois rotear, abordagem baseada no posicionamento do nó, *Node-Centric*, o EMS propôs orientar o posicionamento através do roteamento, abordagem baseada no roteamento da aresta, *Edge-Centric*. Assim, o posicionamento somente será realizado quando as informações sobre o roteamento estiverem disponíveis. EMS mostrou-se mais rápido que o DRESC pois evita muitas tentativas de posicionamento inviáveis de serem roteadas. Porém essa

alternativa gulosa tende a ser vulnerável a mínimos locais, necessitando de outras técnicas para lidarem com o problema. O algoritmo EMS consegue ser até 19x mais rápido que o DRESC, porém a qualidade do escalonamento pode cair para 85% do II alcançado pelo DRESC. Outro trabalho que usa a abordagem *Node-Centric*, é RAMS, descrito por [Oh et al., 2009]. RAMS também obtém ganho em tempo de compilação quando comparado com DRESC. Obtém escalonamentos melhores que EMS, porém é duas vezes mais lento.

Recentemente, foram propostos dois algoritmos EPIMap [Hamzeh et al., 2012] e Graph Minor [Chen & Mitra, 2012]. O EPIMap usa uma arquitetura 4x4, onde frequentemente pode ocorrer um problema de falta de PEs para posicionar todas as operações de um multicast. Para resolver o problema, o algoritmo usa uma heurística chamada de recomputação. Essa heurística posiciona uma operação mais de uma vez a fim de prover roteamento para um multicast e usar o espaço de forma mais eficiente, conseguindo intervalos de iniciação menores que EMS, levando a uma perfomance da execução no CGRA 2x mais rápida. No entanto, o tempo que EPIMap gera para mapear é em média 6x maior do que EMS.

Graph Minor usa os bancos de registros e registradores locais como recursos de roteamento e os usa para compartilhar roteamentos correspondentes a diferentes arestas que possuem um mesmo nó pai, podendo usar somente 1 registro para repassar vários valores entre PEs. O algoritmo MS proposto melhora a alocação dos recursos e alcança o mesmo II comparado ao DRESC, porém num tempo médio até 4 ordens de grandeza menor, próximo de 3 segundos. Também alcança melhor resultado em termos de utilização dos recursos da arquitetura, 62% de média contra 54% do DRESC.

A fim de produzir um algoritmo com tempo de execução baixo que pudesse ser utilizado por ferramentas dinâmicas para *hardware* reconfigurável, como compiladores *Just-In-Time* ou reconfiguração em tempo de execução, foi proposta uma arquitetura CGRA virtual, implementada sob um FPGA, e um algoritmo de mapeamento, por [Ferreira et al., 2011a]. Essa arquitetura tem um conjunto de PEs conectadas a uma rede global multiestágio, que permite que um PE seja conectado a qualquer outro, simplificando o processo de posicionamento e roteamento, entretanto podem ocorrer conflitos na rede. O algoritmo para tal arquitetura consegue mapear em tempo da ordem de milissegundos. No trabalho desta dissertação, melhora-se os resultados de [Ferreira et al., 2011a], propondo-se mudanças na arquitetura e no algoritmo.

2.4 NP-Completude

O processo de mapeamento para CGRAs consiste em três passos: escalonamento, posicionamento e roteamento. Posicionamento e roteamento por si sós são problemas NP-Completos para redes de interconexão em malha [Ahn et al., 2006; Yoon et al., 2008]. A malha tem um custo de conexão $O(N)$. Quando a malha é substituída por uma rede *crossbar*, o custo sobe para $O(N^2)$, entretando o posicionamento e roteamento tornam-se $O(1)$, não importando *onde* a operação é posicionada, pois o roteamento é uma simples atribuição de valor.

Apenas recentemente o problema de mapear um laço em um CGRA foi provado ser NP-Completo, por [Hamzeh et al., 2012]. Embora o CGRA em [Hamzeh et al., 2012] ser definido como um vetor de elementos de processamento interconectados por um *grid* 2-D ou interconexão em malha, a prova é baseada num CGRA como um grafo geral. O mapeamento é descrito como o problema de encontrar um subgrafo em um grafo CGRA extendido no tempo (TEC - como os das Figuras 2.3 e 2.5) que é epimórfico em relação ao grafo de entrada. Portanto, uma vez que o CGRA com *crossbar* também é modelado como um TEC, onde a restrição de escalonamento deve ser satisfeita, o problema permanece NP-Completo. Na rede *crossbar* não há restrição de posicionamento e roteamento, mas ainda há a possibilidade de acelerar o processo de mapeamento, investindo em estratégias para o escalonamento, reduzindo ainda mais o espaço da solução. Essas heurísticas devem sempre balancear os benefícios de um escalonamento de qualidade com a complexidade do tempo.

2.5 Estrutura Geral

Um grupo de pesquisa do Departamento de Informática da Universidade Federal de Viçosa vem desenvolvendo trabalhos com arquiteturas reconfiguráveis envolvendo os problemas de: posicionamento [Ferreira et al., 2009a, 2007, 2011a, 2013b], roteamento de interconexões [Vendramini & Ferreira, 2010; Ferreira et al., 2011b, 2008, 2009b], tolerância a falhas [Ferreira et al., 2011c], escalonamento [Ferreira et al., 2011a], tradução binária [Ferreira et al., 2008, 2009b, 2011c]. Este trabalho faz parte da linha de pesquisa em escalonamento, posicionamento e roteamento de *loops* nas arquiteturas em tempo de execução [Filho & Ferreira, 2012; Mucida et al., 2012; Ferreira et al., 2013a]. Mais especificamente, esta dissertação faz parte de um trabalho onde o objetivo é propor um ambiente de aceleração de laços com computação intensiva, implementado em FPGA. A Figura 2.6 mostra o ambiente.

O ambiente está sendo desenvolvido em duas implementações diferentes, uma com o algoritmo de mapeamento implementado para executar em um processador *softcore*, chamado de MSG, e outra com o algoritmo implementado diretamente no *hardware* do FPGA, chamado de EPR. Ambas as abordagens visam um ambiente de mapeamento dinâmico para laços paralelizáveis, em tempo de execução.

A execução das aplicações será feita pelos dois processadores indicados na Figura 2.6, o processador de propósito geral *softcore* e o acelerador CGRA de laços paralelizáveis, ambos programados no FPGA. Um compilador externo gera as intruções para o *softcore*, que as executa. Esse processador identifica os laços que podem ser acelerados no CGRA e, em tempo de execução, executa o algoritmo de mapeamento para o CGRA. A diferença está na forma que a heurística estará implementada.

No MSG, a execução do algoritmo de mapeamento acontece dentro do próprio *softcore* e este transfere o resultado para a memória de configuração do CGRA, assim pode acontecer a aceleração no CGRA. No EPR, o algoritmo terá uma área no FPGA, na qual estará implementado com uma máquina de estados. O laço que se deseja acelerar é enviado ao EPR, que executa o algoritmo de mapeamento e segue, a partir daí, as mesmas etapas da outra versão. É necessário notar que os dados que a aplicação usa são compartilhados pelo *softcore* e o CGRA.

O custo e as possibilidades de implementação dos CGRAs em FPGA estão sendo avaliados, tendo sua avaliação inicial publicada em [Filho & Ferreira, 2012; Ferreira et al., 2013a]. As duas implementações para a heurística *modulo scheduling* foram avaliadas nesta dissertação (MSG) e em [Mucida et al., 2012; Mucida, 2013] (EPR), respectivamente. Como o objetivo é executar o algoritmo de mapeamento em tempo de execução, ambas as abordagens são baseadas em heurísticas gulosas que percorrem o grafo de fluxo de dados do laço apenas uma vez. O grafo é percorrido em largura.

A abordagem EPR visa uma implementação em *hardware* do mapeamento composto pelas três operações: escalonamento, posicionamento e roteamento (EPR). O grafo é descrito como uma lista de aresta que simplifica a implementação em *hardware* com um modelo em máquina de estados do algoritmo. Uma solução inicial para o algoritmo foi simulado e validado em [Mucida et al., 2012] e implementado em VHDL em [Filho & Ferreira, 2012].

A abordagem MSG, solução proposta nesta dissertação, faz o processamento do grafo por vértice e foi implementada para compilação *Just-in-Time*. O algoritmo de mapeamento foi implementado na linguagem de programação C, para executar no *softcore*. Esta alternativa permite que a heurística seja mais complexa, oferecendo

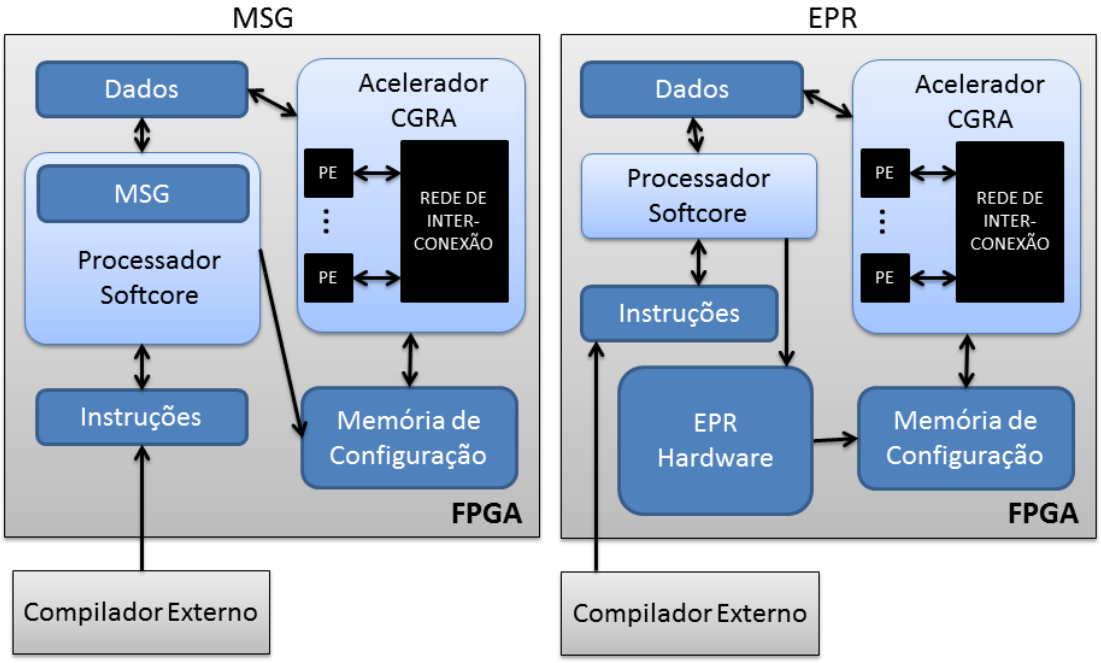


Figura 2.6. Ambiente de aceleração de laços com computação intensiva, em duas abordagens diferentes: à esquerda com o algoritmo de mapeamento MSG implementado para executar sequencialmente no processador *softcore* (MSG) e à direita com o algoritmo EPR implementado em *hardware* (EPR).

possibilidade de melhores mapeamentos, em comparação com a versão em *hardware*. Em contrapartida, a implementação em *hardware* pode ser mais rápida. Nesta dissertação também há um modelo de máquina de estados para a heurística que pode ser implementado em VHDL em um trabalho futuro. Além disso, a abordagem MSG realizou uma redução no número de registradores com *multicasting* utilizados no mapeamento e propõe uma modificação na arquitetura CGRA, que possibilita o melhor uso dos registros de cada elemento de processamento.

3. Heurística Polinomial para Escalonamento de *Loops*

Este capítulo detalha o algoritmo heurístico *modulo scheduling* proposto nesta dissertação, considerando um CGRA com rede de interconexão *crossbar*, em substituição à rede multiestágio proposta por [Ferreira et al., 2011a]. Além disso, duas técnicas de economia de registros que melhoram a qualidade do escalonamento foram propostas. A primeira técnica, chamada compartilhamento de registros, visa economizar o uso de registros, fazendo com que registros sejam reutilizados, compartilhando os valores que armazenam e evitando o uso desnecessário de registros. A outra técnica, chamada de registros locais, propõe uma pequena mudança na arquitetura do CGRA para possibilitar o uso de registros que ficavam ociosos. A heurística *modulo scheduling* proposta será chamada de heurística MSG, em referência a Modulo Scheduling Guloso. Também são apresentadas seções sobre como a heurística pode ser tolerante a falhas e como pode mapear *multikernels*. Por fim, exemplos de execução do algoritmo são apresentados.

3.1 Arquitetura Base

A arquitetura CGRA base deste trabalho é derivada de [Ferreira et al., 2011a]. A Figura 3.1 mostra uma versão dessa arquitetura com apenas 2 PEs, normalmente são usados de 16 a 32 PEs. Cada PE da arquitetura é capaz de realizar qualquer operação e tem seus operandos nos registros de entrada, indicados por R0a,R1a,R0b,R1b na Figura 3.1. Um número limitado de PEs têm também a capacidade de operar sobre valores externos, chamados de PEs de entrada, não indicados na figura. Dessa forma, os PEs realizam a operação indicada, passam o resultado para a rede de interconexão e esta repassa para os registros de destino, referentes ao PE de destino. Não são feitas considerações sobre saída de dados nesta solução.

A arquitetura original tem como rede de interconexão duas redes multiestágios, indicadas por Rede A e Rede B, na Figura 3.1. Porém, com o objetivo de tornar o roteamento mais simples, a rede de interconexão foi trocada para *crossbar* nesta solução. Apesar de o custo da rede *crossbar* ser $O(n^2)$, mais caro que $O(nLog(n))$ da rede multiestágio, o custo total da arquitetura pode ser menor devido à redução

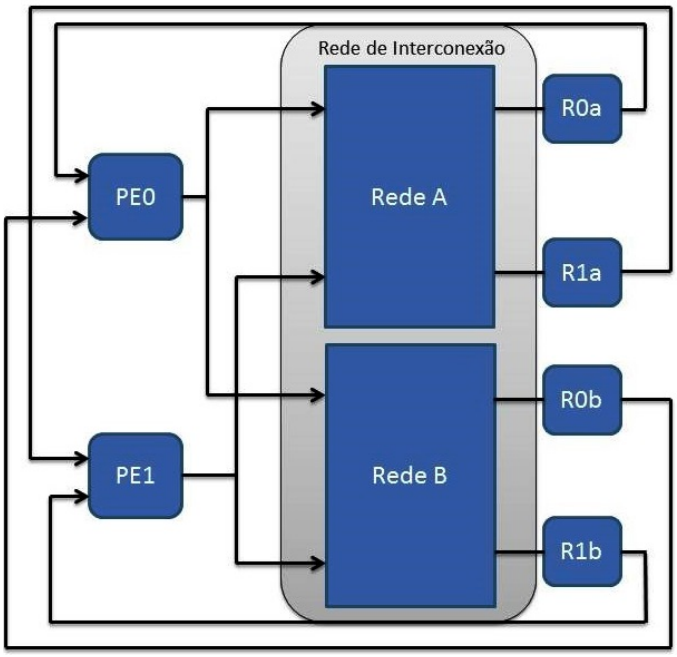


Figura 3.1. Arquitetura base, por [Ferreira et al., 2011a]

do número de PEs necessários, pois a rede *crossbar* não tem conflitos de roteamento, podendo usar intervalos de iniciação menores em arquiteturas com mesmo número de PEs. Além da mudança na rede de interconexão, foi proposta uma outra mudança, descrita na seção 3.4, com o objetivo de otimizar o uso dos registros dos PEs.

3.2 Algoritmo MSG Base

O algoritmo MSG é baseado numa heurística gulosa composta apenas por estruturas de dados simples (vetores) e com complexidade $O(n)$. O pseudo-código é mostrado na Algoritmo 1. É importante lembrar que o pseudo-código se assemelha a um código C, e sua implementação é direta e muito eficiente. A arquitetura utilizada possui 16 elementos de processamento, que incluem uma unidade funcional (FU) para cálculo e dois registros locais conectados à cada FU, e duas redes de interconexão *crossbar* na qual é possível conectar a saída de qualquer FU à entrada de qualquer dos registros locais, como mostrado na Figura 3.1. Algumas variações serão explicadas ao longo do texto.

Similar a outros trabalhos [Hamzeh et al., 2012], o grafo de entrada é percorrido em largura. Entretanto, o grafo não é balanceado, e registros são inseridos ao longo da execução do algoritmo. Neste trabalho, assume-se que o ponto de partida é o grafo de fluxo de dados, similar a outras abordagens [Hamzeh et al., 2012; Park et al.,

2008; Oh et al., 2009; Chen & Mitra, 2012; Hatanaka & Bagherzadeh, 2007]. Além disso, o algoritmo MSG é baseado numa heurística gulosa onde o grafo é percorrido apenas uma vez para cada valor do II, o que reduz o tempo de compilação. Em contraste, outras abordagens tentam diferentes escalonamentos, posicionamentos e roteamentos para cada II. Se o mapeamento não é válido, o II é incrementado e o mapeamento é feito novamente até que uma solução válida seja encontrada. Embora o algoritmo deste trabalho somente tente uma vez para cada II, encontra resultado ótimo na primeira tentativa para 10 dos 15 grafos testados.

O código do algoritmo MSG base usa seis vetores e seus tamanhos variam em função dos seguintes parâmetros: número de elementos de processamento da arquitetura (U), número máximo de partições temporais da arquitetura ou número máximo de configurações (C), número máximo de operações (vértices) do grafo de fluxo de dados a ser mapeado (N). Adiante serão descritos mais dois vetores necessários para a implementação de estratégias de otimização para MSG base.

- Vetor **Posição**: armazena em qual elemento de processamento está posicionada cada operação, sem dizer em qual divisão temporal está. Dada uma operação, o vetor retorna em qual elemento de processamento está mapeada, se não estiver mapeado ainda, retorna o valor que representa vazio. A complexidade em espaço de memória é $O(N)$.

- Vetor **Escalonamento**: armazena o escalonamento temporal de cada operação. Dada uma operação, retorna em qual tempo está escalonada, se não houver escalonamento para a operação, retorna vazio. Esse tempo corresponde exatamente ao nível de profundidade do grafo escalonado. A complexidade em espaço de memória é $O(N)$.

- Vetor **Partição**: armazena em qual partição temporal está cada operação. Dada uma operação, retorna qual é a partição temporal, se houver, senão retorna vazio. A complexidade em espaço de memória é $O(N)$.

- Vetores **Roteamento**: são dois vetores que armazenam o valor do elemento de processamento que será ligada à rede de interconexão numa determinada divisão temporal. São dois vetores, pois, em cada elemento de processamento da arquitetura CGRA há duas entradas, logo, é necessário um vetor para armarzenar cada entrada, de cada unidade. Em resumo, armazena toda a rede de interconexão *crossbar* da arquitetura. A complexidade em espaço de memória é $O(U \times C)$.

- Vetor **Unidade_Livre**: armazena qual é o próximo elemento de processamento livre em cada partição temporal. Ao fim do algoritmo, representa o número total de unidades ocupadas, por divisão. A complexidade em espaço de memória é $O(C)$.

Algorithm 1 Algoritmo MSG Base

```
1:  for cada nó t, com fontes s1 e s2 do
2:    if s1 e s2 são entradas externas de dados then
3:      pt = Inicial;
4:      Posição[t] = Unidade_Livre[pt]++;
5:      Escalonamento[t] = Partição[t] = pt;
6:    else
7:      if Escalonamento[s1] == Escalonamento[s2] then
8:        pt = (Partição[s1] + 1) mod II;
9:        pe = Posição[t] = Unidade_Livre[pt]++;
10:       Escalonamento[t] = Escalonamento[s1] + 1;
11:       Partição[t] = pt;
12:       RoteamentoA[pe][pt] = Posicionamento[s1];
13:       RoteamentoB[pe][pt] = Posicionamento[s2];
14:     else
15:       reg = Insere_Registro(Escalonamento[s1],Escalonamento[s2]);
16:       pe = Posição[t] = Unidade_Livre[pt]++;
17:       Partição[t] = pt;
18:       RoteamentoA[pe][pt] = reg;
19:       RoteamentoB[pe][pt] = Posição[s2];
20:     end if
21:   end if
22: end for
```

A travessia do grafo é executada por nó, considerando-se t o nó destino, e $s1$ e $s2$ os nós de origem. Se t tem entradas de dados externas (linhas 2 a 5 no Algoritmo MSG Base), o escalonamento, posicionamento e roteamento são diretos. É importante notar que o posicionamento é realizado sequencialmente usando uma simples atribuição (linha 4) pois não há conflito para o roteamento na rede *crossbar*, sendo uma operação de custo $O(1)$. Se $s1$ ou $s2$ já foram escalonados, há duas possibilidades. Primeiro, $s1$ e $s2$ tem o mesmo escalonamento e nenhum balanceamento é necessário (linhas 7 a 13), assim t é escalonado na próxima partição temporal, usando a operação *modulo* (linha 8). O roteamento é realizado apenas por duas simples atribuições (linhas 12 e 13). A segunda possibilidade ocorre quando $s1$ e $s2$ têm diferentes escalonamentos. Assim, um ou mais registros são inseridos para

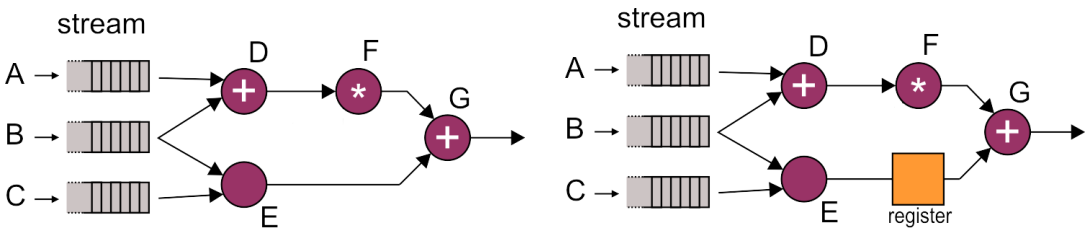


Figura 3.2. Grafo de fluxo de dados não balanceado (esquerda) e balanceado (direita).

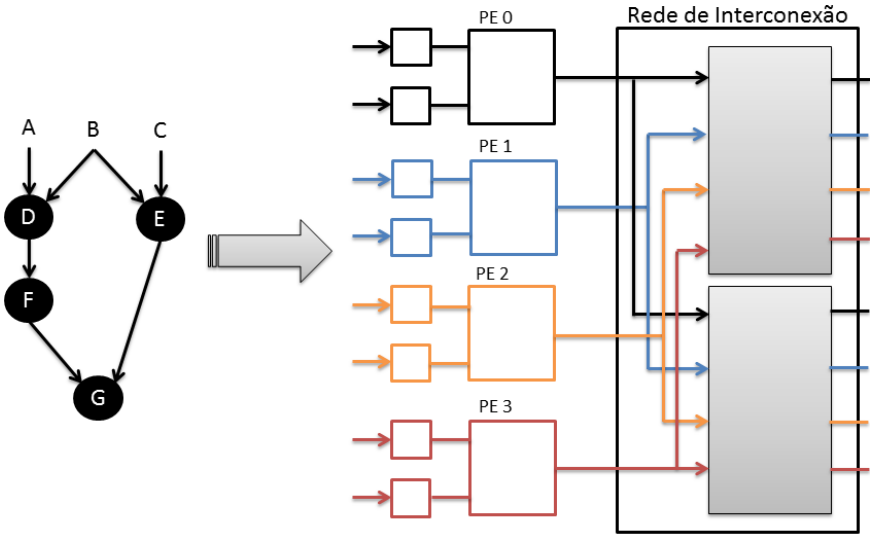


Figura 3.3. Arquitetura com 4 PEs (à direita), sendo os dois primeiros com capacidade de entrada externa de dados, e grafo de fluxo de dados a ser mapeado (à esquerda).

balancear o caminho de dados (linha 15), e em seguida, o escalonamento, posicionamento e roteamento são realizados.

3.2.1 Exemplo: Heurística MSG Base

O exemplo a seguir mostra a execução da heurística MSG base, que mapea para arquitetura base, apresentada nas Figuras 3.1(a) e 3.10. Para começar, temos o grafo mostrado na Figura 3.2 (à esquerda), onde A, B e C são fluxos de dados de entrada e D, E, F e G são operações que utilizam os dados provenientes do fluxo de dados (D e E) ou de cada operação anterior (F e G). Este grafo será a entrada do algoritmo e será percorrido em largura. A arquitetura utilizada neste exemplo tem 4 PEs, sendo 2 com possibilidade de entrada externa de dados (*streams*). A Figura 3.3 mostra a arquitetura alvo e o grafo a ser mapeado.

Inicialmente, o algoritmo precisa definir qual é o II mínimo inicial ($InitII$), ou

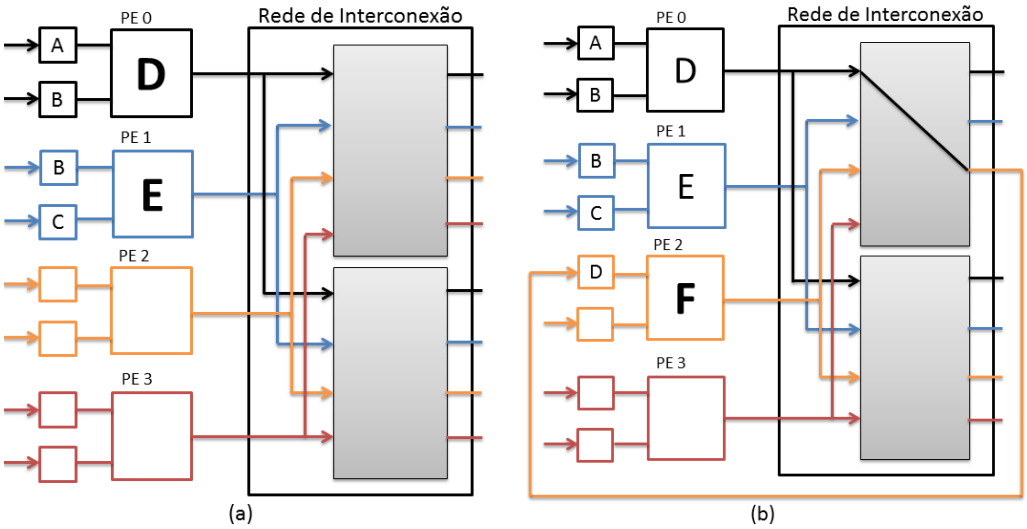


Figura 3.4. Posicionamento das três primeiras operações (a) com entrada externa de dados (D e E) e (b) adicionando a operação F e fazendo o roteamento de $D \rightarrow F$.

seja, o mínimo de partições temporais que serão necessárias para escalonar corretamente o grafo na arquitetura, considerando apenas o número de operações do grafo e o número de operações que têm entrada externa. É necessário notar que o $InitII$ pode não ser igual ao $MinII$, pois o $InitII$ não considera que pode ser necessária a utilização de PEs como registros de balanceamento. No caso deste exemplo, há 4 operações a serem mapeadas na arquitetura com 4 PEs, sendo 2 operações com entrada externa de dados, que é exatamente o que a arquitetura provê. Logo, $InitII = 1$.

Como a heurística MSG varre o grafo em largura, as primeiras operações a serem mapeadas serão D e E. Assim, D é posicionada no PE_0 e E é posicionada no PE_1, como mostra a Figura 3.4(a). Em seguida, F é posicionada no PE_2 e é realizado o roteamento da aresta $D \rightarrow F$ na rede, mostrado na Figura 3.4(b). A última operação a ser mapeada é G, neste caso, necessita de roteamento de duas arestas $E \rightarrow G$ e $F \rightarrow G$. O algoritmo então faz o teste para identificar em que tempo (vetor Escalonamento da heurística MSG base) estão as operações que vão prover a entrada para G. Então, é identificado que E está uma unidade de tempo atrás de F. Logo, será necessário um registro para balancear as entradas de G, como mostrado na Figura 3.2(à direita). Esse registro é alocado no primeiro registro local do PE_3, para a arquitetura base usa-se somente o primeiro registro local, A. Feito isso, G poderá ser posicionado. Porém, é identificado que não há mais PEs livres para alocar G. Logo, o escalonamento falha para o $II = 1$. A Figura 3.5 demonstra

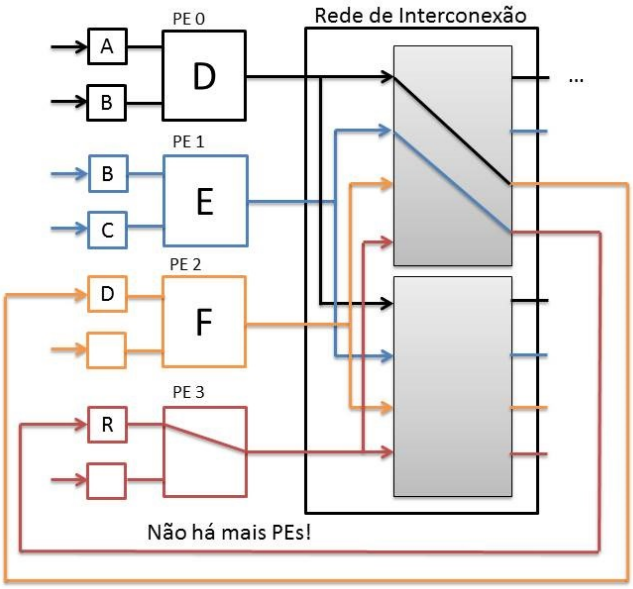


Figura 3.5. Estado final do mapeamento para $II = 1$. Falta posicionar G, porém não há mais PEs disponíveis. É necessário recomeçar com $II = 2$.

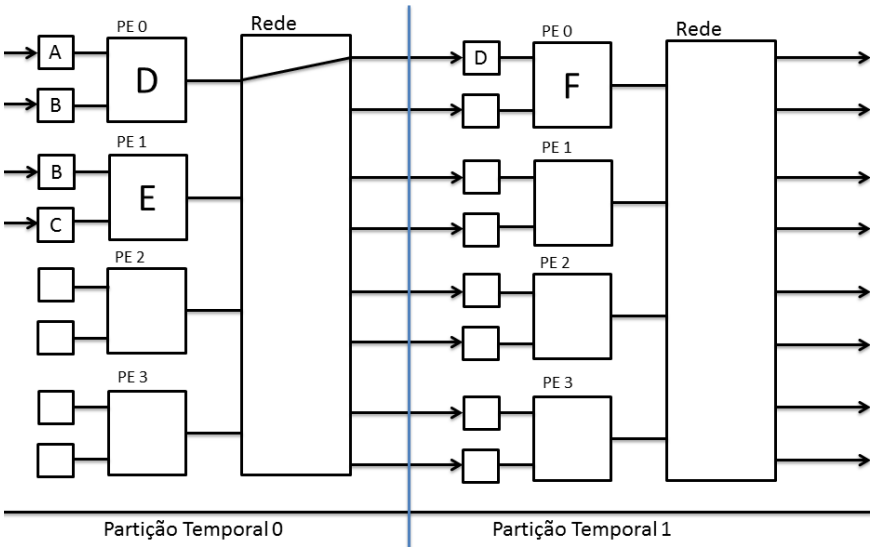


Figura 3.6. Mapeamento das três primeiras operações.

o estado do mapeamento quando é identificada a falta de PEs.

Reiniciando com o $II = 2$, o algoritmo posiciona D e E nas primeiras PEs, na partição temporal 0, assim como foi feito para $II = 1$. Porém, agora a operação F ficará na partição temporal 1, sendo posicionada no PE_0 e, em seguida, é realizado o roteamento da aresta $D \rightarrow F$, como mostrado na Figura 3.6.

Agora, para o posicionamento de G, é identificado novamente o desbalancea-

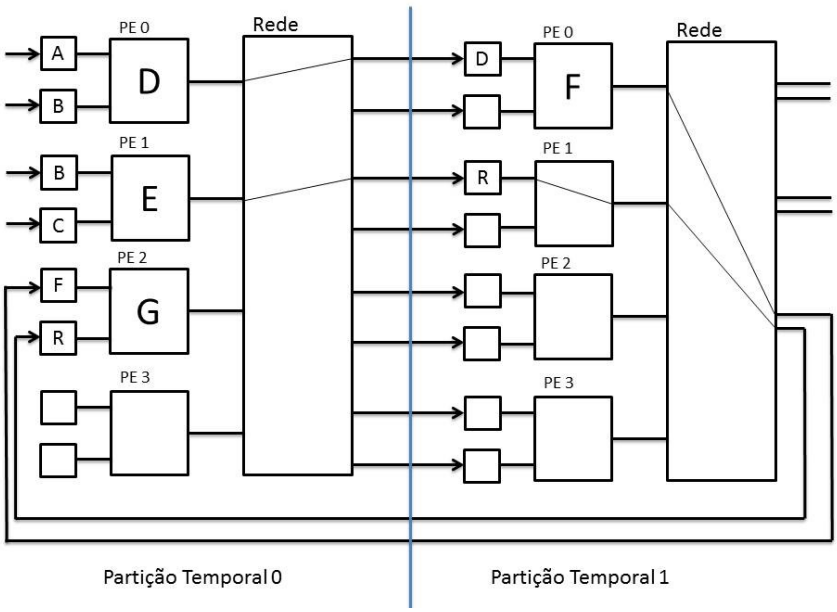


Figura 3.7. Escalonamento final, $II = 2$.

mento entre suas fontes, E e F. Um registro então é alocado no primeiro registro local do PE_1, na partição temporal 1. Em seguida é realizado o roteamento de E para o registro. Para finalizar, G é posicionada no PE_2 da partição temporal 0 e são realizados os roteamentos das arestas $R \rightarrow G$ e $F \rightarrow G$. O algoritmo encontrou resultado para o escalonamento do grafo, com $II = 2$. É interessante notar que o II mínimo ($MinII$) foi alcançado, pois são necessários 4 PEs mais 1 registro para escalonar corretamente o grafo, necessitanto de 5 PEs. A Figura 3.7 mostra o resultado final.

3.3 Compartilhamento de Registros

Como dito anteriormente, quando um nó destino for processado, é necessário verificar o escalonamento de seus nós fonte para saber se haverá a necessidade de usar registros para balancear o caminho de dados. Pode acontecer de um nó ter múltiplos destinos, chamado de *multicast*, e dois ou mais desses destinos necessitarem de registros para balanceamento. A Figura 3.8 mostra um exemplo desses caminhos reconvergentes, muito comuns em algoritmos multimídia. Esses caminhos podem ter diferentes tamanhos.

Nesses casos de *multicasting*, cada aresta é balanceada separadamente usando o primeiro registro (R) do PE, como mostrado na Figura 3.8(a) em $a \rightarrow d$ e $a \rightarrow e$, semelhante ao que foi proposto em abordagens anteriores [Mei et al., 2003; Park

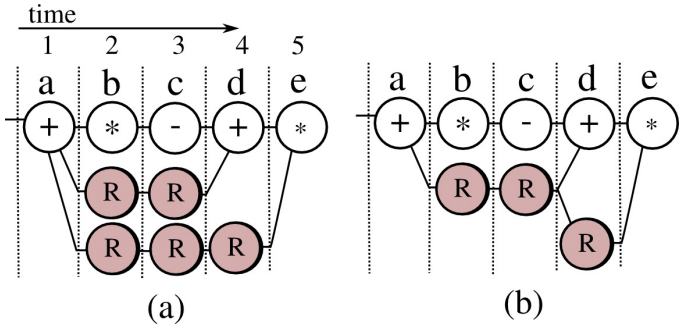


Figura 3.8. (a) Balanceamento normal (b) Compartilhamento de Registros.

et al., 2008; Oh et al., 2009]. Recentemente, um roteamento compartilhado para *multicast* foi apresentado por [Chen & Mitra, 2012] para a arquitetura baseada no ADRES [Mei et al., 2005] a fim de minimizar o número de registros temporários. O presente trabalho usa uma estratégia semelhante, apesar de usar uma arquitetura diferente. A técnica é simples e seu resultado é mostrado na Figura 3.8(b). Quando o nó d é processado, dois registros são inseridos. Um vetor adicional é necessário para manter o valor da última conexão do nó em cada partição temporal, isso é implementado pela função *Insere_Registro* (linha 15 do Algoritmo MSG Base). Quando o nó e está sendo processado, a fonte a é verificada. Uma vez que o nó a tem seu valor armazenado até o tempo 3, sob forma de registro, somente mais um registro é necessário para balancear o caminho de a para e, e não três como seria sem o compartilhamento. O vetor adicional tem sua descrição completa abaixo.

- Vetor **Envia**: armazena em qual PE, da próxima partição temporal, está o registro que irá armazenar o valor de saída do PE atual. Se não houver registro, retorna um valor vazio. Essa estrutura permanecerá geralmente pouco utilizada durante a execução, mas é fundamental para a economia de registros. A complexidade em espaço de memória é $O(U \times C)$.

A Figura 3.9 mostra um exemplo real de balanceamento para o *benchmark ewf*, descrito na seção 5.1. Os nós com números representam as operações do grafo e os nós com a letra R seguida por um número representam os registros necessários para o balanceamento. Pode-se notar que o compartilhamento de registros reduziu em 3 os PEs necessários para balancear o grafo, somente neste trecho. Somente ocorre economia quando há *multicasting* de operações que precisam ser balanceadas. No capítulo de resultados será mostrado com que frequência ocorrem esses *multicastings*, para os *benchmarks* testados.

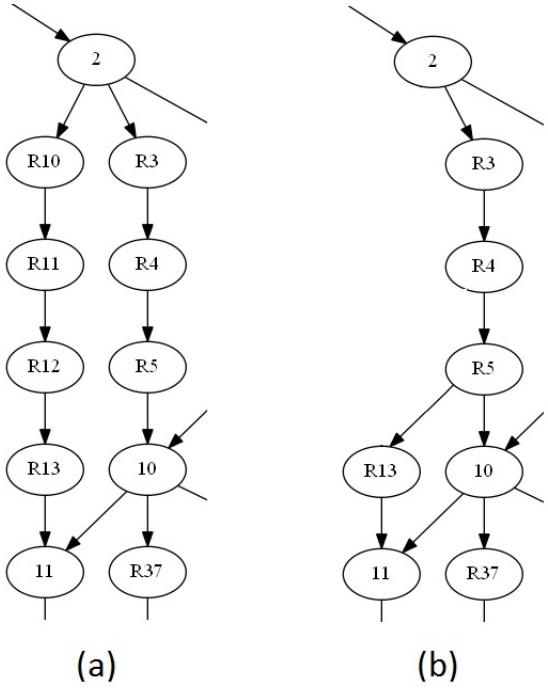


Figura 3.9. Exemplo real: parte extraída do escalonamento do *benchmark ewf* (a) sem compartilhamento de registros (b) com compartilhamento de registros.

3.4 Registros Locais

A arquitetura apresentada neste trabalho é baseada na arquitetura proposta por [Ferreira et al., 2011a], onde a rede de interconexão multiestágio é substituída pela rede *crossbar*. Cada elemento de processamento tem uma unidade funcional (FU) e dois registros: A e B. A saída da FU é conectada às duas redes *crossbar*. Isso permite que as entradas A e B recebam dados de diferentes FUs da partição temporal anterior. Quando um registro de balanceamento é necessário no PE_i, o registro A_i é utilizado. A FU_i e o registro B_i são ignorados. Portanto, o registro B_i e a FU_i ficam ociosos, e somente o registro A_i é usado para implementar um operador de registro, como mostrado na Figura 3.10(a).

Este trabalho propõe outra alternativa para a arquitetura: a possibilidade de usar ambos os registros locais, adicionando multiplexadores à arquitetura, como mostra a Figura 3.10(b). Embora o algoritmo base precise ser alterado, as modificações são simples e não causam grande impacto. Primeiramente, um vetor, chamado REG_Livre, é necessário para guardar o primeiro registro livre, em adição ao vetor Unidade_Livre. Sua descrição é semelhante ao vetor Unidade_Livre, somente mudando que REG_Livre indica qual o próximo registro livre, em cada partição temporal. O vetor REG_Livre é atualizado pela função *Insere_Registro*. Somado

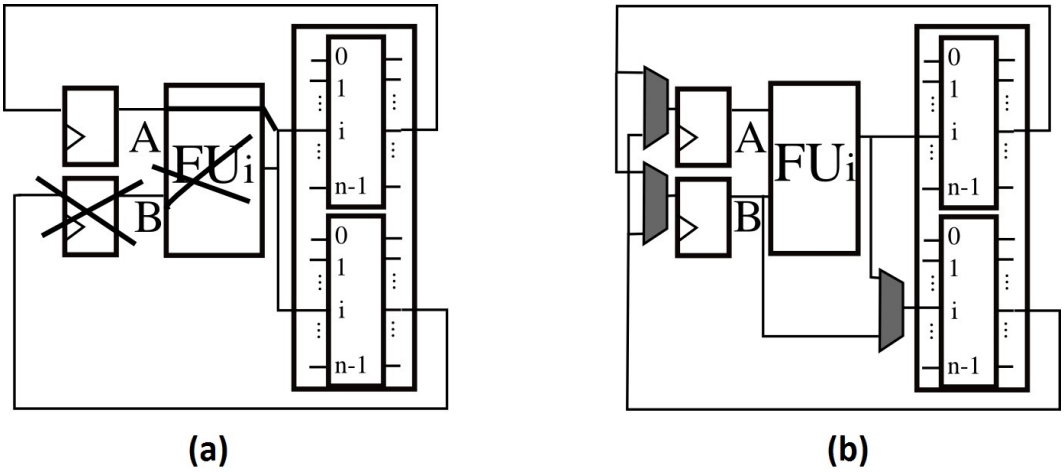


Figura 3.10. (a) Arquitetura base: ignora B, somente A é utilizado (b) Arquitetura para registros locais: pode usar os dois registros.

a isso, somente a linha 19 do Algoritmo MSG Base deve ser alterada se o último registro alocado usa a rede *crossbar* B para rotear. Se isto acontecer, REG_Livre terá que indicar o próximo registro como sendo o registro A da próxima unidade livre. Mais detalhes sobre cada estratégia estão na seção de exemplos.

- Vetor **REG_Livre**: indica qual o próximo registro livre para ser alocado, a cada partição temporal. A complexidade em espaço de memória é $O(U \times C)$, lembrando que U é o número de PEs da arquitetura e C é o número máximo de partições temporais suportado pela arquitetura.

É necessário comentar que a estratégia de registros locais sempre é utilizada em conjunto com a estratégia de compartilhamento de registros, ou seja, as estratégias são incrementais.

3.4.1 Exemplo: Comparação entre estratégias e arquiteturas

Este exemplo tem como objetivo esclarecer as diferenças entre os algoritmos MSG base e o algoritmo MSG que usa ambos os registros locais, assim como evidenciar as diferenças entre as arquiteturas para as duas estratégias. É necessário deixar claro que a estratégia de compartilhamento de registros é usada também na estratégia de registros locais, ou seja, as estratégias são somadas.

A Figura 3.11(a) mostra um grafo de fluxo de dados com 5 operações, de A a E, a ser mapeado. Independente do algoritmo utilizado, é necessário balancear o grafo, de modo que todas as fontes de dados estejam sincronizadas. Isso é feito incluindo os dois registros, $R1$ e $R2$, da Figura 3.11(b).

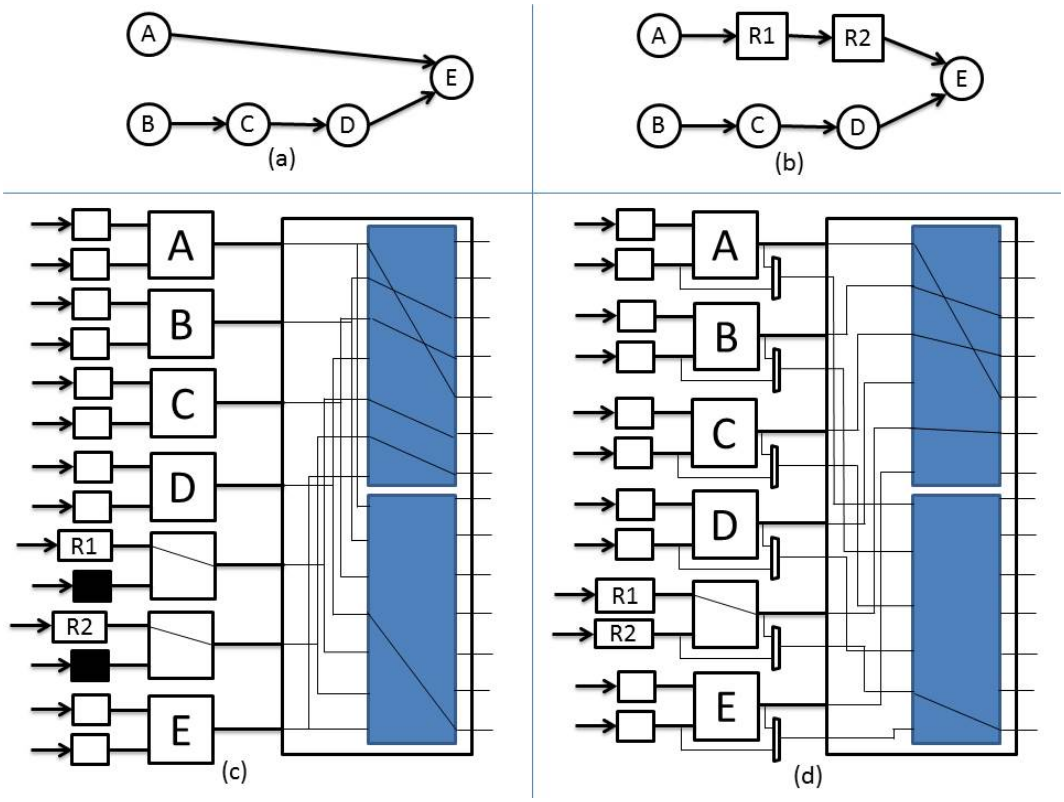


Figura 3.11. (a) Grafo não balanceado (b) Grafo balanceado (c) Mapeamento na arquitetura base (d) Mapeamento na arquitetura para RL: usa os dois registros.

Para o algoritmo e arquitetura base, o resultado do mapeamento é mostrado na Figura 3.11(c), onde dois PEs são usados para alocar dois registros. Nota-se que um registro em cada PE fica ocioso (retângulos preenchidos na figura). No total são utilizados 7 PEs para posicionar todo o grafo balanceado, com $II = 1$. Para o algoritmo e arquitetura com registros locais, é utilizado apenas 1 PE para alocar os dois registros necessários para o balanceamento, mostrado na Figura 3.11(d). Para que essa melhora fosse possível foram necessárias mudanças na arquitetura, possibilitar que dois registros fossem utilizados em um PE, tendo seus valores repassados para a rede, através de uma saída para cada registro, e mudanças na heurística MSG, descritas anteriormente. Foram utilizados 6 PEs nessa estratégia. A melhora da segunda estratégia pode parecer ínfima, mas os resultados mostram grande impacto no resultado do escalonamento, melhorando alguns e possibilitando outros que falham somente com a abordagem base. Esse impacto é devido ao grande número de registros de balanceamento que são necessários nos grafos testados, chegando a alcançar metade dos PEs, para a estratégia base.

3.5 Tolerância a falhas no circuito

Atualmente, sistemas computacionais necessitam tolerar falhas de *hardware*, ou seja, continuar a funcionar mesmo na presença de falhas no circuito. Exemplos disso são sistemas para aeronaves, automóveis e instituições financeiras, aplicações críticas nas quais lidar com falhas é fundamental para evitar consequências fatais.

Os circuitos integrados vêm aumentando sua densidade com o passar do tempo, os componentes do *hardware* ficam menores a cada novo modelo. Isto provê aumento significativo no desempenho dos circuitos, à medida que têm um número maior de dispositivos inseridos numa mesma área. Porém, o atual dimensionamento desta tecnologia gera problemas de confiabilidade, com aumento da densidade de defeitos inseridos durante a fabricação, que geram falhas permanentes no *hardware*. Desse modo, cada circuito fabricado poderá ter falhas em unidades diferentes. Sendo assim, não é possível avaliar falhas de forma genérica, há a necessidade de verificar o estado do circuito a cada novo mapeamento. Em outras palavras, a compilação é gerada em tempo de execução, então o estado da arquitetura CGRA pode ser verificado, em caso de falhas.

A seguir, há um exemplo que mostra o comportamento da heurística MSG em caso de falhas. Basicamente, a heurística deverá testar, à cada tentativa de posicionamento, se a unidade de processamento apresenta falha ou não. Se apresentar, a heurística simplesmente tentar utilizar a próxima unidade disponível. A heurística apenas avalia falhas nas unidades, não considerando a possibilidade de falhas na rede, por exemplo. Avaliar falhas em outras partes do circuito fica como sugestão de trabalho futuro. Seria possível, por exemplo, avaliar falhas na rede de interconexão, considerando que alguns chaveamentos da rede *crossbar* falhassem.

3.5.1 Exemplo: Heurística MSG tolerante a falhas

Neste exemplo está descrito como a heurística MSG pode tolerar falhas nos PEs da arquitetura. Será usado o mesmo grafo de fluxo de dados do exemplo da heurística base, mas agora o CGRA tem uma falha no PE 1 que impossibilita seu uso, como mostrado na Figura 3.12. Relembrando, esse CGRA tem 2 PEs com capacidade de entrada externa de dados, porém um desses PEs não está em condição de ser usado, restando apenas 1 PE com capacidade de entrada externa de dados.

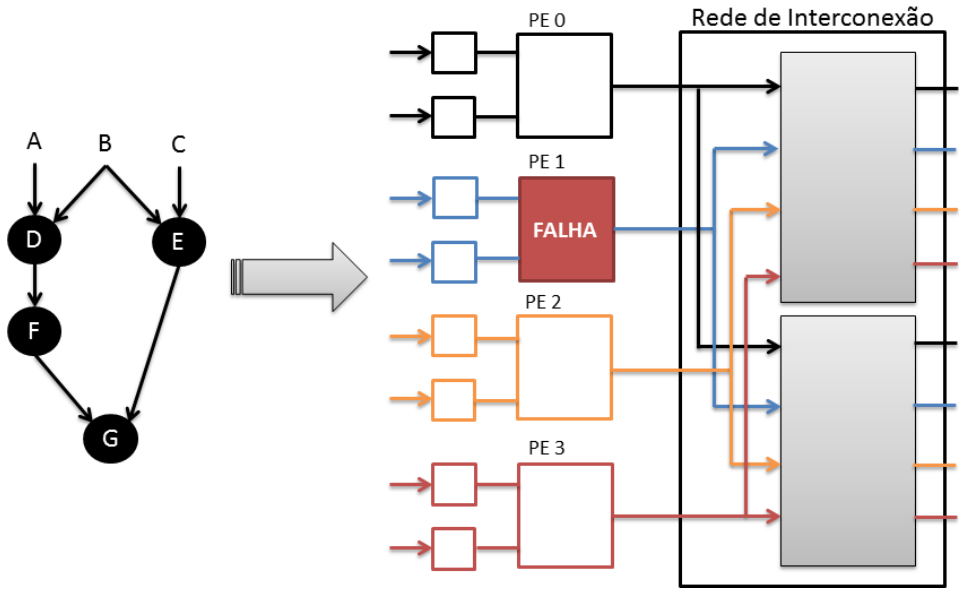


Figura 3.12. Grafo de fluxo da dados a ser mapeado no CGRA com 4 PEs, sendo que o PE 1 apresenta falha.

O mapeamento poderá seguir de forma normal, exceto por uma modificação necessária na heurística MSG. Será necessário verificar, à cada tentativa de posicionamento, se o PE indicado como próximo livre está funcionando normalmente. Supõe-se que há uma lista disponível de PEs que não estão funcionando.

Como mostrado no exemplo 1, este grafo necessita de, pelo menos, duas partições temporais para ser mapeado. Mas neste exemplo, a falha do mapeamento para $II = 1$ ocorrerá no momento em que a heurística tenta posicionar a operação E, que tem entrada externa de dados, e verifica que não há mais PEs com essa capacidade disponíveis, ocasionando o reinicio da heurística, com $II = 2$. Assim, a operação D é posicionada no PE 0 da partição temporal 0 e a operação E é posicionada no PE 0 da partição temporal 1, como mostrado na Figura 3.13.

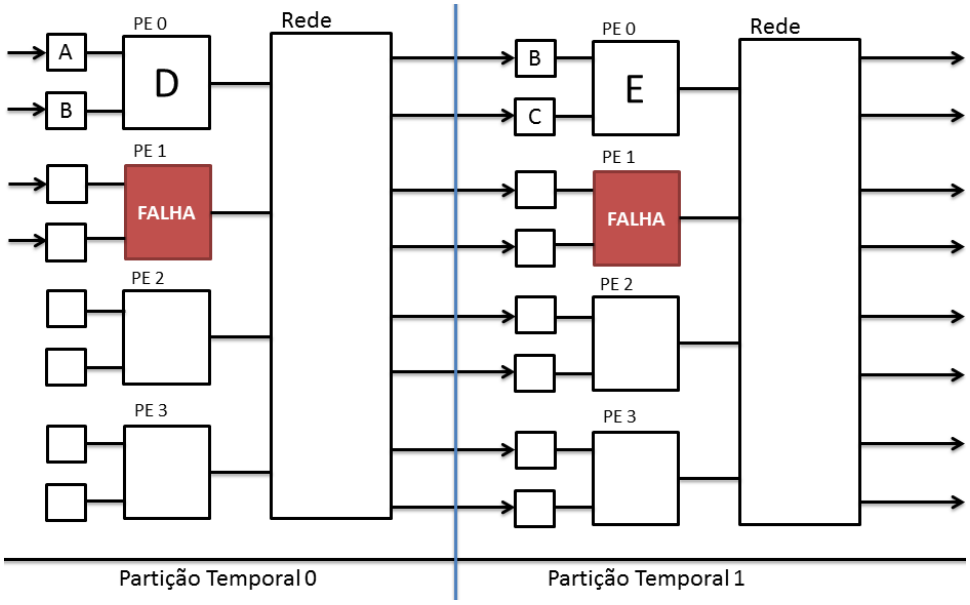


Figura 3.13. Estado do mapeamento em que estão posicionadas as operações com entrada externa de dados.

Em seguida, a heurística percorre o caminho usual, mostrado nos exemplos anteriores. Há apenas mais uma diferença no mapeamento: dessa vez não foi necessário utilizar registros de balanceamento. O motivo é que o desbalanceamento que existia entre as operações que G recebia dados, que são E e F, ficaram no mesmo tempo de escalonamento. A Figura 3.14 apresenta o estado final do mapeamento.

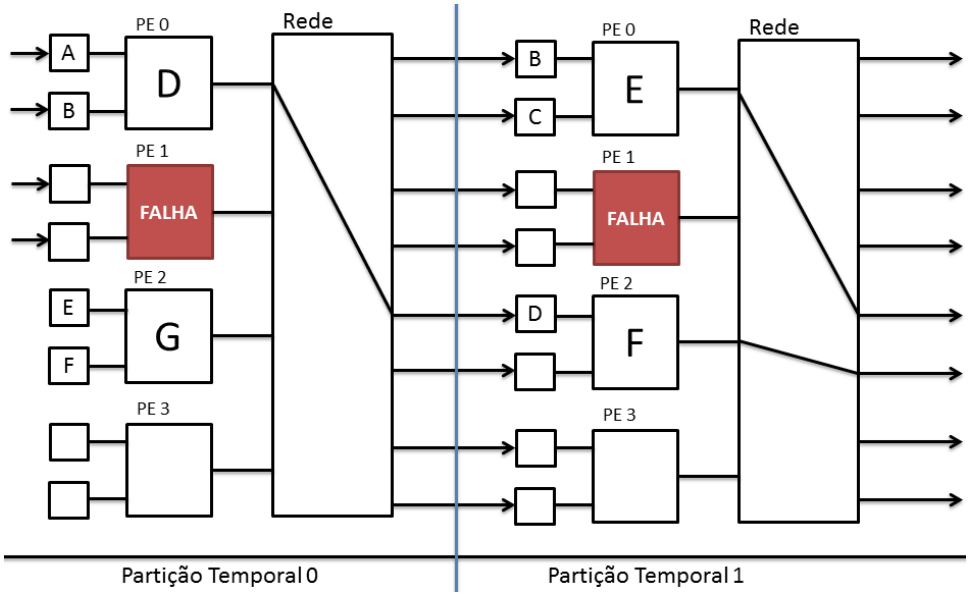


Figura 3.14. Estado final do mapeamento.

3.6 Mapeamento de *multikernels*

Os sistemas computacionais atuais executam múltiplas aplicações ao mesmo tempo, o que exige que o *hardware* seja capaz de lidar com um ambiente dinâmico em que mais de um código está sendo executado, num mesmo instante. Por exemplo, um dispositivo móvel pode executar vídeo e audio em alta definição, ao mesmo tempo em que gera gráficos 3D para outra aplicação. Essa característica deve estender-se também aos aceleradores de aplicações, como CGRAs e FPGAs.

Como mencionado na seção 2.2, a compilação dinâmica torna possível a aceleração de trechos de uma aplicação em tempo de execução. Isso abre possibilidade para o mapeamento de *threads* (ou *kernels*) dinâmicos ou até mesmo o mapeamento de mais de um *thread* ao mesmo tempo, que é a tendência dos processadores atuais.

Porém, como cita [Shrivastava et al., 2011], a maioria dos compiladores para CGRA ao escalonados estaticamente e mapeiam um *kernel* usando toda a capacidade do CGRA. Isso elimina qualquer possibilidade de mapear mais de um *kernel* ao mesmo tempo. Em [Shrivastava et al., 2011], os autores argumentam que um requisito fundamental para o uso de *multikernels* é restringir um *kernel* a usar apenas uma porção do CGRA para seu mapeamento. Entretanto, a ideia é que normalmente o compilador use o CGRA inteiro para mapear um *kernel*. Somente quando precisar de mapear outro *kernel* é que será necessária uma espécie de compressão do escalonamento original para abrir espaço para o segundo *kernel*.

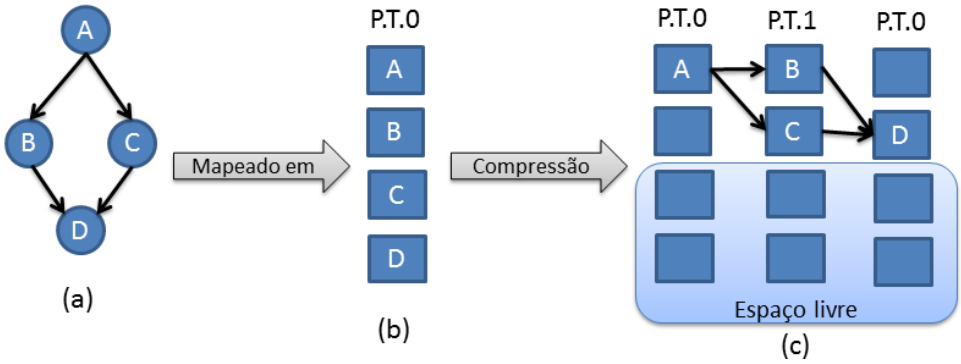


Figura 3.15. Exemplo de compressão de mapeamento (a) Grafo a ser mapeado (b) mapeamento normal (b) mapeamento comprimido.

A Figura 3.15 mostra um exemplo do mecanismo de compressão de mapeamento num CGRA com 4 PEs e rede *crossbar*. O grafo é inicialmente mapeado sem restrição de espaço (Figura 3.15(b)), com apenas uma partição temporal, usando todas os PEs disponíveis na arquitetura. Nesse estado, não havia PE disponível

para o posicionamento de nenhuma operação adicional, assim, se quisermos fazê-lo, será necessária uma compressão ou um novo mapeamento de todas as operações. A Figura 3.15(c) mostra o exemplo para o grafo comprimido, ao invés de usar uma partição temporal, adicionou-se mais uma e foi feito um espalhamento das operações, isso liberou 2 PEs em cada partição temporal, abrindo espaço para o mapeamento de outro grafo.

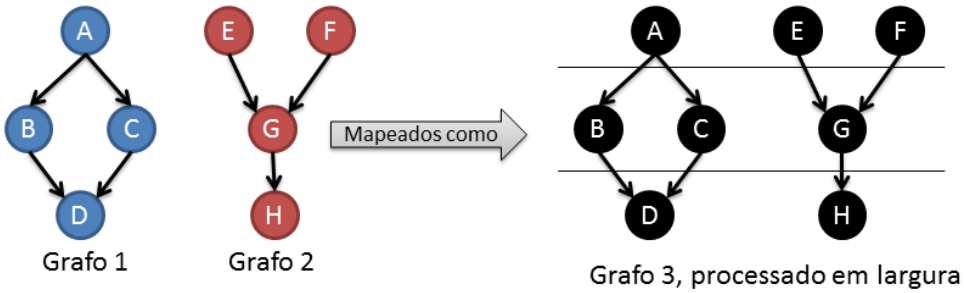


Figura 3.16. Dois grafos a serem mapeados ao mesmo tempo (*multikernel*) são tratados como apenas um.

Nesta dissertação não foi utilizada a abordagem mencionada no parágrafo anterior, de compressão de um escalonamento. A ideia é mapear diretamente dois trechos de códigos diferentes ao mesmo tempo no CGRA, visando a execução concorrente de múltiplos *kernels*. Foi usada a abordagem de refazer completamente o mapeamento, visto que podemos fazer isso de forma eficiente. Supõe-se que, em algum momento, os trechos a serem mapeados estarão disponíveis ao mesmo tempo. Dessa forma, o mapeamento de mais um *kernel* será tratado como o mapeamento de um terceiro *kernel*, que representa a união dos outros. A Figura 3.16 exemplifica isso. A partir desse ponto, a heurística MSG funcionaria da maneira usual, mapeando o grafo resultante da união dos *kernels*.

4. Modelo de Implementação em *Hardware*

Este capítulo apresenta um modelo em *hardware* para a heurística MSG, proposta no capítulo 3. O objetivo de uma implementação em *hardware* é alcançar um tempo de compilação ainda menor que o algoritmo executado por um compilador *Just-In-Time*. O modelo para esta implementação é composto de uma máquina de estados que atua sob os vetores descritos no capítulo anterior. O algoritmo em *hardware* também pode ser implementado em um FPGA, como mostram [Filho & Ferreira, 2012] e [Mucida et al., 2012]. Neste capítulo será mostrado apenas a máquina de estados para a implementação em *hardware* do algoritmo base. As técnicas adicionais de registros locais e compartilhamento de registros não estão descritas, mas podem ser incluídos.

A Figura 4.1 mostra as diferenças gerais entre a implementação em *hardware* e a implementação via *software* no processador *softcore*. Primeiramente, o sistema reconfigurável engloba três diferentes partes: a arquitetura reconfigurável (CGRA), o algoritmo de escalonamento, posicionamento e roteamento, no caso é a heurística MSG, e a memória de configuração. O sistema inteiro é programado sob um FPGA comercial. A diferença está em como o algoritmo é implementado. No caso da implementação via *softcore* (Figura 4.1(c)), o código C do algoritmo é compilado para executar no processador *softcore*, que é montado sob o FPGA. O processador executa cada instrução do algoritmo sequencialmente. Para a implementação direta no *hardware* (à esquerda), não há processador executando instruções de código C, há estruturas de memória e uma máquina de estados implementando o algoritmo diretamente no *hardware* do FPGA. Esse *hardware* especializado executa várias instruções paralelamente, em cada estado da máquina há um conjunto de instruções específicas para realizar uma tarefa. A figura também mostra que um grafo é entrada para o sistema reconfigurável e, quando estiver executando o grafo mapeado, há fluxo de dados de entrada no CGRA.

4.1 Estrutura da Memória e MEF

O algoritmo é implementado como uma máquina de estados finitos (MEF) e usa apenas alguns registros para guardar valores locais e 7 memórias distribuídas: cada memória refere-se à estrutura de mesmo nome, descritas no capítulo 3. A Figura

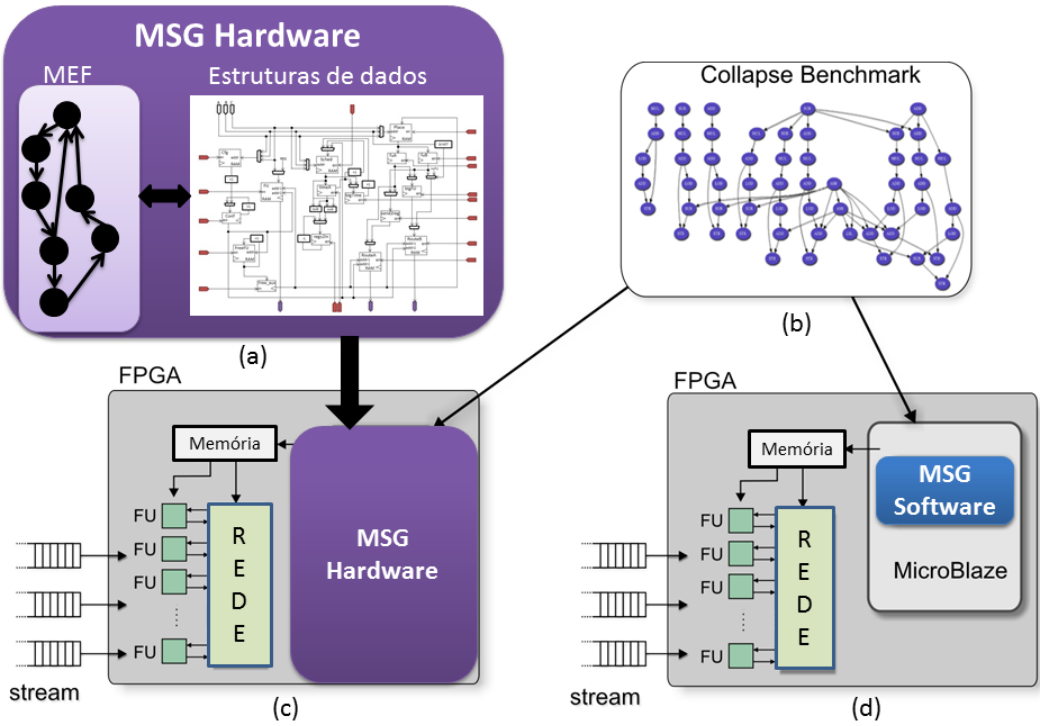


Figura 4.1. Sistema Reconfigurável: (a) estrutura interna da heurística MSG em *hardware* (b) *benchmark* de entrada (c) modelo de implementação em *hardware* (d) o modelo de implementação em *software* usando *softcore* para executar o código C.

4.2 mostra as estruturas utizadas, que realiza as operações de cada estado da MEF em paralelo. A MEF, mostrada na Figura 4.3, controla o acesso à memórias da Figura 4.2. As setas de entrada ligadas em cada estrutura (ligadas no *en* de cada memória), na Figura 4.2, representam o controle da MEF sob as estruturas. A cada estado, um deterninado conjunto de estruturas estarão habilitadas para a gravação, indicado pelo *enable* - *en* das memórias, dependendo de quais operações são feitas no estado. As setas de saída (indicando saída das memórias *FU, RouteA e RouteB*) representam a saída do algoritmo ou a entrada para a memória de configuração do CGRA, que será utilizada ao decorrer da execução no CGRA. A entrada do grafo de fluxo de dados é indicada pelas setas A, B e C, onde A e B representam as fontes, s_1 e s_2 do algoritmo base, seção 3.2, e C representa o destino, nó t do algoritmo base, ou seja, C é a operação que será mapeada.

A cada estado o algoritmo executa ações específicas que respeitam as restrições do *hardware*, listadas abaixo:

- Leitura ou escrita numa posicão de memória, a cada ciclo de relógio. É possível fazer apenas uma das duas operações. Extensível também à indexação de

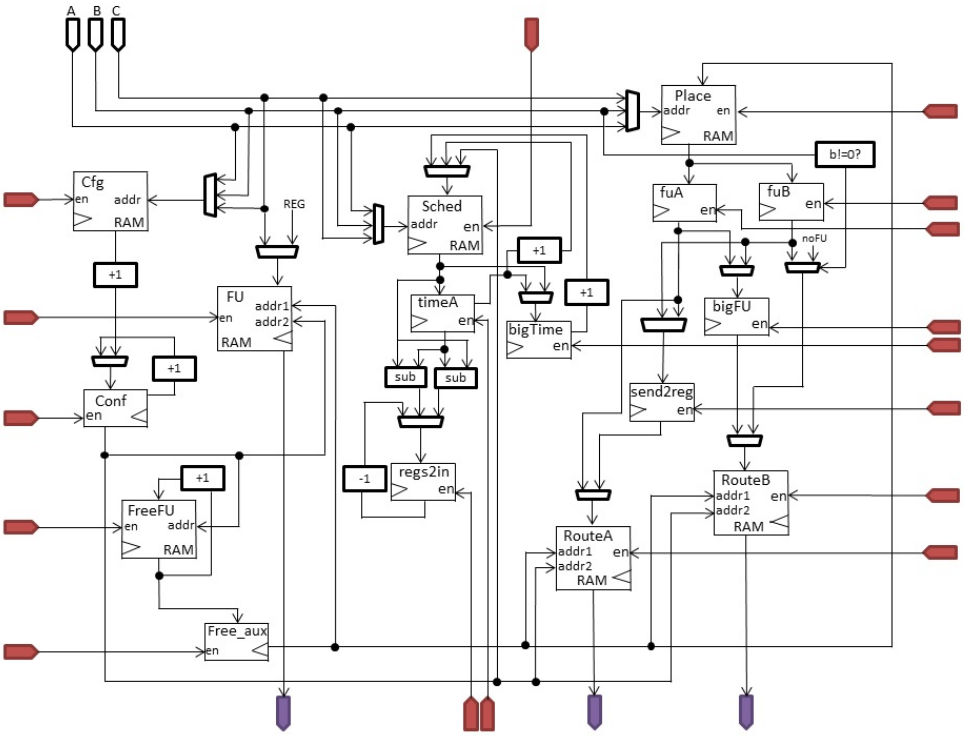


Figura 4.2. Memórias de Escalonamento, Posicionamento e Roteamento para a heurística MSG base.

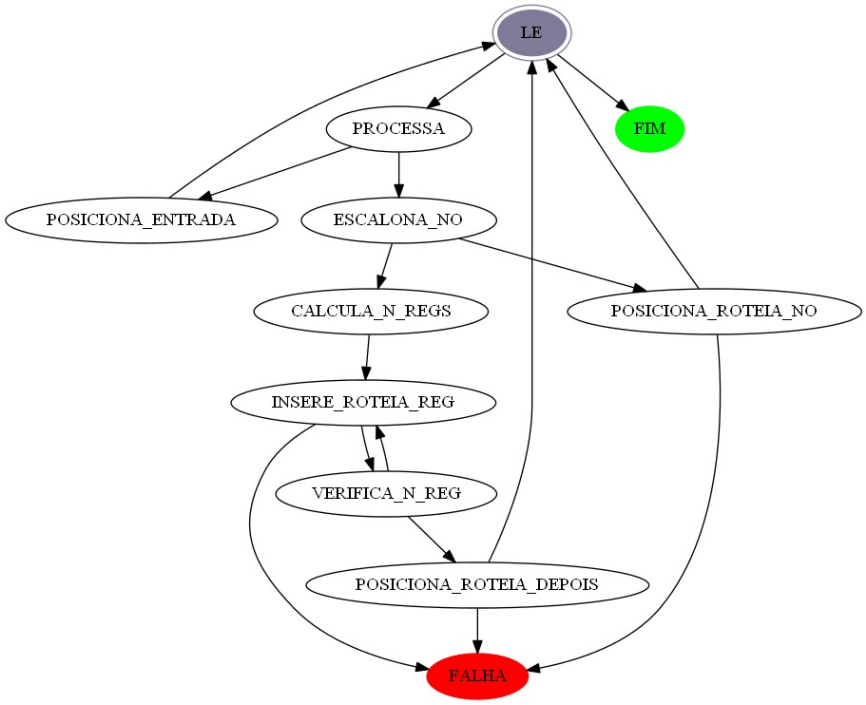


Figura 4.3. Máquina de estados finitos.

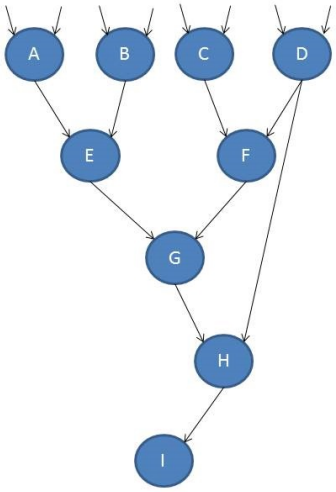


Figura 4.4. Exemplo de grafo de fluxo de dados.

vetores.

- Não há uso de comandos de laço, estes são criados pela máquina de estados.
- Testes condicionais são feitos apenas no fim de um estado, para decidir qual será o próximo estado. A única exceção é atribuição condicional, na qual é mais simples criar uma estrutura para o *hardware*.

Cada estado completa sua execução em 1 ciclo do processador, assim pode-se contar exatamente quantos ciclos cada grafo gasta para ser completamente mapeado e quanto tempo irá demorar para realizar a tarefa.

4.2 Execução da MEF

Para exemplificar a execução do algoritmo, consideremos o grafo da Figura 4.4, na qual as operações estão representadas por letras, de A a I, e uma arquitetura semelhante à das Figuras 3.5, 3.7 e 3.10(a), com 8 PEs. Como o grafo a ser mapeado tem 9 operações, o intervalo de inicialização inicial é 2, assim o algoritmo inicia-se com duas partições temporais.

A máquina de estados começa sempre no estado LE, onde é efetuada a leitura do primeiro nó de entrada, neste caso é a primeira operação a ser mapeada, que é A. Passa-se então para o estado PROCESSA. Este estado identifica que A tem entradas externas, passando ao estado POSICIONA_ENTRADA, que efetua o posicionamento da operação com entradas externas na arquitetura. Um nó

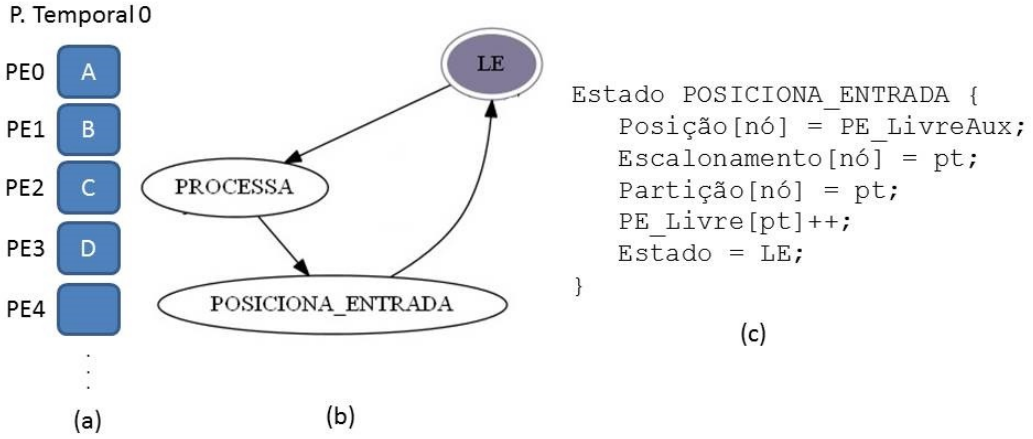
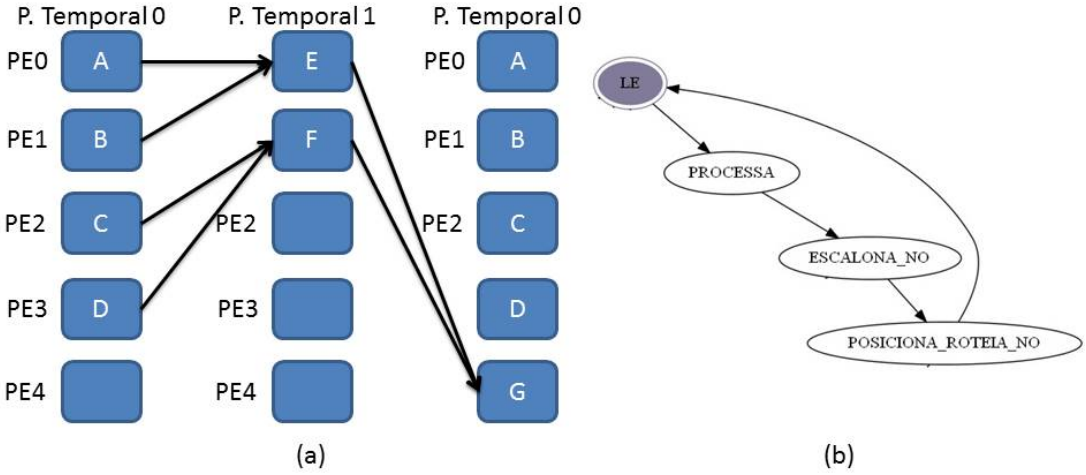


Figura 4.5. (a) Mapeamento parcial (b) Estados percorridos para os nós A, B, C e D (c) Código do estado POSICIONA_ENTRADA: todas as operações do código são executadas paralelamente.

de entrada não necessita de roteamento nenhum, visto que apenas recebe valores externos à rede de interconexão e executa sua tarefa. Por isso, após o estado POSICIONA_ENTRADA, a MEF volta ao estado LE. Como os nós B, C e D também são de entrada, o mesmo caminho é feito para esses nós na máquina de estados, obtendo-se o posicionamento parcial do grafo mostrado na Figura 4.5. A arquitetura tem limite para operações de entrada externa de dados, portanto, é verificado se há PE para mais uma operação de entrada na partição temporal. Essa verificação é feita num estado auxiliar intermediáro, não mostrado na Figura 4.3.

Após o posicionamento das entradas, passa-se à próxima operação que vem na sequência da varredura do grafo em largura, que é a operação E. As fontes da operação E, são $A \rightarrow E$ e $B \rightarrow E$. Depois de realizada a leitura, passa-se ao estado PROCESSA, que verifica se E é uma operação com entrada externa de dados ou não. Essa operação consiste em verificar se o nó tem fonte já posicionada ou não, isso é feito no estado PROCESSA. O teste identifica que E é um nó interno, portanto, não é entrada. Assim, é necessário identificar quais são os tempos das fontes de E (A e B), isso é feito no estado ESCALONA_NO. Este estado testa se os tempos das fontes do nó atual estão no mesmo tempo ou não. Se sim, passa-se ao estado POSICIONA_ROTEIA_NO, senão passa-se ao estado CALCULA_N_REGS. Neste caso, os tempos de A e B são iguais, passando o controle ao estado POSICIONA_NO. Neste estado, E é posicionado no PE_0 da partição temporal 1. Em seguida é preciso rotear os PEs 0 e 1 da partição temporal 0, nas quais estão posicionadas as fontes de E, para o PE_0 na partição temporal 1, isso é feito no mesmo estado, como mostrado na Figura 4.6. Após isso, retorna-se ao




```
Estado POSICIONA_ROTEIA_NO {
   Posição[nó] = PE_LivreAux;
   Escalonamento[nó] = tempoS1+1;
   Partição[nó] = pt;
   RoteamA[PE_LivreAux][pt] = PE_S1;
   RoteamB[PE_LivreAux][pt] = PE_S2;
   PE_Livre[pt]++; Estado = LE;
}
```

(c)

Figura 4.6. (a) Mapeamento parcial. (b) Estados percorridos para os nós E e F.

estado LE, para a próxima entrada. O mapeamento das operações F e G seguem o mesmo caminho, porém G é posicionada na partição temporal 0. A Figura 4.6 mostra a configuração do mapeamento atual.

Na sequência do grafo de entrada, vem a operação H. Segue-se o caminho igual aos anteriores, até chegar no estado ESCALONA_NO. Neste estado o teste de tempo das fontes indica que estas estão em tempos diferentes, apesar de estarem na mesma partição temporal (D e G). A operação D é um nó de entrada, pode-se considerar seu tempo como zero. Já o nó G foi o último a ser posicionado, tendo tempo 2. Logo é necessário sincronizar as entradas do nó H, pois estão em tempos distintos. Para isso, usam-se os PEs como simples registros que armazenam e repassam os valores das saídas dos nós de origem. Passa-se ao estado CALCULA_N_REGS para decidir quantos registros serão necessários. O número de registros é definido pela diferença entre os tempos das fontes, que no caso é 2. Então passa-se ao estado INSERE_ROTEIA_REG, para inserir o primeiro registro e efetuar o roteamento do PE para o registro alocado. Passa-se ao estado VERIFICA_N_REGS, que simplesmente verifica se ainda há registros para inserir.

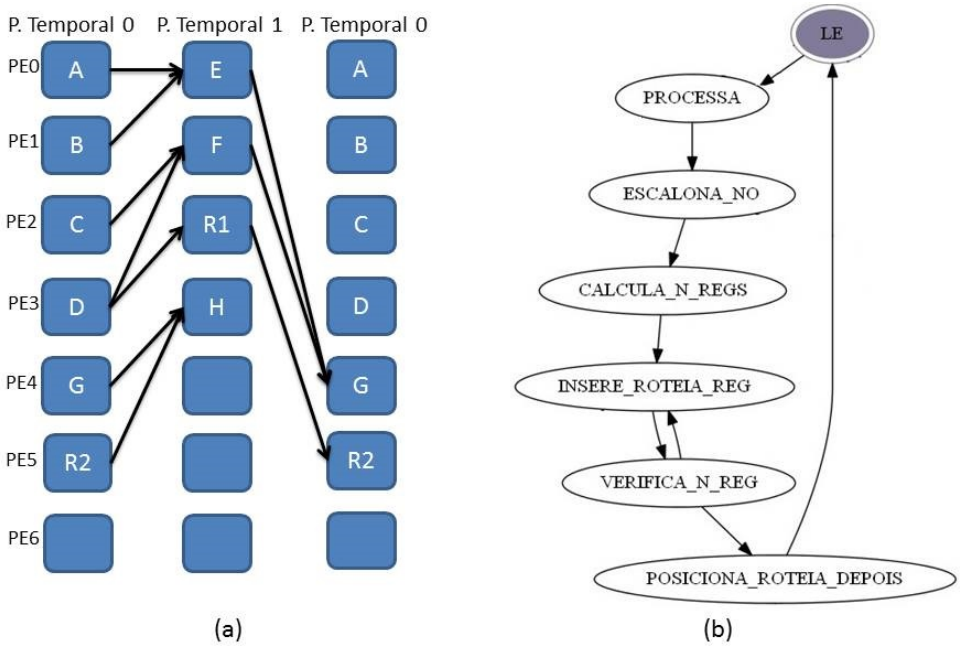


Figura 4.7. (a) Mapeamento final. (b) Estados percorridos para o nó H.

Como há um registro ainda, volta-se ao INSERE_ROTEIA_REG e, em seguida, é necessário verificar se há mais registros para inserir, em VERIFICA_N_REGS. Assim os dois registros estão posicionados e o valor do nó D está guardado no último registro com o tempo exato. Como não há mais registros a inserir, o controle passa para o próximo estado, o POSICIONA_ROTEIA_DEPOIS. Este é o estado que irá posicionar o nó H e que realiza o roteamento do último registro inserido para H. Em seguida, esse mesmo estado também efetua o roteamento de G para H. Após o roteamento $G \rightarrow H$, termina o mapeamento de H e o controle passa novamente para LE. O estado final do mapeamento e o caminho percorrido na MEF para o mapeamento da operação H são mostrados na Figura 4.7.

Nas figuras do exemplo não foi mostrado o estado FALHA da Figura 4.3. Este estado pode ser executado quando o algoritmo não conseguiu posicionar alguma operação ou registro, por falta de PE disponível na partição temporal solicitada. Além disso, alguns estados necessários para a implementação do algoritmo base não foram mostrados na máquina de estados, com o objetivo de simplificar a explicação. Foram omitidos da máquina, por exemplo, estados que verificam se há um PE disponível para ser alocado por uma operação ou registro, que são necessários antes de cada posicionamento.

```
Estado INSERE_ROTEIA_REG {
   n_regs_p_inserir--;
   RoteamA[PE_LivreAux][pt] =
      PE_Envia_P_Reg_Atual;
   PE_Livre[pt]++;
   Estado = VERIFICA_N_REG;
}
```

```
Estado VERIFICA_N_REG {
   IF(n_regs_p_inserir > 0)
     Estado = INSERE_ROTEIA_REG;
   ELSE
     Estado =
       POSICIONA_ROTEIA_DEPOIS;
}
```

Figura 4.8. Descrição dos estados INSERE_ROTEIA_REG e VERIFICA_N_REGS. Todas as operações de cada estado são executadas em paralelo.

A Figura 4.8 mostra um exemplo de código contido na implementação da máquina de estados. As variáveis usadas no estado serão acessadas na memória do algoritmo em *hardware*. Ao fim de cada estado há uma especificação de qual será o próximo estado.

5. Resultados

Neste capítulo será feita a avaliação da solução proposta, utilizando *benchmarks* de aplicações multimídia. Os resultados serão apresentados na seguinte ordem: descrição dos *benchmarks* usados para teste, resultados comparativos do algoritmo *modulo scheduling*, com e sem compartilhamento de registros e registros locais, intervalo de iniciação, tempo de compilação ou tempo de execução do algoritmo, avaliação do impacto de falhas nos PEs do CGRA, mapeamento de *multikernels* e avaliação inicial para o algoritmo implementado em *hardware*.

5.1 *Benchmarks* Utilizados

Para avaliar a eficácia do algoritmo proposto, um conjunto de *benchmarks* de fluxo de dados foram extraídos de [Group, 2013], que é um repositório selecionado entre mais de 1400 grafos de fluxo de dados obtido de aplicações do *MediaBench Benchmark Suite*. Estes algoritmos exploram o paralelismo de operações entre matrizes, processamento de sinais digitais e compressões de imagens. Sobre a aplicação de cada *benchmark*: *hal* é uma solução iterativa para uma equação diferencial de segunda ordem, *horner* é um algoritmo de Horner aplicado à curvas de Bezier, *arf* é um filtro de auto regressão, *cosine1* e *cosine2* são duas implementações de FDCT (*Fast Discrete Cosine Transformation*) de uma dimensão, *ewf* é um filtro de ondas elípticas, *fir* , *fir2*, *Fir16* e *TFir64* são versões do filtro de resposta ao impulso finito (*Finite Impulse Response Filter*), *h2v2* é o núcleo do algoritmo de compressão jpeg, *motion* é um bloco básico que contém dois cálculos similares para *mpeg*, *feedback* é uma função que preenche um *buffer* de vértices e calcula as coordenadas de textura, *collapse* é um QMF (*Quadrature Mirror Filterbank*), *write* é a função *write bmp header*, *idctcol* é o inverso da transformação discreta do cosseno, *interpolate* faz uma interpolação linear entre dois pontos, *matmul* é uma multiplicação de matrizes, *jpegslow* e *jpegfast* são um codificador e um decodificador para *jpeg*. Outro grupo de *benchmarks*, utilizados em [Ferreira et al., 2007], foi usado: *Cplx8* é a implementação do fir com aritmética de números complexos, *FilRGB* é um filtro para uma imagem destacando ou escurecendo pixels, *DCT* é uma transformação discreta do cosseno.

Benchmark	Operações	Entradas
hal	11	5
horner	18	5
arf	28	8
motion	32	14
ewf	34	2
fir2	40	16
fir	44	22
Cplx8	46	1
Fir16	49	1
h2v2	51	16
feedback	53	21
FilRGB	57	2
collapse	59	6
cosine1	66	16
cosine2	81	31
DCT	92	1
write	106	38
interpol	108	48
matmul	108	24
jpegfast	167	24
jpegslow	173	24
idctcol	186	18
TFir64	193	1
smooth	196	64

Tabela 5.1. Conjunto de *benchmarks* selecionados.

Os *benchmarks* utilizados foram pré-processados, para que tivessem suas operações organizadas por uma busca em largura no grafo de fluxo de dados. O número de operações de cada *benchmark* está listado na Tabela 5.1, assim como a relação das operações de possuem entrada externa de dados. O tamanho dos *benchmarks* varia de 28 a 196 operações, sem considerar registros de balanceamento, ou 35 a 347 nós, considerando grafos balanceados. A definição do tamanho dos grafos foi baseada no fato que a maioria dos laços de aplicações multimídia têm menos de 100 operações, como demonstrado por [Park et al., 2008], para 214 laços extraídos dos códigos dos algoritmos H.264, ACC, 3D e MP3, aproximadamente 90% têm menos de 100 operações, como mostrado na motivação desta dissertação.

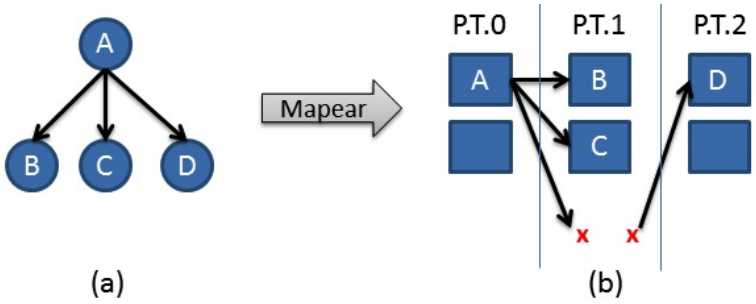


Figura 5.1. (a) Grafo de Fluxo de dados (b) Impossibilidade de roteamento: impossível rotear $A \rightarrow D$.

5.2 Economia de Registros

Nesta seção será feita uma avaliação da estratégia de compartilhamento de registros e de registros locais, descrita na seções 3.3 e 3.4, que visam reduzir o número de registros utilizados e utilizá-los de forma mais eficiente, respectivamente. O experimento consiste em verificar quantos PEs são utilizados como registros de balanceamento, para cada estratégia. É necessário destacar que a arquitetura para a técnica de registros locais é diferente da arquitetura básica, como mostrado na Figura 3.10, Seção 3.4. A arquitetura para o algoritmo MSG base só permite o uso de 1 registro por PE, enquanto que a arquitetura para registros locais permite o uso de ambos. Além disso, a estratégia de registros locais sempre é usada em conjunto com o compartilhamento de registros. Portanto, há três casos diferentes para o algoritmo MSG: base, com compartilhamento registros e com compartilhamento de registros e registros locais.

A Tabela 5.2 mostra o resultado para o total de PEs usados como registros para balancear os grafos, utilizando os 15 menores *benchmarks* descritos na Seção 5.1, estes variam de 28 a 106 operações. A arquitetura alvo consiste em um CGRA de 16 PEs interligados por uma rede *crossbar*, sendo que 4 PEs podem receber dados de entrada externos, em cada partição temporal. Suposições similares são feitas em EMS [Park et al., 2008] e EPIMap [Hamzeh et al., 2012].

As colunas *Nome* e *Operações* mostram o nome e o número de operações de cada *benchmark*, respectivamente. As outras colunas são rotuladas de acordo com a estratégia de registro usada: base, registros compartilhados e registros locais. Considerando que a estratégia base usa um PE inteiro como um registro para balancear e não compartilha registros em multicasts, podem ocorrer casos em que não haverá PEs suficientes para todos os registros, em partições temporais específicas. Nestes casos, aumentar o intervalo de iniciação não resolve, pois a heurística sempre utili-

Nome	Operações	Base	R.Compartilhados	R.Locais
arf	28	13	13	7
motion	32	17	17	10
ewf	34	51	34	18
fir2	40	9	9	5
fir	44	6	6	4
Cplx8	46	34	31	16
Fir16	49	28	28	15
h2v2	51	52	51	27
feedback	53	24	23	13
FilRGB	57	33	31	16
collapse	59	-	-	20
cosine1	66	16	8	5
cosine2	81	-	-	25
DCT	92	27	17	12
write	106	-	-	20

Tabela 5.2. Número de PEs usados como registros para três estratégias: base, registros compartilhados e registros locais.

zará todos os PEs de uma partição temporal, impossibilitando o roteamento, como mostra o exemplo da Figura 5.1. Nesta figura, não é possível repassar o resultado da operação A para a entrada de D, pois não há PEs disponíveis na partição temporal 1, o que é necessário para o roteamento. Por esta razão, o escalonamento falha para 3 dos 15 *benchmarks*, indicados por traços na Tabela 5.2.

A estratégia de compartilhamento de registros reduz o número de registros em 7 dos 15 casos. Entretanto, essa estratégia não resolve os casos dos 3 *benchmarks* que falham no algoritmo base. A redução no número de registros para a estrategia de compartilhamento é mais expressiva em: ewf (33%), cosine1 (50%) e DCT (37%), como mostra a Tabela 5.2. Nesses casos, há um número considerável de arestas de *multicast*. A estratégia de registros locais é mais eficaz, pois os dois registros de um PE podem ser utilizados. No melhor caso, essa estratégia reduz em 50% o número de PEs gastos como registros de balanceamento e, em média economiza 42% de PEs utilizadas, para 12 *benchmarks*. Adicionalmente, os 3 *benchmarks* que não podiam ser mapeados na arquitetura pelas outras estratégias, são mapeados com sucesso pela abordagem de registros locais.

Nome	In	Op	MinII	II Alcançado BA	RC	RL	Op+R
arf	8	28	3	**3**	**3**	**3**	35
motion	14	32	4	**4**	**4**	**4**	42
ewf	2	34	4	6	5	**4**	52
fir2	16	40	4	**4**	**4**	**4**	45
fir	22	44	6	**6**	**6**	**6**	48
Cplx8	1	46	4	6	5	**4**	62
Fir16	1	49	4	5	5	5	64
h2v2	16	51	5	10	10	6	78
feedback	21	53	6	**6**	**6**	**6**	66
FilRGB	2	57	5	6	6	**5**	73
collapse	6	59	5	-	-	6	79
cosine1	16	66	5	7	7	7	71
cosine2	31	81	8	-	-	**8**	106
DCT	1	92	7	11	11	11	104
write	38	106	10	-	-	**10**	126

Tabela 5.3. Intervalo de Iniciação para três estratégias: base (BA), registros compartilhados (RC), registros locais (RL). Em negrito os resultados ótimos.

5.3 Intervalo de Iniciação

Esta seção tem como objetivo mostrar a evolução da qualidade do escalonamento baseada no intervalo de iniciação, para as três diferentes estratégias. Além disso, foi realizada uma comparação entre o algoritmo MSG com a estratégia de registros locais e um compilador chamado REGIMap [Hamzeh et al., 2013], que é uma versão nova proposta pelos autores de EPIMap [Hamzeh et al., 2012].

A Tabela 5.3 mostra o intervalo de iniciação (II) obtido pelas três estratégias de registros. As colunas *In*, *OP* e *MinII* mostram o número de entradas externas, número de operações do grafo e o II mínimo, respectivamente. O MinII é computado pela seguinte função: $MAX((OP + R)/NFU, In/MaxIn)$, onde $OP + R$ (última coluna da Tabela 5.3) é o número de PEs somado ao número de PEs utilizados como registros, somente para a estratégia de registros locais, In é o número de entradas externas de cada grafo e $MaxIn$ é o máximo de entradas externas permitidas pela arquitetura, a cada partição temporal. Para prover comparação com trabalhos recentes [Park et al., 2008; Chen & Mitra, 2012; Hamzeh et al., 2012], este trabalho faz a mesma suposição quanto ao número de entradas externas, $MaxIn = 4$. No entanto, o algoritmo MSG não trata saída de dados, deixando a possibilidade para trabalhos futuros.

A qualidade dos resultados das três diferentes estratégias foi medida comparando o mínimo II com o II alcançado. As estratégias base (BA) e compartilhamento de registros (RC) falham para 3 *benchmarks* com mais de 79 PEs utilizados ($OP + R$), mas alcançam resultados ótimos para 5 dos 12 casos em que consegue escalonar. O motivo de as estratégias anteriores não conseguirem mapear três grafos está relacionado ao enchimento de alguma partição temporal, não sobrando espaço para posicionar mais uma operação. Como a heurística MSG é gulosa, ela não tenta fazer o nenhum tipo de movimentação de operação para tentar posicionar a operação que falta, apenas reinicia com novo II. Quando um determinado II é alcançado, o mapeamento é abortado com resultado de falha. A estratégia de registros compartilhados é melhor que a base em apenas 2 casos, devido ao baixo número de *multicasting* dos *benchmarks* utilizados.

A estratégia de registros locais (RL) alcança melhor resultado, obtendo o II mínimo em 10 dos 15 *benchmarks* (mostrados em negrito na tabela) e mapeando todos. Esse melhor resultado é consequência da melhor utilização dos registros locais da arquitetura.

O experimento a seguir é uma comparação. REGIMap [Hamzeh et al., 2013] compila para uma variedade de arquitetura diferentes, baseadas no modelo ADRES, semelhante à arquitetura da Figura 5.2. REGIMap, diferente da maioria dos trabalhos anteriores, incluindo MSG, não bloqueia o PE quando utiliza registros locais, liberando-o para efetuar computação. Essa otimização resulta em ganho no desempenho, ou seja, menor intervalo de iniciação para o laço mapeado no CGRA.

As características da arquitetura, como número de PEs e tamanho do banco de registros locais, são passadas como parâmetro ao compilador REGIMap. Para efetuar a comparação foi utilizada a arquitetura CGRA 4x4 (16 PEs), com um banco com 8 registradores. Para a heurística MSG foi usada a arquitetura de 16 PEs, com rede *crossbar*, descrita na Seção 3.4. O sistema usado para executar o experimento tem a seguinte configuração: processador Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz, 8GB de memória RAM e sistema Ubuntu/Linaro 4.6.3-1ubuntu4. O compilador REGIMap foi previamente compilado pelo autor.

Na Tabela 5.4 são mostrados resultados comparativos de tempo de compilação e intervalo de iniciação entre a heurística MSG com a estratégia de registros locais e o compilador REGIMap. Os *benchmarks* usados para teste têm de 11 a 66 operações e foram escolhidos porque o REGIMap não consegue achar um escalonamento válido em tempo menor que 30 minutos para grafos maiores, quando usada a arquitetura de 16 PEs em malha. Por exemplo, o REGIMap não conseguiu encontrar uma solução em menos de 24 horas para o *benchmark* DCT, que tem 92 operações. A coluna

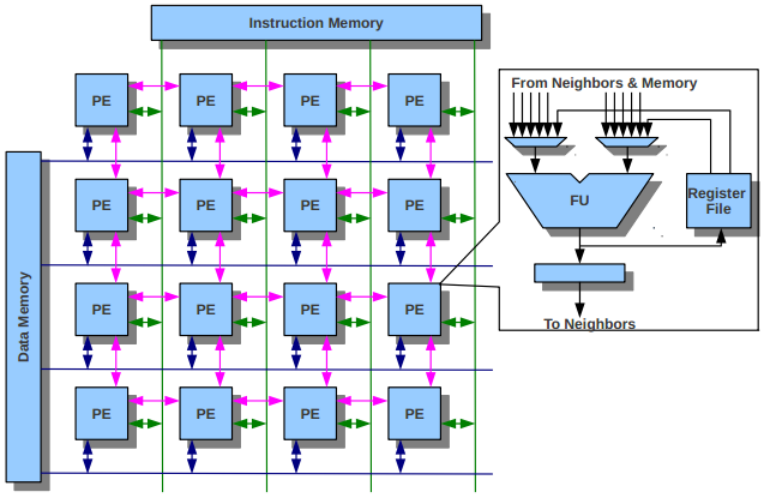


Figura 5.2. CGRA 4x4, PEs são interconectados em malha 2-D. Cada PE é uma ALU somado a um banco de registros locais. Figura extraída de [Hamzeh et al., 2013].

Benchmark	Op	Op+R	Tempo MSG	Tempo REGIM.	II MSG	II REGIM.
hal	11	12	0,61	24	2	1
horner	17	21	0,48	92	2	2
arf	28	35	0,23	496	3	2
ewf	34	52	0,37	931326	4	9
fir	44	48	0,12	3408	6	3
Fir16	49	64	0.37	215853	5	6
h2v2	51	78	0,57	7072	6	4
feedback	53	66	0,16	7244	6	4
FilRGB	57	73	0,39	9117	5	5
cosine1	66	71	0,48	20493	7	6
Média	**41**	**52**	**0,38**	**119512**	**4,6**	**4,2**

Tabela 5.4. Comparação entre a heurística MSG com registros locais e REGIMap. Os tempos estão em milissegundos (10^{-3}s).

OP+R mostra o número PEs utilizados para computação e registros, pela heurística MSG com registros locais. Em seguida, os tempos obtidos por cada heurística são apresentados, assim como o desempenho do mapeamento obtido, valor do intervalo de iniciação ou vazão.

Observa-se na tabela que a diferença média no tempo de compilação é de 5 ordens de grandeza, REGIMap é aproximadamente 315 mil vezes mais lento, variando de 40 a 2 milhões e 500 mil vezes mais lento. O motivo dessa diferença foi discutida no capítulo 2: diferenças entre as arquiteturas, malha e *crossbar*, e diferença entre as heurísticas, várias tentativas para cada valor do II no REGIMap

contra uma tentativa por II do MSG. Além disso, os tempos do REGIMap variam muito em relação à média, devido à característica de muitas tentativas da heurística.

Com relação ao intervalo de iniciação, o REGIMap consegue resultados pouco melhores, 91% do valor médio obtido pela heurística MSG. Isto ocorre porque REGIMap otimiza a utilização de registros locais e evita o uso de PEs apenas como registradores. A consequência disso está no valor total de PEs utilizados para o mapeamento. Enquanto a heurística MSG utilizará exatamente o valor descrito na coluna OP+R da Tabela 5.4, que é 52 na média, REGIMap utilizará um número menor de PEs para o mapeamento de todas as operações do laço. REGIMap alcança bons resultados de desempenho, quando comparado à heurística MSG e ao estado da arte. Porém, limita-se apenas à compilação estática, devido ao seu tempo de compilação médio ser da ordem de minutos.

5.4 Tamanho da Arquitetura

Nesta seção são avaliadas arquiteturas com número de PEs diferentes, com o objetivo de mostrar a evolução do escalonamento quando aumenta-se o número de PEs disponíveis. O número de PEs para cada arquitetura testada é referente à avaliação mostrada na Tabela 1.2. Os resultados são relativos à arquitetura e heurística com a estratégia de registros locais (Figura 3.10(b)), que se mostrou mais eficiente. São avaliados o intervalo de iniciação, o ILP (*Instruction Level Paralelism*) e a ocupação, para três tamanhos diferentes de arquitetura: 16, 20 e 24 PEs. As três arquiteturas têm 4 PEs com suporte a operações de entrada externa de dados. O tamanho dos grafos varia de 28 a 196 nós, sem registros de balanceamento, e estão ordenados por tamanho na tabela, totalizando 22 grafos. Neste experimento são testados *benchmarks* maiores, pois são avaliadas duas arquiteturas maiores.

A Figura 5.3 mostra o gráfico com os resultados referentes ao intervalo de iniciação. A Figura 5.4 apresenta o gráfico do paralelismo em nível de instrução (ILP), que é calculado pelo número de operações divido pelo II, sem contar registros. A Figura 5.5 apresenta o gráfico da ocupação, que representa a porcentagem de recursos da arquitetura que serão usados durante a execução, onde 100 indica que todos os PEs estão ocupados durante a execução e subtrair a ocupação de 100 indica o percentual de recursos ociosos. Todos os gráficos mostram o resultado de cada *benchmark* para cada um dos três tamanhos de arquiteturas propostos. Alguns *benchmarks* não apresentam resultados para a menor arquitetura ou para as duas menores, pois a heurística MSG falha ao tentar mapear nestes casos, assim, os

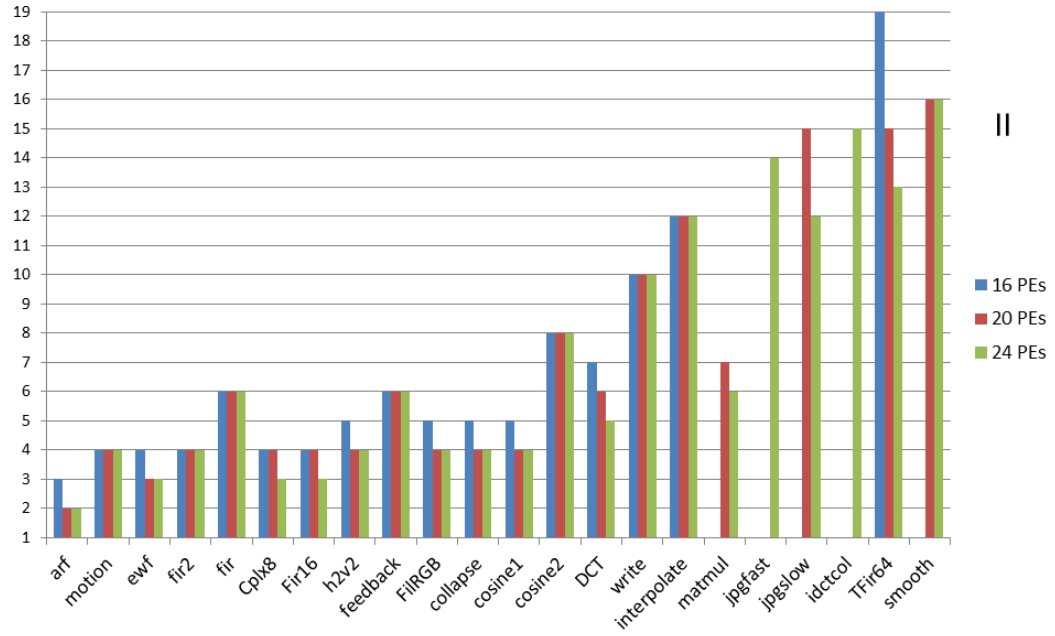


Figura 5.3. Intervalo de Iniciação para três arquiteturas: 16, 20 e 24 PEs.

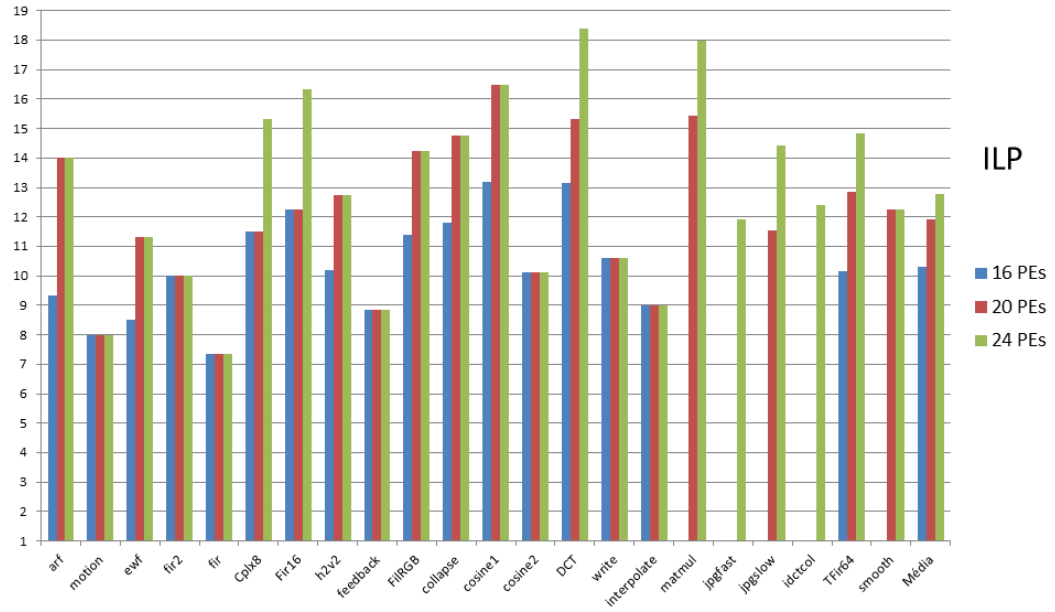


Figura 5.4. Paralelismo em nível de instrução para três arquiteturas: 16, 20 e 24 PEs.

gráficos mostram apenas as barras referentes aos *benchmarks* nos quais foi possível gerar um mapeamento válido.

Este experimento permite fazer um conjunto de diferentes análises. Primeiramente, para 7 dos 22 *benchmarks* testados, a qualidade do escalonamento não melhora quando mais PEs são adicionados. Por exemplo, o *benchmark interpol*

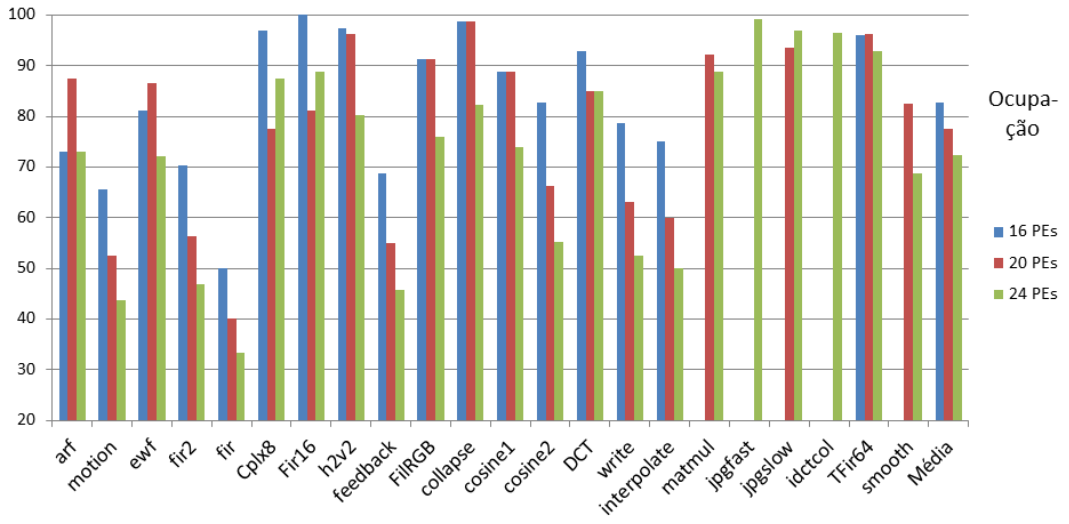


Figura 5.5. Ocupação para três arquiteturas: 16, 20 e 24 PEs.

permanece com $II = 12$ para todas as arquiteturas (Figura 5.3). Neste caso, o número máximo de operações entradas ($MaxIn$) permitidas na arquitetura limita o paralelismo máximo. Isso ocorre também para os *benchmarks feedback, fir, fir2, h2v2, cosine1, cosine2* e *write*. Além disso, a arquitetura 16 PEs tem uma ocupação média melhor (Figura 5.5), pois aumentar o número de PEs não traz benefícios no caso dos *benchmarks* limitados pelo número máximo de operações de entradas, como mostram as médias exibidas nas últimas barras dos gráficos.

A segunda análise está relacionada às médias apresentadas na Figura 5.4, que mostra o ILP médio de 10,3, 11,9 e 12,8 para 16, 20 e 24 PEs, respectivamente. Os resultados demonstram que a arquitetura de 20 PEs melhora o ILP apenas em 15% mas usa 25% mais PEs. Além disso, a arquitetura de 24 PEs aumenta o número de PEs em 50%, no entanto o ILP é somente 24% melhor que a arquitetura de 16 PEs. Para a maioria dos casos, a arquitetura de 16 PEs mostra um bom resultado para uma arquitetura pequena, em termos de qualidade do escalonamento, ILP e ocupação. Entretanto, para grafos grandes, últimos seis *benchmarks* de cada gráfico, devido à heurística gulosa que faz apenas uma tentativa de mapeamento por II e o pequeno número de registros locais, o escalonamento para a arquitetura de 16 PEs pode falhar e, por esta razão, arquiteturas maiores, como 20 e 24 PEs, são necessárias.

Para mostrar o impacto do número de PEs com suporte a entradas externas, foi realizado o segundo experimento, que consiste em avaliar uma arquitetura de 24 PEs com dois PEs a mais com suporte a entradas externas, em relação à arquitetura de 24 PEs do experimento anterior. Foram selecionados somente os *benchmarks* que

Nome	II 4in / 6in	ILP 4in / 6in	Ocupação (%) 4in / 6in
fir2	4 / 3	10,0 / 13,3	46,9 / 66,7
fir	6 / 4	7,3 / 11,0	33,3 / 54,2
h2v2	4 / 4	12,8 / 12,8	79,2 / 79,2
feedback	6 / 4	8,8 / 13,3	45,8 / 65,6
cosine1	4 / 4	16,5 / 16,5	74,0 / 71,9
cosine2	8 / 6	10,1 / 13,5	55,2 / 67,4
write	10 / 7	10,6 / 15,1	52,5 / 76,8
Média		10,9 / 13,6	55,3 / 68,8

Tabela 5.5. Comparação das arquituras de 24 PEs com 4 e 6 entradas externas, para os 7 *benchmarks* limitados por entradas.

têm o II mínimo limitado pelo número de entradas, totalizando 7. Os *benchmarks* que não têm MinII limitado por entrada sofrem pouco impacto quando aumenta-se o número de entradas.

A Tabela 5.5 mostra os resultados em termos de II, ILP e Ocupação, para as arquiteturas de 24 PEs com 4 (4in) e 6 (6in) entradas externas. Para 5 dos 7 *benchmarks* o resultado do II melhorou, para os outros dois ficou igual. A média do ILP aumentou em 25% e a média da ocupação em 24,5%.

5.5 Tempo de Compilação

Esta seção mostra os resultados do tempo de compilação da heurística MSG com a estratégia de registros locais. A solução proposta nesta dissertação apresenta uma redução significativa no tempo de compilação, quando comparada às principais soluções da literatura da área. Esta redução substancial permite o uso de uma implementação *Just-In-Time*, que é uma técnica não factível em soluções anteriores. Considerando que abordagens anteriores [Park et al., 2006; Oh et al., 2009; Yoon et al., 2009; Hamzeh et al., 2012; Chen & Mitra, 2012; Hatanaka & Bagherzadeh, 2007] usam um processador de propósito geral (CPU) para medir o tempo de compilação, nesta seção é mostrado o resultado da execução da heurística MSG numa CPU comercial. Adicionalmente, a heurística foi avaliada em dois processadores *softcore* implementados em FPGA. *Softcores* podem ser uma solução para futuras abordagens completamente dinâmicas.

A Tabela 5.6 mostra o tempo de compilação para três plataformas diferentes. Primeiro, foi usado um processador superscalar comercial: um Intel i7, 1.73Ghz com 256Kb L1 cache. Para este processador, o algoritmo foi compilado usando o

Nome	CPU i7	MicroBlaze	ρVex
arf	7,20	232,79	107,23
motion	3,80	127,75	60,35
ewf	5,40	368,99	165,17
fir2	3,70	125,56	58,68
fir	3,80	120,54	56,52
Cplx8	8,80	339,37	153,86
Fir16	9,50	370,13	168,36
h2v2	14,20	570,02	261,17
feedback	4,30	163,60	75,86
FilRGB	9,70	387,17	175,27
collapse	14,60	610,13	274,04
cosine1	12,70	484,17	218,60
cosine2	5,60	246,49	111,82
DCT	33,30	1460,56	628,74
write	5,60	246,43	111,83
Tempo médio	9,5	390	175
Tempo total	142	5854	2627
Tempo por nó	0,17	6,99	3,14
Nº de ciclos por nó	292,57	1047,80	313,54

Tabela 5.6. Tempo de compilação em μ segundos para três processadores diferentes: Intel Core i7, MicroBlaze e ρ-Vex.

gcc 3.4.2 com a opção de otimização -O3. O segundo processador é o *softcore* Xilinx Microblaze, executando a uma frequência de 150Mhz e o código foi compilado usando Cygwin make 3.79.1 com a opção de otimização -O3. O terceiro processado é o ρ-Vex, um processador *softcore* VLIW [Wong & Anjam, 2009], executando a 100Mhz, com 4-issue. O código do algoritmo foi compilado usando o compilador HP Vex 3.41 com otimizações -fno-xnop e -O3.

A primeira análise significante está relacionada à unidade de tempo. Enquanto soluções anteriores usam segundos como unidade de tempo, a solução aqui proposta tem seus resultados apresentados em μ segundos (10^{-6}s). Esses resultados evidenciam reduções de até 6 ordens de magnitude para o tempo de compilação. As últimas três linhas na Tabela 5.6 mostram resultados médios para 15 *benchmarks* avaliados. Primeiro, o tempo total mostra que o processador i7 é, em média, 41,2x e 18,5x mais rápido que os processadores *softcore* Microblaze e ρ-Vex, respectivamente. Além disso, o *clock* do i7 é 11,3x e 17x mais rápido que esses processadores, respectivamente. Segundo, propõe-se neste trabalho normalizar os resultados para comparar a complexidade da implementação do algoritmo. A linha **Tempo por nó** retrata o tempo médio gasto para escalonar uma operação do *benchmark*. Por fim,

Algoritmo	Clock(Ghz)	Tempo (sec.) Grafo	Nó	Ciclos	Fator de Redução i7	ρVex
DRESC	2.66	104	0.73	$2.0\ 10^9$	$6.8\ 10^6$	$6.2\ 10^6$
EPIMAP	2.66	30	0.17	$4.5\ 10^8$	$1.6\ 10^6$	$1.4\ 10^6$
RF	1.0	110	0.5	$4.9\ 10^8$	$1.7\ 10^6$	$1.6\ 10^6$
EMS	2.66	5.6	0.04	$1.04\ 10^8$	$3.6\ 10^5$	$3.3\ 10^5$
G-Minor	2.66	3.4	0.04	$1.04\ 10^8$	$3.6\ 10^5$	$3.3\ 10^5$
RAM	2.66	3.9	0.01	$2.7\ 10^7$	$9.3\ 10^4$	$8.6\ 10^4$
MSPR	2.66	0.09	0.0002	$4.6\ 10^5$	$1.6\ 10^3$	$1.5\ 10^3$

Tabela 5.7. Tempo de compilação e ciclos por nó.

a linha **Nº de ciclos por nó** mostra o número médio de ciclos de processador gasto para mapear um nó. Esse resultado mostra que a complexidade de implementação é similar para os processadores i7 e ρ-Vex, e está em torno de 300 ciclos por nó.

A Tabela 5.7 apresenta o tempo de compilação, em ordem de magnitude, para sete abordagens *modulo scheduling* encontradas na literatura: DRESC [Mei et al., 2002], EMS [Park et al., 2008], RF [De Sutter et al., 2008], RAM [Oh et al., 2009], MSPR [Ferreira et al., 2011a], G-Minor [Chen & Mitra, 2012] e EPIMAP [Hamzeh et al., 2012], os quais os nomes estão listados na coluna *Algoritmo*. Os tempos de compilação foram obtidos nas respectivas referências, com exceção do DRESC, cujo resultado de tempo foi relatado por [Oh et al., 2009] e [Chen & Mitra, 2012]. A coluna *Clock* exibe a frequência de ciclo do processador usado para medir o tempo de compilação. A coluna *Grafo* mostra o tempo médio necessário para compilar um grafo inteiro. Em 6 dos 7 algoritmos, o tempo varia de 3,4 a 110 segundos por grafo, o que é praticável para compilação estática ou *offline*. Apenas o algoritmo MSPR apresenta tempo apropriado para a compilação *Just-In-Time*. Baseado nesses dados, tem-se computado o tempo médio gasto por nó, mostrado na coluna *Nó*. Além disso, também foi computado o número médio de ciclos para processar um nó. Finalmente, as duas últimas colunas representam o fator de redução em número de ciclos de *clock* por nó obtido pela abordagem proposta para a execução nos processadores i7 e ρ-Vex. Os resultados mostram que uma redução massiva variando de 3 a 6 ordens de magnitude foi alcançada.

Para alcançar essa redução significante no tempo de compilação, um conjunto de estratégias de projeto foram adotadas. Primeiramente, todos os trabalhos relacionados anteriores apresentam pseudo-algoritmos de alto nível, onde a implementação de alguns passos é complexa. A heurística MSG é muito simples e tem seu pseudo código (Algoritmo 1) muito próximo da implementação real na linguagem C. Além disso, na maioria das soluções, muitos escalonamentos possíveis são avaliados a cada

II e, para cada escalonamento, muitos passos de posicionamento e roteamento são realizados. Por outro lado, a solução proposta neste trabalho tenta apenas um escalonamento por II, e para nó é visitado somente uma vez. Adicionalmente, uma redução de complexidade é gerada devido ao uso da rede *crossbar* que simplifica os passos de posicionamento e roteamento a $O(1)$.

Como comparação final, compara-se com a solução MSPR, proposta por [Ferreira et al., 2011a]. MSPR usa uma abordagem similar, mas com rede de interconexão multiestágio. Entretanto, há uma diferença significante no tempo de compilação. A primeira razão de o MSPR ter resultado pior está relacionada ao roteamento, este tem custo $O(logN + 2^E)$, para a rede multiestágio, onde E é o número de estágios/níveis extras da rede. Além disso, há conflitos no roteamento, pois a rede multiestágio é bloqueante. Em segundo lugar, por causa dos conflitos da rede, todos os PEs são avaliados durante o posicionamento de um nó. Finalmente, o algoritmo MSPR tem uma implementação mais complexa e a qualidade dos resultados é pior, pois não economiza registros, através de compartilhamento de registro, e não usa registros locais para alcançar melhor II.

5.6 Falhas no CGRA

Esta seção tem por objetivo mostrar o impacto que falhas nos elementos de processamento podem acarretar ao resultado do algoritmo, em termos de II (vazão), ILP e ocupação. Mostra-se que o algoritmo suporta falhas e continua apresentando resultado satisfatório para falhas em até 25% dos PEs da arquitetura.

Para obter uma análise satisfatória, as falhas foram introduzidas aleatoriamente na arquitetura, somente nos PEs, impedindo o uso das unidades funcionais e registros dos PEs com falha. A quantidade de falhas varia de 6% a 25% das unidades. A geração de falhas sorteia, a cada execução do algoritmo, um conjunto de PEs com falha, mas sempre garantindo que haveria no mínimo 1 PE que suporte entrada externa de dados, pois, sem estes, ficaria impossível mapear, tornando o *hardware* inútil. O algoritmo MSG base foi testado e os *benchmarks* foram os 15 menores da Tabela 5.1, assim como utilizado em seções anteriores. Em seguida, também são mostrados resultados para a heurística MSG com registros locais.

A Figura 5.6 mostra a evolução da média do II, do ILP e da ocupação, em relação ao número de falhas inseridas na arquitetura, para a heurística básica. Para 1 falha há um aumento superior a 10% no intervalo de inicialização médio, chegando a alcançar os 31% de aumento para 4 falhas, 25% do total de unidades funcionais.

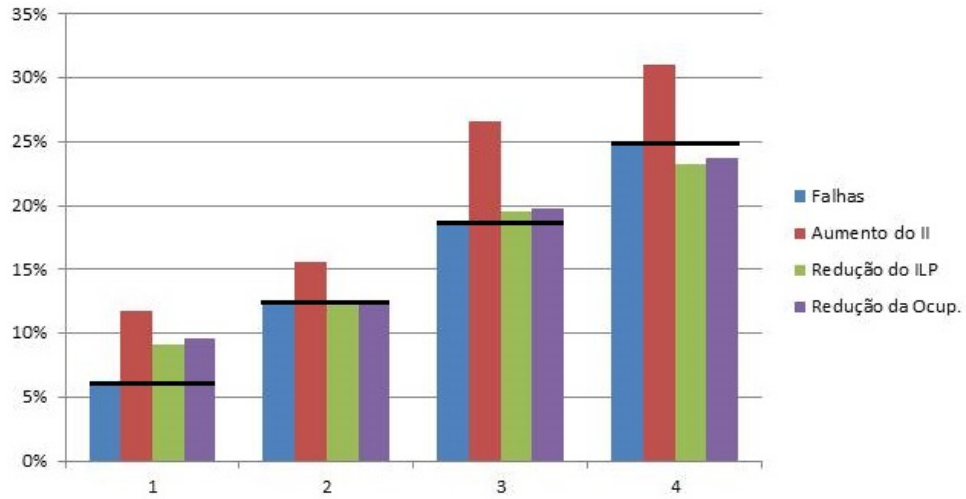


Figura 5.6. Impacto da inserção de falhas nos PEs da arquitetura de 16 PEs. Algoritmo MSG e arquitetura básicos.

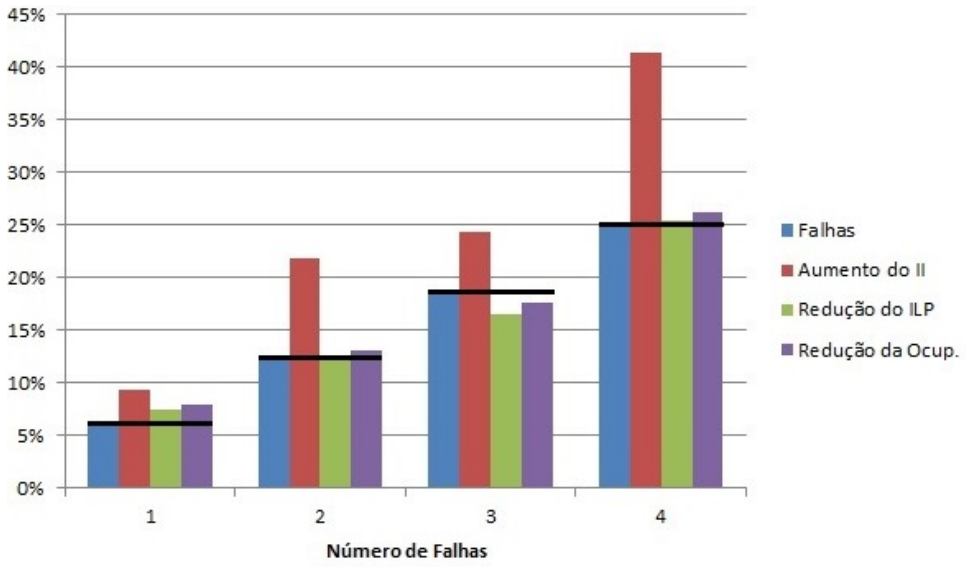


Figura 5.7. Impacto da inserção de falhas nos PEs da arquitetura de 16 PEs. Algoritmo MSG e arquitetura com registros locais.

Já para o ILP e a ocupação há também uma redução média com o aumento do número de falhas, sendo possível notar que a redução é quase idêntica ao aumento das falhas, percentualmente.

A Figura 5.7 mostra o resultado da inserção de falhas para o algoritmo com registros locais, para os mesmos *benchmarks*. O resultado é semelhante ao do algoritmo base, exceto pelo aumento mais significativo do intervalo de iniciação, que chegou a 41% para 25% de PEs com falhas, diferente da heurísitca base, que chegou

a 31% no mesmo teste. Essa piora mais significativa do II deve-se ao fato de que a arquitetura para a heurística com registros locais provê suporte ao uso dos dois registros de cada PE e cada redução de um PE reduz dois registros disponíveis, fazendo com que a heurística perca parte de sua vantagem, que é o uso dos dois registros. Assim, quanto mais falhas há na arquitetura, a estratégia dos registros locais obterá resultados mais semelhantes à estratégia base.

Apesar de somente a arquitetura de 16 PEs ter seus resultados mostrados nesta seção, os resultados se estendem para as demais arquiteturas, 20 e 24 PEs, e são similares, por esse motivo foram omitidas dos resultados. Em resumo, podemos dizer que o número percentual de falhas nos PEs corresponde à mesma perda percentual no rendimento da arquitetura, para o algoritmo MSG base. Em termos de vazão (II), que é a medida mais importante pois trata do intervalo em que os resultados serão obtidos, há uma queda média de 5% a mais que o percentual de falhas. Já para a estratégia de registros locais, a vazão cai cerca de 10% mais que o percentual de falhas, enquanto que as outras medidas seguem a taxa de falhas.

5.7 *Multikernels*

Esta seção apresenta resultados referentes ao mapeamento de dois *benchmarks* ao mesmo tempo. Os resultados mostram que a heurística consegue mapear esses *multikernels* e, além disso, há um ganho na qualidade do escalonamento em relação ao mapeamento dos trechos separadamente.

No experimento, foi testada uma nova arquitetura, com 32 PEs, sendo 6 com possibilidade de entrada externa de dados. O motivo do uso de uma arquitetura maior que as anteriores está relacionado ao tamanho dos *benchmarks* unidos, que variam de 62 a 216 operações. A arquitetura utilizada teve o número de entradas externas aumentado para 6, ao invés de 4, como nas arquiteturas anteriores. O motivo deste aumento é o mesmo explicado na seção 5.4: apenas 4 entradas externas limitam muito a qualidade do escalonamento para grafos com muitas operações de entrada. Baseado nos resultados de [Filho & Ferreira, 2012] e [Ferreira et al., 2013a], supomos que a arquitetura com 32 PEs também é implementável em FPGA, sob um custo aceitável.

A Tabela 5.8 mostra os resultados para os *multikernels* testados. Cada *multikernel* corresponde à união dos *benchmarks* que lhes dão nome. Suas intruções foram mapeadas simultaneamente de acordo com uma busca em largura, assim como *benchmarks* que representam apenas um *kernel*.

Nome	In	Op	Op+R	MinII	II_m	$II_1//II_2$	ΔII	ILP	Occ
arf+fir2	24	68	80	4	4	2//3	+1	17.0	62.5%
ewf+arf	10	62	84	3	3	2//2	+1	20.7	87.5%
collapse+arf	14	87	111	4	4	3//2	+1	21.8	86.7%
collapse+cosine1	22	125	148	5	6	3//3	0	20.8	77.1%
collapse+feedback	27	112	141	5	6	3//4	+1	18.7	73.4%
feedback+ewf	23	87	114	4	4	4//2	+2	21.8	89.1%
cos1+cos2	47	147	169	8	8	3//6	+1	18.4	66.0%
matmul+write	62	214	255	11	11	5//7	+1	19.5	72.4%
matmul+interpol	72	216	262	12	12	5//8	+1	18.0	68.2%

Tabela 5.8. *Multikernels.*

Os resultados mostram que em todos os casos testados houve ganho na utilização dos recursos do CGRA. A coluna $II_1//II_2$ mostra o intervalo de iniciação do primeiro e do segundo *benchmark* que forma o *multikernel*, quando estes estão mapeados sozinhos na mesma arquitetura. Por exemplo, o *multikernel arf+fir2* significa que o *benchmark arf* foi mapeado simultaneamente com *fir2* e a coluna $II_1//II_2$ mostra que *arf* tem II=2 quando mapeado sozinho na arquitetura e *fir2* tem II=3.

A coluna ΔII representa a diferença entre a soma $II_1 + II_2$ e o II alcançado para o *multikernel* (coluna II_m). Ou seja, $\Delta II = II_1 + II_2 - II_m$. Assim, a coluna ΔII mostra que o resultado do II dos *benchmarks* unidos é melhor do que a soma dos *benchmarks* separados. Isso mostra um ganho mínimo de 1 intervalo de iniciação a menos no escalonamento, aumentando a vazão.

Além disso, quando um grafo com um número reduzido de operações é mapeado, há uma sobra de PEs ociosas na arquitetura. Por exemplo, o *benchmark arf* tem apenas 28 operações. Ao mapeá-lo na arquitetura com 32 PEs é necessário II = 2, pois são usadas 28 PEs para as operações e 5 PEs como registros. Portanto sobram 31 PEs. Já o *benchmark* fir2 tem um II igual a 3 devido ao número restrito de entradas da arquitetura, usando somente 47 PEs, das 96 disponíveis (32 PEs com 3 partições temporais). Ao unir o mapeamento de arf com fir2, obtém-se uma ocupação maior e um II igual ao do maior *kernel*. Resumindo, aproveita-se o espaço livre deixado por um *kernel* para mapear outro. Para os *multikernel* testados, obteve-se uma ocupação média de 75,9% e um ILP médio de 37,8.

Trabalhos futuros devem considerar como adicionar um trecho de código quando o CGRA já está executando um outro código, como parar a execução de um trecho sem perder o contexto, entre outras considerações.

5.8 Algoritmo em *Hardware*

Esta seção tem como objetivo mostrar os resultados obtidos para o algoritmo em máquina de estados, que pode ser implementado em *hardware*, como mostrado no capítulo 4. Aqui apresenta-se apenas a estimativa inicial de qual seria o tempo gasto pelo algoritmo e uma comparação com a versão em *software*. A implementação prática do circuito não foi realizada.

Como é mostrado no capítulo 4, uma tarefa é realizada a cada estado da máquina, através de um conjunto de instruções realizadas e é mostrado um modelo de implementação para o algoritmo base. Porém, considera-se que com uma pequena extensão do modelo apresentado, adicionando estados à MEF e algumas estruturas adicionais à memórias, é possível estender a funcionalidade do modelo básico para o algoritmo com registros locais. Assim os resultados deste capítulo referem-se ao algoritmo com registros locais. Chamaremos esse algoritmo de RLMEF (registros locais em máquina de estados finitos).

A máquina de estados do algoritmo tem 32 estados, 18 estados a mais que a estratégia base. O motivo do aumento do número de estados deve-se a adição de funcionalidades de registros locais e compartilhamento de registros, e à exigência de maior controle que isso acarreta. É importante notar que cada estado da MEF respeita as retrições do *hardware*, não efetuando uma escrita e uma leitura ao mesmo ciclo/estado e não lendo de mais de uma posição de memória simultaneamente. Apesar de o número de estados ser 32 e apenas 5 *bits* serem necessários para codificar cada estado, o número médio de estados percorridos por cada operação mapeada é 20, considerando que o algoritmo reinicia a cada novo II.

A Tabela 5.9 mostra o número de ciclos (*clocks*) que o algoritmo RLMEF gasta para mapear cada *benchmark* de entrada, na arquitetura com 16 PEs. Foram feitas duas considerações em termos de frequência de cada ciclo. A primeira é cada ciclo a 100Mhz, cujo tempo é mostrado na coluna Tmp100. A segunda é 50Mhz, com o tempo total mostrado em Tmp50. O trabalho de [Filho & Ferreira, 2012] mostra que a frequência de uma implementação desse tipo pode chegar a 300Mhz, porém o algoritmo RLMEF é mais complexo.

Comparando o tempo do algoritmo RLMEF com os tempos do algoritmo em *software*, na Tabela 5.6, observa-se que RLMEF a 100Mhz tem um resultado muito semelhante à CPU i7, tendo a maior diferença no número de ciclos por nó, na qual a CPU i7 gasta quase 15 vezes mais. O motivo disso deve-se ao fato de cada ciclo do algoritmo em *hardware* realizar várias operações, enquanto que i7 é um processador

Benchmark	Clocks	Tmp100	Tmp50	MicroBlaze
arf	496	5,0	9,9	232,8
motion	337	3,4	6,7	127,8
ewf	904	9,0	18,1	369,0
fir2	346	3,5	6,9	125,6
fir	363	3,6	7,3	120,5
Cplx8	847	8,5	16,9	339,4
Fir16	971	9,7	19,4	370,1
h2v2	1711	17,1	34,2	570,0
feedback	529	5,3	10,6	163,6
FilRGB	1050	10,5	21,0	387,2
collapse	1737	17,4	34,7	610,1
cosine1	1403	14,0	28,1	484,2
cosine2	909	9,1	18,2	246,5
DCT	4372	43,7	87,4	1460,6
write	973	9,7	19,5	246,4
Médias	1129,9	11,3	22,6	390,3
Tempo total	-	169,5	339,0	5854,0
Tempo por nó	-	0,2	0,4	7,0
Nº de ciclos por nó	-	20,2	20,2	1047,8

Tabela 5.9. Clocks e tempos (em μ segundos) dos *benchmarks* para o algoritmo RLMEF. Tmp100 e Tmp50 representam os tempos para RLMEF, quando o ciclo do *hardware* é 100 e 50Mhz, respectivamente. MicroBlaze é o tempo de compilação no *softcore* Microblaze.

superescalar, que realiza *pipeline* de instruções, mas não consegue a mesma eficiencia de paralelismo do algoritmo em *hardware*. Com RLMEF a 50Mhz, o tempo da CPU i7 é mais de duas vezes mais rápido. Apesar de, em termos de tempo, a CPU i7 seja um pouco melhor que o algoritmo implementado em *hardware*, não há comparação quando leva-se em conta que o RLMEF estaria programado no mesmo FPGA que o próprio CGRA para o qual mapeia. Já a CPU i7 necessitaria de uma conexão externa ao FPGA.

Ao comparar o algoritmo RLMEF a 100Mhz com o MicroBlaze e o Vex, há um ganho de 34,5x e 15,5x no tempo, respectivamente. Ambas as soluções, *softcores* e RLMEF, estariam programadas diretamente no mesmo *hardware* que o CGRA, dentro do FPGA, o que é ideal para a solução de compilação dinâmica. Portanto, a versão RLMEF apresenta a possibilidade de um tempo configuração ainda menor, que pode ser avaliada em trabalhos futuros. Também deve-se considerar a maior complexidade de programação do algoritmo em *hardware*, onde cada trecho de código precisa de implementação e controle específicos.

6. Conclusões

O objetivo deste trabalho foi propor uma solução para o mapeamento de laços paralelizáveis por *software* em uma arquitetura CGRA, através de uma heurística *modulo scheduling* para o mapeamento e uma arquitetura CGRA derivada de [Ferreira et al., 2011a]. O requisito fundamental desta solução foi o tempo de execução da heurística, que deveria ser compatível com a compilação *Just-In-Time*. A solução partia do princípio de que tínhamos o grafo de fluxo de dados que descreve o laço paralelizável preparado no momento da execução da heurística, não fazendo a extração deste laço diretamente da aplicação.

Primeiramente foi proposta uma modificação na arquitetura proposta por [Ferreira et al., 2011a], substituindo a rede de interconexão multiestágio por uma rede *crossbar*. O objetivo desta troca era simplificar o roteamento, visto que na rede multiestágio é necessário encontrar um caminho para ligar uma unidade em outra, podendo haver bloqueios, e na rede *crossbar* a ligação é direta. No entanto, a rede *crossbar* tem elevado custo em área, $O(n^2)$, assim foram utilizadas apenas arquiteturas pequenas, onde o custo da rede *crossbar* é viável. Para o CGRA com 16 PEs, por exemplo, os custos são semelhantes à arquitetura em malha. Em seguida seguimos para a proposta da heurística MSG, que varre o grafo de fluxo de dados em largura, escalonando, posicionando e roteando suas operações. A heurística MSG é gulosa, fazendo apenas uma tentativa de mapeamento para cada valor de intervalo de iniciação, o que a torna seu tempo de execução baixo. Porém, MSG base não obteve resultados satisfatórios por não conseguir mapear alguns grafos com mais de 59 operações numa arquitetura de 16 PEs, devido à falta de registros para balanceamento. Assim, duas estratégias de melhoramento da heurística foram propostas.

A primeira estratégia, chamada de compartilhamento de registros, visou diminuir o número de PEs utilizadas apenas como registradores de balanceamento. Para isso foi realizada uma pequena modificação na heurística, que fazia com que PEs utilizadas como registros de balanceamento fizessem *multicasting*. Essa adição à heurística mostrou-se eficiente apenas para os *benchmarks* que tinham *multiscasts*, mas foi um passo importante para a próxima estratégia, a de registros locais.

A estratégia de registros locais foi também uma adição à heurística MSG base, mas que necessitou de modificação na arquitetura do CGRA, para que fosse possível utilizar os dois registros de um PE. Essa estratégia possibilitou uma melhor utiliza-

ção dos recursos do CGRA, visto que reduziu quase pela metade os PEs utilizados como registros de balanceamento. Por exemplo, da estratégia com registros compartilhados para a registros locais, a média de utilização de PEs como registros foi de 22 para 12 (dados extraídos da Tabela 5.2). Os resultados dessa economia foram mapeamentos melhores e abertura de possibilidade de mapeamento de um conjunto maior de grafos. A estratégia de registros locais sempre foi utilizada em conjunto com a estratégia de compartilhamento de registros.

Diversos experimentos foram realizados para avaliar a solução proposta. Foram mostrados os resultados relativos à economia de registros, intervalo de iniciação, tamanhos diferentes de arquitetura, tempo de compilação, falhas nos PEs, *multikernel* e implementação em *hardware*. Como dito anteriormente, os PEs foram tendo sua utilização otimizada com a evolução da heurística por meio das duas estratégias. O intervalo de iniciação, que é a medida para a vazão, também melhorou da versão base para registros locais, alcançando o valor ótimo em 10 dos 15 *benchmarks* testados. Tamanhos diferentes de arquitetura foram testados, considerando a limitação do número de PEs possíveis para a rede *crossbar*. Mostrou-se que aumentar o número de PEs melhora pouco o intervalo de iniciação, mas possibilita o mapeamento de laços com mais operações.

O tempo de compilação é um fator fundamental para esta solução que visa compilação em tempo de execução. Somente a heurística MSG com registros locais teve seu tempo de execução medido, pois mostrou-se a melhor estratégia para a solução. O tempo médio foi da ordem de microssegundos, promovendo um ganho de até 6 ordens de grandeza quando comparado a soluções da literatura. Assim, a solução proposta cumpre os requisitos para a compilação JIT.

O experimento para verificar o comportamento da heurística MSG em presença de falhas no circuito mostrou que a qualidade do mapeamento cai praticamente na mesma proporção que o número de falhas aumenta. A solução foi capaz de gerenciar as falhas até um total de 25% dos PEs, e acima deste valor, a heurística começa a não conseguir mapear os grafos, por falta de PEs disponíveis.

Foi também avaliada a possibilidade de mapear mais de um laço ao mesmo tempo no CGRA (*multikernels*), visando futuras implementações compatíveis com os sistemas computacionais dinâmicos atuais.

Por fim foi mostrada uma avaliação dos resultados teóricos para o modelo em *hardware* da solução. Os resultados experimentais mostraram que a implementação em *hardware* pode alcançar tempos de compilação ainda menores. Mas, como o modelo foi apenas simulado, os resultados contribuem apenas no sentido de uma avaliação inicial. A implementação é uma sugestão para trabalho futuro. Os tra-

balho de [Filho & Ferreira, 2012] e [Mucida et al., 2012] mostram implementações para algoritmos semelhante à heurística MSG.

Em resumo, foi mostrado neste trabalho que a combinação de uma arquitetura CGRA que possibilita posicionamento e roteamento diretos com uma heurística baseada em decisões gulosas e estruturas de dados simples, torna possível alcançar alto desempenho para a compilação *Just-In-Time*.

6.1 Trabalhos Futuros

Esta seção visa propor sugestões para a continuação do trabalho apresentado nesta dissertação. As propostas baseiam-se em resultados deste trabalho e de outros trabalhos da literatura [Hamzeh et al., 2012, 2013]

6.1.1 Rebalanceamento

Propõe-se adicionar a funcionalidade de rebalancear grafos na heurística MSG, para melhorar o intervalo de iniciação para alguns casos e possibilitando o mapeamento de outros. Para isso, usa-se um registrador para armazenar o resultado de uma operação, enquanto o posicionamento de algumas operações será redistribuído através das partições temporais. Porém, adiciona-se o desafio de não fazer com que a heurística gaste muito tempo para executar o rebalanceamento.

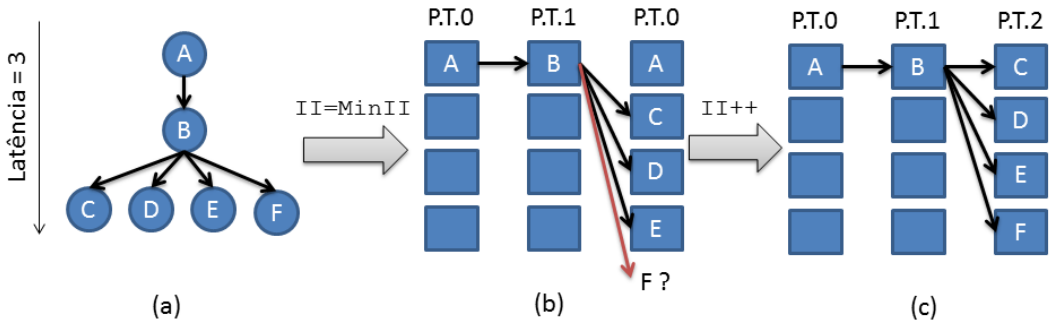


Figura 6.1. Exemplo de aumento desnecessário do II: (a) Grafo de fluxo de dados (b) Tentativa de mapear com o $II = MinII$ (c) Mapeamento final com II=3.

A Figura 6.1 mostra um caso em que a heurística MSG não alcança o mínimo II mesmo quando há PEs suficientes e não há limite de entradas. O motivo de não conseguir mapear com o intervalo de iniciação mínimo é devido ao desbalanceamento de operações alocadas em cada partição temporal, sendo que a partição 0 necessitaria

de 1 PE e a partição 1, de 5 PEs. A Figura 6.1(a) mostra um grafo de fluxo da dados em que há um *multicasting* da operação B para C, D, E e F, que deve ser mapeado numa arquitetura com 4 PEs e rede *crossbar*. Como são 6 operações, o MinII é 2.

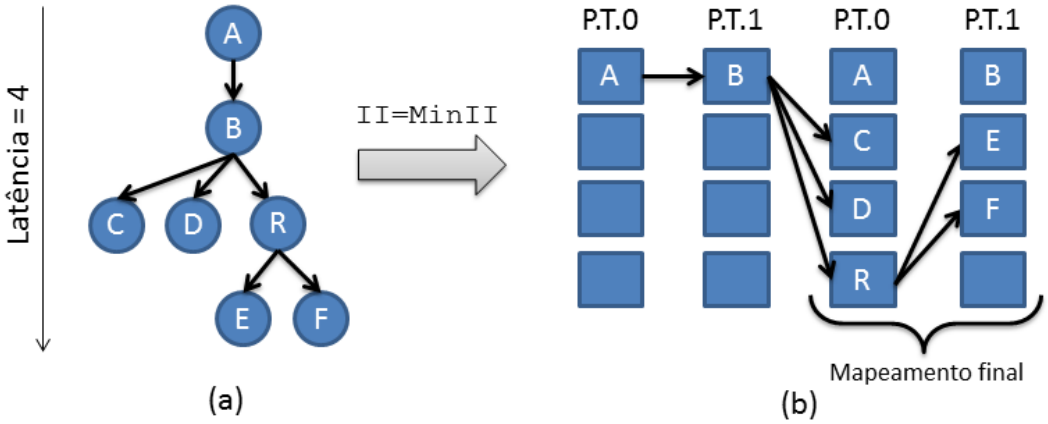


Figura 6.2. Deslocamento de operações (a) Grafo de fluxo de dados com operações deslocadas (b) Mapeamento realizado com $II = MinII$.

A heurística inicia com o MinII, porém falha ao posicionar a operação F na partição temporal 1, por falta de um PE disponível, como mostrado na Figura 6.1(b). Assim, o algoritmo reinicia com $II = 3$, consegue encontrar um mapeamento válido e termina, como mostra a Figura 6.1(c). Entretanto, não é necessário usar três partições temporais. A Figura 6.2 mostra que se o grafo sofrer uma pequena modificação no seu escalonamento, através da inserção de um registrador para armazenar o valor de B, a heurística poderá evitar o uso desbalanceado das partições temporais e, consequentemente, conseguirá mapear com II igual a 2. Assim, a heurística deve identificar quando ocorrerá esse desbalanceamento e tentar resolver.

A Figura 6.3 mostra um exemplo real no qual o intervalo de iniciação poderia ser menor se a técnica anterior fosse utilizada. O escalonamento do *benchmark cosine1* é mostrado, para uma arquitetura fictícia que tem número infinito de PEs disponíveis e com intervalo de iniciação fixo em 5, que é o II mínimo para um CGRA de 16 PEs, na qual a heurística MSG obtém $II = 7$ para este *benchmark*. O motivo de fixar o II em 5 é obter a informação de qual partição temporal está sendo sobrecarregada e causando a falha que implica em incremento do intervalo de iniciação, como mostrado nas figuras anteriores. As operações estão indicadas por formas geométricas sem preenchimento e os registros indicados pelas formas com preenchimento preto, cada tipo de forma indica uma partição temporal diferente.

Abaixo estão indicados os totais de operações e registros, por partição temporal. Podemos observar que a latência do escalonamento é 9 (T0 a T8) e, se usarmos a técnica de rebalanceamento, a latência deve aumentar, porém tentando fazer com que o intervalo de iniciação permaneça no mínimo. Além disso, observa-se a necessidade de utilização de 8 registros de balanceamento, correspondendo a utilização de 5 PEs. Isso leva a ocupação da arquitetura para 95% com $II = 5$, o que dificulta o reposicionamento de operações da recomputação pela falta de espaço para mover. Mas, não conseguindo mapear com $II = 5$, ainda há a possibilidade de tentar com 6 partições temporais, que é melhor do que as 7 que a heurística MSG consegue.

É possível observar que as partições temporais 1, 2 e 3 estão sobrecarregadas, como mostra a Figura 6.4. A técnica de rebalanceamento poderia tentar utilizar mais a partição temporal 4, pois ela está apenas com metade dos PEs utilizados. Como esta técnica fará isso será o trabalho futuro sugerido.

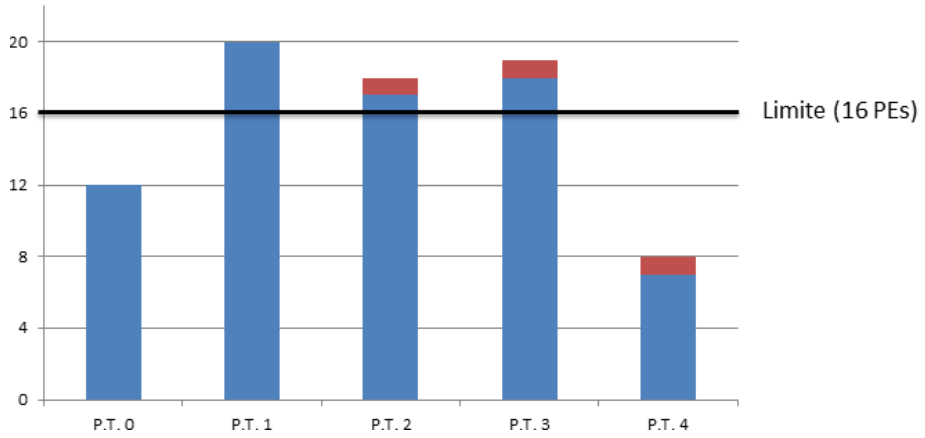


Figura 6.4. Espalhamento do posicionamento do *benchmark cosine1* nas partições temporais.

6.1.2 Banco de registradores

Baseado na ideia de tentar não bloquear o elemento de processamento quando usa-se registros [Hamzeh et al., 2013], propõe-se como trabalho futuro implementar esta estratégia. Como mostrado nos resultados, a solução MSG gasta aproximadamente 20% dos PEs utilizados como registros de balanceamento. Se conseguirmos reduzir esse valor, conseguiríamos escalonamentos melhores. Assim, podemos utilizar uma arquitetura que possibilite que os registros de balanceamento não usem os PEs.

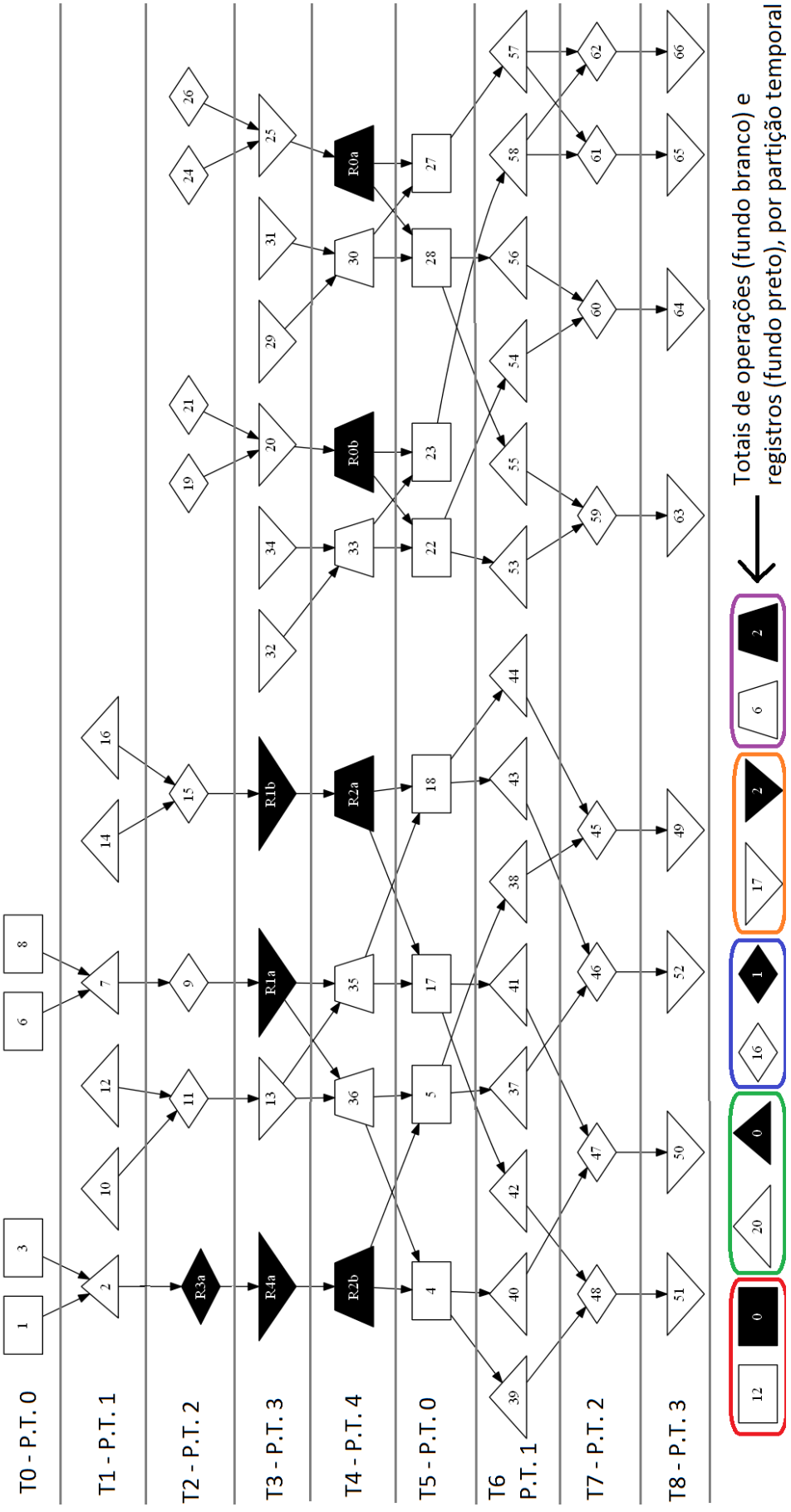


Figura 6.3. Exemplo de escalonamento do *benchmark* cosine1.

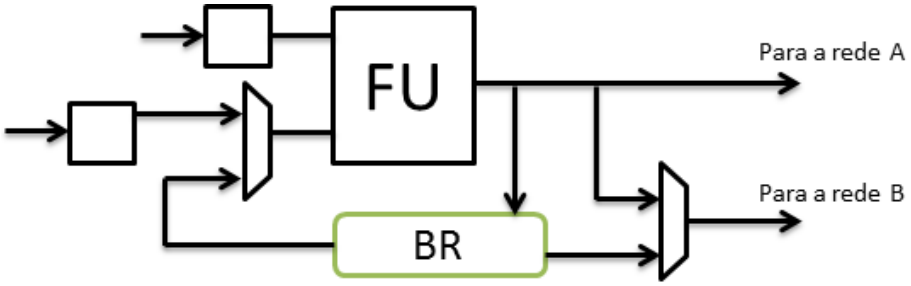


Figura 6.5. Estrutura da arquitetura com banco de registros locais (BR).

Um exemplo é a arquitetura apresentada na Figura 6.5, que tem um banco de registradores no qual o resultado de cada PE pode ser gravado ou lido para um PE atrvés da rede. A ideia seria mapear o grafo da Figura 6.6 com intervalo de iniciação igual a 1. Na arquitetura usada nesta dissertação isso não é possível, como mostrado no exemplo 1 da Seção 3.7. O mapeamento para a nova arquitetura é mostrado na Figura 6.6.

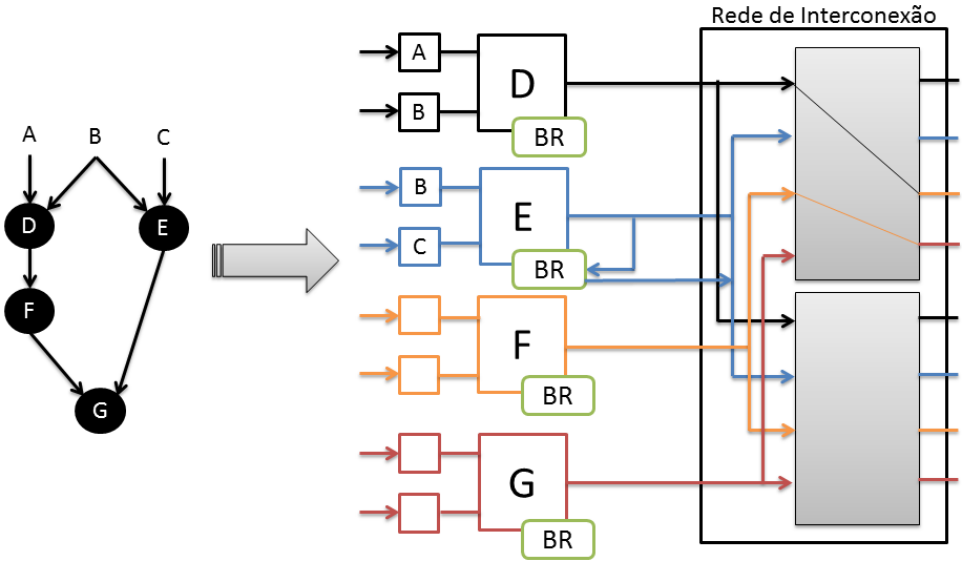


Figura 6.6. Grafo de fluxo de dados de exemplo com mapeamento com II = 1.

A operação E está desbalanceada em relação a operação F, ambas são entradas para a operação G. Ao invés de alocar um registrador de um PE para armarzenar o valor de E, o algoritmo irá armazenar a saída de E no banco de registradores local e fazer com que o PE que está G leia este valor através da rede de interconexão. Detalhes da implementação ficam para o trabalho futuro.

Referências Bibliográficas

Ahn, M.; Yoon, J. W.; Paek, Y.; Kim, Y.; Kiemb, M. & Choi, K. (2006). A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. Em *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pp. 363--368, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

Allan, V. H.; Jones, R. B.; Lee, R. M. & Allan, S. J. (1995). Software pipelining. *ACM Comput. Surv.*, 27(3):367--432.

Babb, J.; Frank, M.; Lee, V.; Waingold, E.; Barua, R.; Taylor, M.; Kim, J.; Devabhaktuni, S. & Agarwal, A. (1997). The raw benchmark suite: computation structures for general purpose computing. Em *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pp. 134 –143.

Bouwens, F.; Berekovic, M.; Kanstein, A. & Gaydadjiev, G. (2007). Architectural exploration of the adres coarse-grained reconfigurable array. Em *Proc. ARC*, pp. 1–13.

Brant, A. & Lemieux, G. (2012). Zuma: An open fpga overlay architecture. Em *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pp. 93 –96.

Callahan, T. J. & Wawrzynek, J. (2000). Adapting software pipelining for reconfigurable computing. Em *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '00, pp. 57--64, New York, NY, USA. ACM.

Chen, D. C. (1992). *Programmable arithmetic devices for high speed digital signal processing*. PhD thesis, Berkeley, CA, USA. UMI Order No. GAX93-04875.

Chen, L. & Mitra, T. (2012). Graph minor approach for application mapping on cgras. Em *FPT*, pp. 285–292.

Coole, J. & Stitt, G. (2010). Intermediate fabrics: virtual architectures for circuit portability and fast placement and routing. Em *Proceedings of the eighth*

IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES/ISSS '10, pp. 13--22, New York, NY, USA. ACM.

De Sutter, B.; Coene, P.; Vander Aa, T. & Mei, B. (2008). Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. Em *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pp. 151--160, New York, NY, USA. ACM.

Ferreira, R.; Damiany, A.; Vendramini, J.; Teixeira, T. & Cardoso, J. (2009a). On simplifying placement and routing by extending coarse-grained reconfigurable arrays with omega networks. Em Becker, J.; Woods, R.; Athanas, P. & Morgan, F., editores, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5453 of *Lecture Notes in Computer Science*, pp. 145–156. Springer Berlin Heidelberg.

Ferreira, R.; Duarte, V.; Meireles, W.; Pereira, M.; Carro, L. & Wong, S. (2013a). A just-in-time modulo scheduling for virtual coarse-grained reconfigurables architectures. Em *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, Samos, Greece. IEEE Computer Society.

Ferreira, R.; Garcia, A.; Teixeira, T. & Cardoso, J. M. P. (2007). A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures. *VLSI, IEEE Computer Society Annual Symposium on*, 0:61–66.

Ferreira, R.; Laure, M.; Beck, A.; Lo, T.; Rutzig, M. & Carro, L. (2009b). A low cost and adaptable routing network for reconfigurable systems. Em *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8.

Ferreira, R.; Laure, M.; Rutzig, M.; Beck, A. & Carro, L. (2008). Reducing interconnection cost in coarse-grained dynamic computing through multistage network. Em *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 47–52.

Ferreira, R.; Rocha, L.; Santos, A.; Nacif, J.; Carro, L. & Wong, S. (2013b). A run-time graph-based polynomial placement and routing algorithm for virtual fpgas. Em *23rd International Conference on Field Programmable Logic and Applications*, Porto, Portugal. IEEE Computer Society.

Ferreira, R.; Vendramini, J.; Mucida, L.; Pereira, M. & Carro, L. (2011a). An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. Em

Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2011 Proceedings of the 14th International Conference on, pp. 195 –204.

Ferreira, R.; Vendramini, J. & Nacif, M. (2011b). Dynamic reconfigurable multicast interconnections by using radix-4 multistage networks in fpga. Em *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pp. 810–815.

Ferreira, R. S.; Cardoso, J. M.; Damiany, A.; Vendramini, J. & Teixeira, T. (2011c). Fast placement and routing by extending coarse-grained reconfigurable arrays with omega networks. *Journal of Systems Architecture*, 57(8):761 – 777.

Filho, W. D. M. & Ferreira, R. S. (2012). A hardware-assisted modulo scheduling, placement and routing algorithm for stream computing in coarse-grained reconfigurable architectures. Em *25th Symposium on Integraded Circuits and Systems Design - XI Microelectronics Students Forum (SForum)*.

Friedman, S.; Carroll, A.; Van Essen, B.; Ylvisaker, B.; Ebeling, C. & Hauck, S. (2009). Spr: an architecture-adaptive cgra mapping tool. Em *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pp. 191--200, New York, NY, USA. ACM.

Goldstein, S.; Schmit, H.; Budiu, M.; Cadambi, S.; Moe, M. & Taylor, R. (2000). Piperench: a reconfigurable architecture and compiler. *Computer*, 33(4):70 –77.

Group, E. (2013). Extensible, programmable and reconfigurable embedded systems group. http://express.ece.ucsb.edu/benchmark/.

Guo, M. (2005). Energy-aware compiler scheduling for vliw embedded software. Em *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pp. 197–.

Hamzeh, M.; Shrivastava, A. & Vrudhula, S. (2012). Epimap: using epimorphism to map applications on cgras. Em *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 1284--1291, New York, NY, USA. ACM.

Hamzeh, M.; Shrivastava, A. & Vrudhula, S. (2013). Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras). Em *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, New York, NY, USA. ACM.

Hartenstein, R. (2001). A decade of reconfigurable computing: a visionary retrospective. Em *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pp. 642 –649.

Hartenstein, R.; Kress, R. & Reinig, H. (1994). A dynamically reconfigurable wavefront array architecture for evaluation of expressions. Em *Application Specific Array Processors, 1994. Proceedings. International Conference on*, pp. 404 –414.

Hatanaka, A. & Bagherzadeh, N. (2007). A modulo scheduling algorithm for a coarse-grain reconfigurable array template. Em *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1 –8.

Hauck, S. & DeHon, A. (2007). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hoo, C. H. & Kumar, A. (2012). An area-efficient partially reconfigurable crossbar switch with low reconfiguration delay. Em *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 400–406.

Mei, B.; Lambrechts, A.; Mignolet, J.-Y.; Verkest, D. & Lauwereins, R. (2005). Architecture exploration for a reconfigurable architecture template. *Design Test of Computers, IEEE*, 22(2):90 – 101.

Mei, B.; Vernalde, S.; Verkest, D.; De Man, H. & Lauwereins, R. (2002). Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. Em *Proc. FPT*, pp. 166 – 173.

Mei, B.; Vernalde, S.; Verkest, D.; Man, H. D. & Lauwereins, R. (2003). Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. Em *Proc. DATE*.

Mucida, L. (2013). Uma heurística gulosa para modulo scheduling em arquiteturas reconfiguráveis em tempo de execução. Em *Dissertaçãoo*, Viçosa, Minas Gerais. Universidade Federal de Viçosa.

Mucida, L.; Lopes, V.; Meireles, W. & Ferreira, R. (2012). Problem oriented approach to hardware-assisted algorithm design in c: A case study for scheduling, placement and routing. Em *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*, pp. 1–8.

Oh, T.; Egger, B.; Park, H. & Mahlke, S. (2009). Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. *SIGPLAN Not.*, 44(7):21--30.

Park, H.; Fan, K.; Kudlur, M. & Mahlke, S. (2006). Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. Em *Proc. CASES*, pp. 136--146.

Park, H.; Fan, K.; Mahlke, S. A.; Oh, T.; Kim, H. & Kim, H.-s. (2008). Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. Em *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 166--176, New York, NY, USA. ACM.

Park, H.; Park, Y. & Mahlke, S. (2009). Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. Em *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 370--380, New York, NY, USA. ACM.

Rau, B. R. (1994). Iterative modulo scheduling: an algorithm for software pipelining loops. Em *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pp. 63--74, New York, NY, USA. ACM.

Sanchez, E.; Sipper, M.; Haenni, J.-O.; Beuchat, J.-L.; Stauffer, A. & Perez-Uribe, A. (1999). Static and dynamic configurable systems. *Computers, IEEE Transactions on*, 48(6):556–564.

Sánchez, J. & González, A. (2000). Modulo scheduling for a fully-distributed clustered vliw architecture. Em *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pp. 124--133, New York, NY, USA. ACM.

Shrivastava, A.; Pager, J.; Jeyapaul, R.; Hamzeh, M. & Vrudhula, S. (2011). Enabling multithreading on cgras. Em *Parallel Processing (ICPP), 2011 International Conference on*, pp. 255–264.

Singh, H.; Lee, M.-H.; Lu, G.; Kurdahi, F.; Bagherzadeh, N. & Chaves Filho, E. (2000). Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 49(5):465 –481.

Swanson, S.; Michelson, K.; Schwerin, A. & Oskin, M. (2003). Wavescalar. Em *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 291 – 302.

Vendramini, J. & Ferreira, R. (2010). Parallel routing algorithm for extra level omega networks on reconfigurable systems. Em *Computing Systems (WSCAD-SCC), 2010 11th Symposium on*, pp. 1–8.

Wong, S. & Anjam, F. (2009). The delft reconfigurable vliw processor. Em *17th International Conference on Advanced Computing and Communications (ADCOM)*.

Yoon, J.; Shrivastava, A.; Park, S.; Ahn, M.; Jeyapaul, R. & Paek, Y. (2008). Spkm : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. Em *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pp. 776–782.

Yoon, J. W.; Shrivastava, A.; Park, S.; Ahn, M. & Paek, Y. (2009). A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(11):1565--1578.

Yun, H.-S. & Kim, J. (2001). Power-aware modulo scheduling for high-performance vliw processors. Em *Low Power Electronics and Design, International Symposium on, 2001.*, pp. 40–45.

Zalamea, J.; Llosa, J.; Ayguade, E. & Valero, M. (2001). Modulo scheduling with integrated register spilling for clustered vliw architectures. Em *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pp. 160–169.