Algoritmos para Alocação de Recursos em Arquiteturas Reconfiguráveis

Nahri Balesdent Moreano

Tese de Doutorado

Instituto de Computação
Universidade Estadual de Campinas

Algoritmos para Alocação de Recursos em Arquiteturas Reconfiguráveis

Nahri Balesdent Moreano[1]

Novembro de 2005

Banca Examinadora:

- Guido Costa Souza de Araújo (Orientador)
- Siang Wun Song
  Instituto de Matemática e Estatística – USP
- Flávio Rech Wagner
  Instituto de Informática – UFRGS
- Tomasz Kowaltowski
  Instituto de Computação – Unicamp
- Cid Carvalho de Souza
  Instituto de Computação – Unicamp
- Mário Lúcio Côrtes (Suplente)
  Instituto de Computação – Unicamp

[1]Financiada por UFMS e CAPES/PICDT

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Miriam Cristina Alves – CRB8a./859

Moreano, Nahri Balesdent
M813a Algoritmos para alocação de recursos em arquiteturas reconfiguráveis / Nahri Balesdent Moreano -- Campinas, [S.P. :s.n.], 2005.

Orientador : Guido Costa Souza de Araújo.
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Sistemas embutidos de computador. 3. Alocação de recursos. I. Araújo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Resource allocation algorithms for reconfigurable architectures

Palavras-chave em inglês (Keywords): 1. Computer architectures. 2. Embedded systems. 3. Resource allocation.

Área de concentração: Sistemas de computação.

Titulação: Doutora em Ciência da Computação.

Banca examinadora: Prof. Dr. Guido Costa Souza de Araújo (IC-UNICAMP)
Prof. Dr. Siang Wun Song (IME-USP)
Prof. Dr. Flávio Rech Wagner (INF-UFRGS)
Prof. Dr. Tomasz Kowaltowski (IC-UNICAMP)
Prof. Dr. Cid Carvalho de Souza (IC-UNICAMP)

Data da defesa: 09/11/2005

Algoritmos para Alocação de Recursos em Arquiteturas Reconfiguráveis

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Nahri Balesdent Moreano e aprovada pela Banca Examinadora.

Campinas, 09 de novembro de 2005.

Guido Costa Souza de Araújo (Orientador)

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

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 09 de novembro de 2005, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Siang Wun Song
IME/USP

Prof. Dr. Flávio Rech Wagner
Inst. de Informática - UFRGS

Prof. Dr. Tomasz Kowaltowski
IC - UNICAMP

Prof. Dr. Cid Carvalho de Souza
IC – UNICAMP

Prof. Dr. Guido Costa Souza de Araújo
IC – UNICAMP

Resumo

Pesquisas recentes na área de arquiteturas reconfiguráveis mostram que elas oferecem um desempenho melhor que os processadores de propósito geral (GPPs – *General Purpose Processors*), aliado a uma maior flexibilidade que os ASICs (*Application Specific Integrated Circuits*). Uma mesma arquitetura reconfigurável pode ser adaptada para implementar aplicações diferentes, permitindo a especialização do hardware de acordo com a demanda computacional da aplicação.

Neste trabalho, nós estudamos o projeto de sistemas dedicados baseado em uma arquitetura reconfigurável. Adotamos a abordagem de extensão do conjunto de instruções, na qual o conjunto de instruções de um GPP é acrescido de instruções especializadas para uma aplicação. Estas instruções correspondem a trechos da aplicação e são executadas em um *datapath* dinamicamente reconfigurável, adicionado ao hardware do GPP.

O tema central desta tese é o problema de compartilhamento de recursos no projeto do *datapath* reconfigurável. Dado que os trechos da aplicação são modelados como grafos de fluxo de dados e controle (*Control/Data-Flow Graphs* – CDFGs), o problema de combinação de CDFGs consiste em projetar um *datapath* reconfigurável com área mínima. Nós apresentamos uma demonstração de que este problema é NP-completo.

Nossas principais contribuições são dois algoritmos heurísticos para o problema de combinação de CDFGs. O primeiro tem o objetivo de minimizar a área das interconexões do *datapath* reconfigurável, enquanto que o segundo visa a minimização da área total. Avaliações experimentais mostram que nossa primeira heurística resultou em uma redução média de 26,2% na área das interconexões, em relação ao método mais utilizado na literatura. O erro máximo de nossas soluções foi em média 4,1% e algumas soluções ótimas foram obtidas. Nosso segundo algoritmo teve tempos de execução comparáveis ao método mais rápido conhecido, obtendo uma redução média de 20% na área. Em relação ao melhor método para área conhecido, nossa heurística produziu áreas um pouco menores, alcançando um *speedup* médio de 2500. O algoritmo proposto também produziu áreas menores, quando comparado a uma ferramenta de síntese comercial.

Abstract

Recent work in reconfigurable architectures shows that they offer a better performance than general purpose processors (GPPs), while offering more flexibility than ASICs (Application Specific Integrated Circuits). A reconfigurable architecture can be adapted to implement different applications, thus allowing the specialization of the hardware according to the computational demands.

In this work we describe an embedded systems project based on a reconfigurable architecture. We adopt an instruction set extension technique, where specialized instructions for an application are included into the instruction set of a GPP. These instructions correspond to sections of the application, and are executed in a dynamically reconfigurable datapath, added to the GPP's hardware.

The central focus of this theses is the resource sharing problem in the design of reconfigurable datapaths. Since the application sections are modeled as control/data-flow graphs (CDFGs), the CDFG merging problem consists in designing a reconfigurable datapath with minimum area. We prove that this problem is NP-complete.

Our main contributions are two heuristic algorithms to the CDFG merging problem. The first has the goal of minimizing the reconfigurable datapath interconnection area, while the second minimizes its total area.

Experimental evaluation showed that our first heuristic produced an average 26.2% area reduction, with respect to the most used method. The maximum error of our solutions was on average 4.1%, and some optimal solutions were found. Our second algorithm approached, in execution times, the fastest previous solution, and produced datapaths with an average area reduction of 20%. When compared to the best known area solution, our approach produced slightly better areas, while achieving an average speedup of 2500. The proposed algorithm also produced smaller areas, when compared to an industry synthesis tool.

Agradecimentos

Inúmeras pessoas contribuíram, direta ou indiretamente, para o desenvolvimento do presente trabalho, tornando este período de quatro anos extremamente enriquecedor para mim.

Agradeço em especial ao professor Guido Araújo que, com sua dedicação ao trabalho e entusiasmo admiráveis, aceitou me orientar e incontáveis vezes me guiou ao longo do caminho trilhado. Meu profundo e sincero agradecimento.

Aos professores Paulo Centoducatte, Rodolfo Jardim de Azevedo, Sandro Rigo e Cid Carvalho de Souza, que contribuíram para este trabalho com suas sugestões, críticas e colaborações.

Aos colegas do LSC e de sala, por diversas colaborações e por proporcionarem um ambiente de trabalho tão agradável. Em especial ao amigo Eduardo Wanderley Netto que esteve sempre ao lado, trocando apoio e incentivo, além de revisar meus textos e apresentações.

Agradeço à UFMS que investiu em minha capacitação e aos colegas do DCT/UFMS que possibilitaram minha ausência durante este período. Agradeço também ao IC/Unicamp por me oferecer a oportunidade de realizar este curso e a toda a sua comunidade por ter me acolhido tão bem.

A UFMS e a CAPES, pelo apoio financeiro indispensável. A Unicamp, UFMS, ACM e IC/Unicamp pelo auxílio financeiro para apresentação de trabalhos em conferências.

Finalmente, a meu marido, sempre incentivador da realização desta etapa, e a minha filha que, além de compreender meus tempos de ausência, ilumina todos os meus dias. E a meus pais que me ajudaram a chegar até aqui.

Sumário

Resumo **vii**

Abstract **viii**

Agradecimentos **ix**

Acrônimos **xvii**

1 Introdução **1**
1.1 Sistemas Dedicados e Computação Reconfigurável 1
1.2 Objetivos e Contribuições do Trabalho 3
1.2.1 Publicações 4
1.3 Organização do Texto 5

2 Arquiteturas Reconfiguráveis **7**
2.1 Modelo e Critérios de Classificação 7
2.1.1 Acoplamento 7
2.1.2 Granularidade do Bloco Lógico 9
2.1.3 Rede de Interconexões 10
2.2 Exemplos de Arquiteturas Reconfiguráveis 10
2.2.1 Chimaera 10
2.2.2 Garp 11
2.2.3 MorphoSys 12
2.2.4 PipeRench 13
2.2.5 RaPiD 14
2.2.6 Resumo 14
2.3 Desafios da Computação Reconfigurável 16
2.3.1 Reconfiguração Dinâmica 16
2.3.2 Mapeamento da Aplicação 17

2.3.3 *Memory Wall* . . . 18
2.3.4 Avaliação . . . 19

3 Modelo de Arquitetura Utilizado **20**
3.1 Especialização do Conjunto de Instruções . . . 20
3.2 Fluxo do Projeto . . . 21
3.3 Modelo de Arquitetura Reconfigurável . . . 24
3.3.1 Componente Reconfigurável . . . 25
3.4 Avaliação da Especialização do Conjunto de Instruções . . . 28
3.4.1 Ganho de Desempenho com Cluster . . . 28
3.4.2 Repetição de Clusters . . . 30
3.4.3 *Speedup* de ASIPs . . . 32
3.5 Exemplo de *Datapath* Reconfigurável . . . 33

4 Compartilhamento de Recursos **37**
4.1 O Problema de Combinação de CDFGs . . . 39
4.1.1 Portas de Entrada e Comutatividade . . . 44
4.2 Prova de NP-Completude . . . 44
4.3 Trabalhos Relacionados . . . 47
4.3.1 Emparelhamento de Peso Máximo em Grafo Bipartido . . . 47
4.3.2 Busca no Espaço de Soluções . . . 49
4.3.3 Modelo de Programação Inteira . . . 51
4.3.4 Outras Soluções . . . 51
4.4 Combinação Global vs. Incremental de CDFGs . . . 52

5 Otimização da Área de Interconexões **54**
5.1 Grafo de Compatibilidade . . . 54
5.2 Clique Máximo . . . 56
5.3 Algoritmo de Combinação de CDFGs . . . 58
5.4 Explorando a Comutatividade . . . 60
5.5 Ordem de Combinação dos CDFGs . . . 62
5.6 Resultados Experimentais . . . 63
5.6.1 Comparação com Emparelhamento Máximo . . . 64
5.6.2 Análise de Erro das Soluções . . . 65
5.6.3 Grau de Compartilhamento das Interconexões . . . 68

6 Otimização da Área Total **69**
6.1 Grafo de Compatibilidade e Clique de Peso Máximo . . . 69
6.2 Algoritmo de Combinação de CDFGs . . . 73

6.3 Resultados Experimentais 73
6.3.1 Comparação das Técnicas de Combinação de CDFGs 74
6.3.2 Avaliação da Abordagem de Combinação de CDFGs 80

7 Conclusão **84**
7.1 Resultados e Contribuições 84
7.2 Trabalhos Futuros 86

Bibliografia **88**

Lista de Tabelas

2.1 Características de algumas arquiteturas reconfiguráveis 15

3.1 Exemplos de categorias de clusters [16] 31
3.2 *Speedup* obtido com melhor cluster para cada aplicação [7] 33

5.1 CDFGs $G_i = (V_i, E_i), i = 1, \ldots, n$, de cada aplicação 64
5.2 Comparação com técnica de Emparelhamento Máximo 65
5.3 Limite inferior pelo modelo de PI e erro máximo do Clique Máximo 67

6.1 CDFGs $G_i = (V_i, E_i), i = 1, \ldots, n$, de cada aplicação 76
6.2 Comparação do método de Clique Compatível com outros algoritmos . . . 78
6.3 Comparação das abordagens de agrupamento e combinação de CDFGs . . 83

Lista de Figuras

1.1 Compromisso entre flexibilidade e desempenho 2

2.1 Modelo abstrato de arquitetura reconfigurável 8

3.1 Passos do fluxo de projeto 22
3.2 *Design productivity gap* [69] 24
3.3 Modelo da arquitetura do componente reconfigurável 26
3.4 Componente reconfigurável como RFU 27
3.5 Componente reconfigurável como coprocessador 28
3.6 Partes do código fonte dos programas "ADPCM encoder" e "decoder" . . . 29
3.7 Unidade funcional especializada para clusters do ADPCM 29
3.8 Código fonte de uma aplicação exemplo 34
3.9 CDFGs correspondentes aos corpos dos laços da Figura 3.8 e *datapath* reconfigurável gerado 34
3.10 *Datapath* configurado para executar corpo dos laços modificados 35

4.1 Combinação de CDFGs 38
4.2 Mapeamento de arcos e número de entradas nos MUXes 41
4.3 MUX de 8 entradas modelado como árvore de MUXes de 2 entradas 42
4.4 Diferentes grafos combinados: $A_b(\bar{G}') > A_b(\bar{G}'') = A_b(\bar{G})$ e $A_i(\bar{G}'') > A_i(\bar{G})$ 43
4.5 Compartilhamento de interconexões através da comutação de operandos . . 45
4.6 Combinação de CDFGs por emparelhamento de peso máximo em um grafo bipartido 49

5.1 Mapeamentos de arcos incompatíveis: a_3 é mapeado em b_4 e b_5 56
5.2 Passos do algoritmo de combinação de CDFGs 57
5.3 Combinação sem comutação dos CDFGs G_1 e G_2 da Figura 5.2 62
5.4 Número de interconexões de $\bar{G}$ e seus limites inferior e superior 66
5.5 Grau de compartilhamento das interconexões de $\bar{G}$ 68

6.1 Passos do segundo algoritmo de combinação de CDFGs 71

6.2 Aumento de área das demais técnicas em relação a Clique Compatível (barras ausentes representam um aumento de área de 0%) 77
6.3 *Speedup* de Clique Compatível (barras ausentes representam *speedup* de 0.1) 77
6.4 Aumento de área e tempo de execução, para "GSM encoder" (barras ausentes representam aumento de área de 0%) 79
6.5 Aumento de área e tempo de execução, para "JPEG encoder" (barras invertidas representam aumento de área de -0,1%) 79
6.6 CDFGs G_0 e G_1 81
6.7 Código VHDL agrupando G_0 e G_1 e *datapath* agrupado gerado por DC . . 81
6.8 Código VHDL combinando G_0 e G_1 e *datapath* combinado gerado por DC . 81
6.9 Relatórios de alocação e compartilhamento de recursos gerados por DC . . 82

Lista de Algoritmos

5.1 Combinação de CDFGs para minimizar área de interconexões 59
5.2 Construção do grafo de compatibilidade G_c 60
5.3 Construção do grafo combinado $\bar{G}$ 61

6.1 Combinação de CDFGs para minimizar área total 73
6.2 Construção do grafo de compatibilidade G_c 74
6.3 Construção do grafo combinado $\bar{G}$ 75

Lista de Acrônimos

Relacionamos a seguir os principais acrônimos usados neste texto.

ADL	Architecture Description Language
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application-Specific-Instruction Set Processor
CDFG	Control/Data-Flow Graph
FPGA	Field-Programmable Gate Array
FU	Functional Unit
GPP	General Purpose Processor
HDL	Hardware Description Language
HLS	High-Level Synthesis
LUT	Look-Up Table
MUX	Multiplexador
PI	Programação Inteira
RFU	Reconfigurable Functional Unit

Capítulo 1

Introdução

1.1 Sistemas Dedicados e Computação Reconfigurável

Como o mercado de sistemas dedicados tem crescido drasticamente nos últimos anos, programas executados nestes sistemas oferecem grandes desafios para projetistas de compiladores e arquiteturas. Alto desempenho, baixo consumo de energia, baixo custo, curto tempo de desenvolvimento e capacidade de manipular grandes volumes de dados são alguns dos requisitos impostos no desenvolvimento destas aplicações [85]. Diferentes arquiteturas têm sido utilizadas na implementação de sistemas dedicados, incluindo processadores de propósito geral (GPPs – *General Purpose Processors*), plataformas de computação reconfigurável e *Application Specific Integrated Circuits* (ASICs) [30]. Cada uma destas abordagens traz compromissos entre flexibilidade e desempenho do sistema.

Os GPPs representam as arquiteturas mais flexíveis, pois implementam um determinado conjunto de instruções e permitem a execução de programas descritos a partir destas instruções. Alterando-se simplesmente o software, diferentes aplicações podem ser executadas. No entanto, esta flexibilidade compromete seu desempenho. O processador precisa ler cada instrução da memória, decodificá-la e só então executá-la. Além disso, o conjunto de instruções de um GPP é imutável a partir do momento em que o processador é projetado. Assim, qualquer outra operação necessária para uma aplicação precisa ser implementada usando apenas as instruções existentes, o que pode resultar em um alto *overhead* de execução considerando-se cada operação individual. Para obter um desempenho razoável em uma grande diversidade de aplicações, os GPPs gastam uma

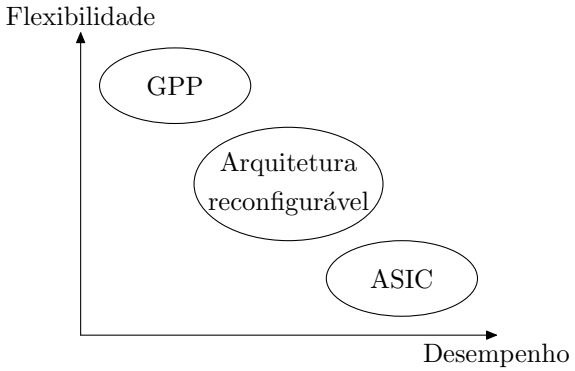


Figura 1.1: Compromisso entre flexibilidade e desempenho

área substancial do chip com memórias *cache* de instruções e de dados e mecanismos complexos para execução especulativa e predição de desvios.

Na outra extremidade do espectro estão os ASICs que implementam toda a aplicação em hardware. ASICs possuem um melhor desempenho quando comparados aos GPPs, pois a arquitetura é especializada para uma determinada aplicação. Assim, são capazes de extrair da aplicação bastante paralelismo de granularidade fina, e implementar de forma eficiente operações específicas desta. No entanto, implementações em ASIC não são flexíveis, pois após a fabricação o circuito não pode ser alterado. Se for necessária a modificação de alguma parte do circuito, ele deve ser reprojetado e fabricado novamente. Além disso, custos elevados de projeto e fabricação e a pressão por lançar produtos no mercado em tempos cada vez menores tornam a utilização de ASICs viável apenas para aplicações muito bem definidas e amplamente utilizadas, onde os altos custos possam ser amortizados pela grande quantidade de unidades vendidas [80].

Um novo paradigma em Computação vem surgindo [28, 67], no qual arquiteturas reconfiguráveis permitem um compromisso intermediário entre flexibilidade e desempenho. Tal compromisso possibilita um desempenho melhor que aquele de programas executados em GPPs, aliado a uma maior flexibilidade que as implementações em hardware dos ASICs [21], como ilustrado na Figura 1.1. Computação reconfigurável utiliza uma arquitetura que pode ser adaptada para realizar diferentes computações. A capacidade de reconfiguração permite que os recursos da arquitetura sejam reutilizados, simplesmente configurando o hardware novamente após o fim de uma computação. A reconfiguração pode ser definida estaticamente ou ser baseada em resultados intermediários gerados pelas computações anteriores [6].

Arquiteturas reconfiguráveis permitem, com sua maior flexibilidade, executar

diferentes classes de aplicação em uma única plataforma. Devido à sua adaptabilidade, elas também permitem a exploração do paralelismo de granularidade fina e grossa, através da adaptação do hardware para computações específicas de cada aplicação. Arquiteturas reconfiguráveis oferecem o que é denominado *density advantage*, por obterem um desempenho maior por unidade de área de silício que os GPPs [27].

Reconfiguração dinâmica permite também a alteração da configuração da arquitetura em tempo de execução, durante a operação do sistema. Assim, o hardware pode ser especializado não somente para cada aplicação mas também para diferentes partes de uma aplicação. De maneira similar ao mecanismo de memória virtual em GPPs, uma arquitetura dinamicamente reconfigurável pode ser vista como um "hardware virtual", em que configurações de hardware são carregadas e retiradas da arquitetura quando necessário, permitindo inclusive que implementações maiores que o hardware disponível possam ser realizadas [21].

Algumas arquiteturas reconfiguráveis foram projetadas e vêm sendo pesquisadas [13, 31, 35, 43, 65, 72]. Estas arquiteturas foram utilizadas para implementar aplicações de diversos domínios e algumas permitem um ganho significativo de desempenho quando comparadas aos GPPs. Alguns exemplos destas aplicações são: segurança, criptografia, compressão de dados, comunicações, algoritmos genéticos, emparelhamento de padrões, processamento de imagens, processamento de sinais e operações aritméticas complexas.

1.2 Objetivos e Contribuições do Trabalho

Este trabalho faz parte de um projeto desenvolvido no Laboratório de Sistemas de Computação (LSC) do Instituto de Computação da UNICAMP, cujo objetivo é o estudo e desenvolvimento de sistemas dedicados e arquiteturas reconfiguráveis.

O propósito final do projeto é desenvolver um processador com conjunto de instruções específico para a aplicação (ASIP – *Application-Specific Instruction Set Processor*), empregando-se a abordagem de especialização do conjunto de instruções. Nesta abordagem, o conjunto de instruções de um GPP é estendido com instruções específicas para um domínio ou aplicação. O problema consiste em detectar trechos na aplicação que, quando implementados em hardware como instruções especializadas, maximizem o desempenho da aplicação. Tais instruções são implementadas em unidades funcionais específicas, adicionadas ao hardware do GPP, formando assim um processador especializado.

A contribuição deste trabalho ao projeto do LSC é pesquisar técnicas que permitam sintetizar um *datapath* reconfigurável capaz de implementar uma unidade funcional especializada. O *datapath* deve poder ser reconfigurado dinamicamente realizando a computação correspondente a cada instrução específica. Para isso, os trechos da aplicação correspondentes às novas instruções são representados como grafos de fluxo de dados e controle (*Control/Data-Flow Graphs* – CDFGs).

A alocação e o compartilhamento de recursos no projeto do *datapath* reconfigurável são os principais objetos de estudo deste trabalho. Mais especificamente, deseja-se desenvolver técnicas para um problema denominado "Combinação de CDFGs", que consiste em encontrar um *datapath* com área mínima.

As principais contribuições do presente trabalho são:

- A estruturação das etapas do fluxo de projeto de sistemas dedicados, baseado em um modelo de arquitetura reconfigurável.
- Uma demonstração de que o problema de combinação de CDFGs para o projeto de um *datapath* reconfigurável, é NP-completo.
- O desenvolvimento de um algoritmo heurístico para este problema, visando minimizar a área das interconexões do *datapath* reconfigurável. Experimentos mostraram que esta heurística obteve uma redução média da área das interconexões de 26,2%, em relação ao método mais utilizado na literatura. O erro máximo das soluções geradas foi em média 4,1% e algumas soluções ótimas foram encontradas.
- O desenvolvimento de um segundo algoritmo heurístico para o problema, com o objetivo de minimizar a área total do *datapath*. Nos experimentos, esta heurística teve tempos de execução comparáveis ao método mais rápido conhecido, obtendo uma redução média da área de 20%. Em relação ao melhor método para área conhecido, o algoritmo proposto produziu *datapaths* com áreas um pouco melhores, com *speedups* significativos. Quando analisado em relação a uma ferramenta de síntese comercial, a heurística proposta também produziu *datapaths* com área menor.

1.2.1 Publicações

Este texto organiza os resultados parciais divulgados nas seguintes revistas, conferências e relatório técnico:

- *Efficient Datapath Merging for Partially Reconfigurable Architectures*,
  N. Moreano, E. Borin, C. Souza e G. Araujo,
  IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), volume 24, número 7, pp. 969-980, Julho 2005.

- *Fast Instruction Set Customization*,
  E. Borin, F. Klein, N. Moreano, R. Azevedo e G. Araujo,
  Proceedings of the Second IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMEDIA), pp. 53-58, Setembro 2004.

- *The Design of Dynamically Reconfigurable Datapath Coprocessors*,
  Z. Huang, S. Malik, N. Moreano e G. Araujo,
  ACM Transactions on Embedded Computing Systems (TECS), volume 3, número 2, pp. 361-384, Maio 2004.

- *The Datapath Merging Problem in Reconfigurable Systems: Lower Bounds and Heuristic Evaluation*,
  C. Souza, A. Lima, N. Moreano e G. Araujo,
  Proceedings of the Third International Workshop on Experimental and Efficient Algorithms (WEA), Lecture Notes in Computer Science, volume 3059, pp. 545-558, Maio 2004.

- *Datapath Merging and Interconnection Sharing for Reconfigurable Architectures*,
  N. Moreano, G. Araujo, Z. Huang e S. Malik,
  Proceedings of the 15th ACM International Symposium on System Synthesis (ISSS), pp. 38-43, Outubro 2002.

- *CDFG Merging for Reconfigurable Architectures*,
  N. Moreano, G. Araujo e C. Souza,
  Relatório Técnico número 18/2003, Instituto de Computação – UNICAMP, 2003.

1.3 Organização do Texto

Este texto está dividido em sete capítulos. O Capítulo 2 apresenta uma introdução à área de arquiteturas reconfiguráveis. Critérios para classificação destas arquiteturas e os

desafios relacionados ao seu projeto são abordados. Além disso, são descritas arquiteturas propostas em alguns trabalhos desenvolvidos na área de computação reconfigurável.

O Capítulo 3 apresenta uma plataforma para o projeto de sistemas dedicados, baseada em uma arquitetura reconfigurável. A abordagem de especialização do conjunto de instruções adotada neste trabalho é descrita, assim como as etapas do fluxo de desenvolvimento de um sistema dedicado. O modelo de arquitetura reconfigurável utilizado também é apresentado.

O problema de combinação de CDFGs para o projeto de um *datapath* reconfigurável é tratado no Capítulo 4. Uma descrição formal deste problema é apresentada, juntamente com uma prova de que ele é NP-completo. São também descritas diversas soluções já propostas para este problema, encontradas na literatura.

Os Capítulos 5 e 6 apresentam os dois algoritmos heurísticos propostos neste trabalho para o problema de combinação de CDFGs. O primeiro tem o objetivo de minimizar a área das interconexões do *datapath* reconfigurável, enquanto que o segundo visa a minimização da área total. Cada algoritmo é descrito e avaliado através de uma série de experimentos.

Finalmente, o Capítulo 7 apresenta as principais contribuições deste trabalho e aponta possíveis trabalhos futuros.

Capítulo 2

Arquiteturas Reconfiguráveis

Este capítulo dá uma visão geral da área de computação reconfigurável. A Seção 2.1 apresenta um modelo básico de arquitetura reconfigurável e enumera os principais critérios utilizados na classificação destas arquiteturas. Alguns dos principais projetos de arquiteturas reconfiguráveis existentes na literatura são descritos na Seção 2.2, enquanto que a Seção 2.3 discute alguns dos desafios mais relevantes da computação reconfigurável.

2.1 Modelo e Critérios de Classificação

As arquiteturas reconfiguráveis propostas até hoje (*circa* 2005) diferenciam-se consideravelmente em vários aspectos de sua organização. A Figura 2.1 mostra um modelo bastante abstrato de arquitetura reconfigurável que engloba os principais aspectos deste paradigma [6, 57]. Dentre as várias características que distingüem arquiteturas reconfiguráveis, as principais são o grau de acoplamento do componente reconfigurável com o processador hospedeiro, a granularidade do bloco lógico básico do componente reconfigurável e a rede de interconexões entre os blocos lógicos dentro do componente reconfigurável [15, 21].

2.1.1 Acoplamento

O acoplamento de uma arquitetura reconfigurável refere-se à forma de conexão entre o processador hospedeiro e o componente reconfigurável. Em geral, quanto maior o grau de acoplamento entre o processador hospedeiro e o componente reconfigurável, menor é o *overhead* de comunicação entre eles, logo o componente reconfigurável pode ser utilizado

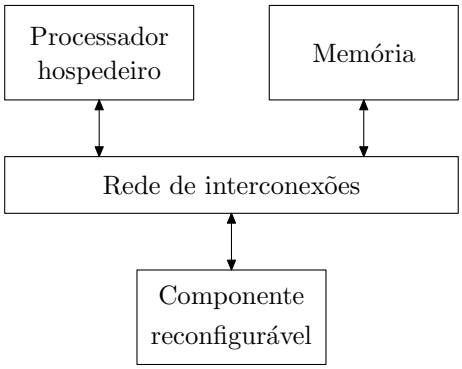


Figura 2.1: Modelo abstrato de arquitetura reconfigurável

mais freqüentemente dentro da aplicação. Por outro lado, componentes mais fracamente acoplados geralmente possuem uma maior capacidade computacional (e por conseguinte uma maior área), operando por mais tempo sem intervenção do processador, e permitindo assim uma maior exploração de paralelismo [6, 21].

Em sistemas fortemente acoplados, o componente reconfigurável é uma unidade funcional reconfigurável (RFU – *Reconfigurable Functional Unit*), integrada ao *datapath* do processador hospedeiro da forma semelhante às demais unidades funcionais do mesmo. A RFU é capaz de executar operações específicas de uma aplicação ou domínio e usa o conjunto de registradores do processador para obter operandos e armazenar resultados. Assim, tem-se o ambiente de programação tradicional do processador, acrescido de instruções especializadas que podem ser modificadas dinamicamente. PRISC [65] e Chimaera [87] são exemplos de arquiteturas com RFUs.

O componente reconfigurável pode ser utilizado como um coprocessador do processador hospedeiro, em um sistema de médio acoplamento, formando uma arquitetura híbrida. O coprocessador é capaz de executar por vários ciclos sem a intervenção do processador (e possivelmente concorrentemente com ele). O processador inicializa o coprocessador, enviando os dados necessários, o coprocessador realiza a computação independentemente do processador e retorna os resultados. Um exemplo desta abordagem é a arquitetura Garp [13].

Um sistema de médio acoplamento pode também usar o componente reconfigurável como mais uma unidade de processamento de um multiprocessador. A forma de arquitetura reconfigurável mais fracamente acoplada é aquela em que o componente reconfigurável é uma unidade de processamento *standalone*, que se comunica com o processador hospedeiro (se houver) através de uma rede de comunicação.

2.1.2 Granularidade do Bloco Lógico

A maioria das arquiteturas reconfiguráveis é baseada em um conjunto de blocos lógicos que são repetidos de maneira a formar uma estrutura maior. A granularidade do bloco lógico refere-se ao tamanho e à complexidade deste bloco. A granularidade mais fina provê uma maior flexibilidade em adaptar o hardware à computação desejada, porém traz um maior *overhead* de reconfiguração [6, 21].

Componentes reconfiguráveis de granularidade fina possuem blocos lógicos capazes de implementar funções com duas ou três entradas de 1 bit. Estes componentes são úteis para manipular dados no nível de bits e construir operadores e *datapaths* que operam com dados de largura diferente dos padrões usuais. Muitas arquiteturas reconfiguráveis são baseadas em FPGAs (*Field Programmable Gate Arrays*) que são dispositivos de lógica programável de granularidade muito fina, consistindo de uma matriz de blocos lógicos muito simples e regulares cuja funcionalidade é determinada pela configuração. Estes blocos são conectados por uma rede de interconexões também programável. Várias FPGAs comerciais estão disponíveis no mercado como por exemplo a família Virtex da Xilinx [86].

Arquiteturas reconfiguráveis de granularidade média possuem blocos lógicos que formam *bit-slices* de unidades funcionais (por exemplo, um somador de 4 bits) e são usados para implementar unidades funcionais que usam palavras mais largas. Estas arquiteturas podem ser usadas para implementar *datapaths* que operam com dados de diferentes larguras, porém de forma mais simples e eficiente que as arquiteturas de granularidade fina. Arquiteturas com *datapaths* reconfiguráveis possuem em geral blocos lógicos de granularidade mais grossa, compostos por unidades funcionais (como ALUs – *Arithmetic Logic Units*), que podem ser combinados e configurados para obter um *datapath* adequado a uma aplicação ou conjunto de aplicações. Exemplos deste tipo de arquitetura são os projetos RaPiD [24] e PipeRench [68].

Componentes reconfiguráveis de granularidade grossa possuem diversas unidades funcionais completas, que podem ser interconectadas de forma a implementar *datapaths* da largura das unidades funcionais. Estas arquiteturas podem ser muito mais eficientes em desempenho e em área que as arquiteturas de granularidade fina, quando se pretende implementar funções próximas à largura da palavra de seus blocos lógicos.

Há ainda arquiteturas reconfiguráveis de granularidade muito grossa em que os blocos lógicos são processadores simples, ligados entre si por uma rede de interconexões e

possivelmente gerenciados por um processador hospedeiro.

2.1.3 Rede de Interconexões

Os blocos lógicos do componente reconfigurável são interligados por uma rede de interconexões que também pode ser programável. A reconfigurabilidade das interconexões permite a implementação de diversas topologias entre blocos lógicos. A rede de interconexões contribui significativamente para a área total do componente, e quanto mais fina for a granularidade do bloco lógico (e em conseqüência maior a quantidade deles), mais complexa será a rede.

Em geral, a rede de interconexões impõe uma determinada organização dos blocos lógicos, como por exemplo, sob a forma de uma matriz bidimensional ou de um vetor linear. As redes bidimensionais são mais flexíveis, permitindo a implementação de uma grande variedade de estruturas, porém possuem um maior *overhead* de reconfiguração [21].

2.2 Exemplos de Arquiteturas Reconfiguráveis

Várias arquiteturas reconfiguráveis têm sido desenvolvidas em projetos acadêmicos e industriais. Nesta seção algumas destas arquiteturas são descritas.

2.2.1 Chimaera

A arquitetura Chimaera é organizada na forma de um processador hospedeiro com uma RFU fortemente acoplada ao seu *datapath*. A RFU tem acesso ao conjunto de registradores do processador através de um *shadow register file*, podendo ler até 9 registradores por ciclo do *clock*. Isto permite o mapeamento em hardware de seqüências de instruções executadas originalmente por software, formando operações com até 9 operandos de entrada e um resultado de saída [43, 87].

A RFU é estruturada como uma matriz de 9 linhas e 32 colunas de blocos lógicos com granularidade fina, onde cada bloco pode ser configurado como uma LUT (*Look-Up Table*) de 4 bits ou duas LUTs de 3 bits, ou ainda uma LUT de 3 bits e lógica de *carry*. O mecanismo de reconfiguração parcial permite que apenas algumas linhas da matriz sejam configuradas. Além disso, uma memória *cache* é utilizada para reduzir o tempo de reconfiguração.

Um compilador C foi desenvolvido a partir do GCC para mapear automaticamente grupos de instruções na RFU. Estes grupos variam de simples pares *add*/*sub* e *shift* até mais de 10 instruções incluindo desvios [88].

Os resultados mostram um ganho médio de desempenho de 21%, porém estes experimentos foram realizados apenas com o uso de simuladores. Além disso, a hipótese de que a RFU pode fazer até nove leituras e uma escrita de registradores a cada ciclo do *clock* é muito forte, dado que isto provavelmente resultaria no aumento do ciclo da máquina devido ao aumento no tempo de acesso do banco de registradores.

2.2.2 Garp

O projeto Garp possui uma arquitetura híbrida de acoplamento médio que engloba um processador hospedeiro MIPS e um componente reconfigurável sob a forma de um coprocessador, que tem acesso à hierarquia de memórias do sistema. O conjunto de instruções do processador foi estendido para incluir instruções capazes de configurar e controlar o coprocessador [13, 45].

O componente reconfigurável possui granularidade bem fina e é organizado como uma matriz de pelo menos 32 linhas e 24 colunas de blocos lógicos. Cada bloco é baseado em LUTs e opera dados de 2 bits. Desta forma, cada linha pode implementar uma operação de 32 bits e a matriz forma um *datapath* reconfigurável. O sistema permite a reconfiguração parcial, onde apenas algumas linhas da matriz são configuradas, e uma memória *cache* distribuída é utilizada para reduzir o *overhead* de reconfiguração.

Um compilador C foi desenvolvido para gerar o código que executa no processador MIPS e as configurações para o coprocessador, a partir da aplicação. O programador não precisa fornecer nenhuma informação sobre o particionamento hardware/software. Usando *profiling* o compilador seleciona laços que consomem grande parte do tempo de execução e forma hiperblocos [56], juntando blocos básicos [1] e usando predicação [2], ao longo do fluxo de controle com maior freqüência de execução dentro do laço. A computação representada por estes hiperblocos é mapeada no coprocessador, enquanto que os demais blocos básicos do laço são executados em software [12]. O compilador também explora a técnica de *pipelining* dos laços [14], onde a execução de diferentes iterações do laço são sobrepostas, com um intervalo de iniciação entre elas definido pelas dependências

[1] Um bloco básico é uma seqüência maximal de instruções, tal que sua execução é sempre iniciada na primeira instrução e terminada na última [61].

loop-carried [61] e pelos conflitos por uso de recursos. Para permitir a execução eficiente dos laços no coprocessador, a arquitetura aceita *loads* especulativos e possui três "filas de memória", que são estruturas para acesso de dados seqüenciais na memória.

Os resultados experimentais do estudo desta arquitetura [13] foram obtidos através de simulação e comparados com o processador MIPS básico e com um processador superescalar UltraSparc. Inicialmente, duas aplicações foram investigadas e, em cada uma delas, diversos laços foram mapeados, automaticamente no coprocessador pelo compilador. Na primeira aplicação, os laços tiveram *speedups* entre 2,1 e 12,7, enquanto que o *speedup* global da aplicação foi 2,9, em relação ao processador MIPS. Na segunda aplicação, embora para cada laço isoladamente tenha havido ganho de desempenho, para a aplicação como um todo não houve ganho, devido a limitações no paralelismo da aplicação e ao fato de que o uso do coprocessador interferiu nas otimizações globais do restante da aplicação. Em seguida, realizou-se o mapeamento manual no coprocessador de seis funções de uma biblioteca de um domínio específico e supôs-se que a memória *cache* de configurações já continha a configuração necessária. Foram obtidos *speedups* entre 1,84 e 43, em relação ao processador UltraSparc. Dos casos em que o *speedup* foi baixo, alguns tinham um volume de dados pequeno, fazendo com que o *overhead* de configuração fosse mais significativo. Outros eram funções que faziam acessos não seqüenciais à memória, impedindo o uso das filas de memória.

2.2.3 MorphoSys

O sistema MorphoSys integra em um mesmo chip um processador hospedeiro de propósito geral (RISC), um componente reconfigurável e uma interface para memória, possuindo assim um grau de acoplamento médio. O processador é responsável por iniciar todas as transferências da/para memória, envolvendo dados da aplicação e da configuração. O seu conjunto de instruções foi estendido para incluir instruções para controlar acessos à memória e configurar e controlar o componente reconfigurável [53, 72].

O componente reconfigurável é uma matriz de 8×8 blocos lógicos de granularidade grossa, onde cada bloco possui uma ALU/multiplicador, um deslocador e um conjunto de registradores que operam com dados de 16 bits. O sistema permite a reconfiguração dinâmica: as configurações podem ser carregadas da memória principal para uma área não ativa de uma memória de contextos, sem interromper a execução da matriz. Todos os blocos lógicos de uma mesma linha ou de uma mesma coluna recebem a mesma

configuração, porém operam sobre dados diferentes, implementando assim um modelo de execução SIMD (*Single Instruction Stream, Multiple Data Streams*).

A linguagem SA-C, baseada em *single assignments* [61], é usada para descrever as aplicações e um compilador realiza o particionamento da aplicação entre o processador RISC e a matriz reconfigurável, selecionando laços para execução em hardware. O compilador realiza o escalonamento e a alocação destes laços no componente reconfigurável e gera as configurações e o código executado no processador [83].

Experimentos foram realizados através da simulação do sistema com modelos VHDL e C++, e os tempos de execução foram estimados para um grupo de quatro funções de processamento de imagens. Os *speedups* obtidos, em relação a um processador Pentium III, variam de 3 a 42, porém todos os tempos de execução (do MorphoSys e do Pentium III) ficaram abaixo de 0,2s.

2.2.4 PipeRench

O projeto PipeRench possui um componente reconfigurável que implementa um *pipeline* virtual, anexado a um processador hospedeiro. O componente reconfigurável possui um conjunto de estágios do *pipeline* físico, denominados *stripes*, que podem ser reconfigurados dinamicamente em separado. Assim, embora o número de estágios físicos seja limitado, o *pipeline* virtual implementado pode ter um número maior de estágios. Enquanto um estágio físico é reconfigurado, os demais executam normalmente [35, 36].

Os estágios físicos são conectados de maneira a formar um vetor linear. Cada estágio físico possui uma rede de interconexões e um conjunto de elementos de processamento, que são compostos por uma ALU e um conjunto de registradores. A ALU é formada por um determinado número de LUTs e lógica adicional.

As aplicações são descritas na linguagem DIL (*Dataflow Intermediate Language*), baseada em *single assignments*, que pode ser vista como uma HDL de mais alto nível. O compilador otimiza a aplicação, aplicando *function inlining* a todos os módulos e desenrola completamente todos os laços [61], gerando um programa com o fluxo de execução totalmente seqüencial. O hardware é então sintetizado e a configuração para o PipeRench gerada [10].

A arquitetura PipeRench foi implementada em um chip com um processo de 0,18 micron, resultando em um total de 3,65 milhões de transistores operando na freqüência de 120MHz. Para uma determinada aplicação de criptografia, PipeRench obteve um

speedup de 5,96 em relação a um Pentium III [68].

2.2.5 RaPiD

A arquitetura RaPiD (*Reconfigurable Pipelined Datapath*) combina um processador hospedeiro e um componente reconfigurável de granularidade grossa, usado para implementar *datapaths* que mapeiam laços aninhados da aplicação. Diferentes implementações da arquitetura variam de acordo com diversos parâmetros como largura dos dados, número e tipos de unidades funcionais e número e organização dos barramentos [24, 31].

O componente reconfigurável é organizado na forma de um vetor linear de 8 a 32 blocos lógicos idênticos. Um bloco típico possui um multiplicador, três ALUs, seis registradores de propósito geral e três pequenas memórias locais. Estas unidades funcionais são interligadas através de um conjunto de barramentos segmentados que seguem ao longo do vetor. Geradores de endereços permitem o acesso eficiente a seqüências de dados na memória.

Os sinais que controlam a estrutura e operação do *datapath* reconfigurável são divididos em configuração estática, armazenada em células de memória, e controle dinâmico, atualizado a cada ciclo do *clock*. A configuração estática é utilizada para construir a estrutura desejada para o *datapath*, para uma aplicação específica ou para uma determinada parte da aplicação. O controle dinâmico é utilizado para escalonar as operações a cada ciclo. Esta divisão faz com que o *overhead* da reconfiguração dinâmica seja reduzido.

As aplicações são descritas na linguagem RaPiD-C, na qual o paralelismo da aplicação é exposto pelo programador. O compilador extrai da aplicação a organização estática do *datapath*, identifica os sinais de controle dinâmico e gera o código e os endereços das seqüências de dados a serem acessadas [23]. O desempenho do sistema é estimado através de modelos analíticos e não é comparado com outras arquiteturas.

2.2.6 Resumo

A Tabela 2.1 resume as principais características das arquiteturas reconfiguráveis descritas, assim como a sua classificação, de acordo com os critérios apresentados na Seção 2.1.

Projeto	Chimaera	Garp	MorphoSys	PipeRench	RaPiD
Arquitetura	processador + RFU	MIPS + coprocessador	RISC + componente	pipeline acoplado	coprocessador
Acoplamento	forte	médio	médio	médio	médio
Granularidade	fina	fina	grossa	grossa	grossa
Rede de Interconexões	matriz bidimensional	matriz bidimensional	matriz bidimensional	vetor linear	vetor linear
Reconfiguração	parcial / cache	parcial / cache	parcial	parcial (estágios)	estática / controle dinâmico
Operação mapeada	grupo de 2 a 10 instruções	laço e fluxo de controle	laço	aplicação	laços aninhados
Linguagem	C	C	SA-C	DIL	RaPiD-C
Implementação	simulação	simulação	simulação	chip	—
Resultados	ganho de desempenho médio: 21%	*speedup*: 2,9 (1 aplicação) 1,8 a 43 (manual)	*speedup*: 3 a 42	*speedup*: 5,96 (1 aplicação)	não comparado (estimativa)

Tabela 2.1: Características de algumas arquiteturas reconfiguráveis

2.3 Desafios da Computação Reconfigurável

Apesar da existência de vários projetos de arquiteturas reconfiguráveis bem sucedidos, ainda há diversas questões a serem resolvidas e/ou melhoradas para que a Computação reconfigurável possa ser utilizada de forma sistemática em domínios de aplicação mais amplos. Estas questões, detalhadas a seguir, delineiam problemas relevantes de pesquisa nesta área.

2.3.1 Reconfiguração Dinâmica

Como sistemas com reconfiguração dinâmica demandam mudanças na arquitetura durante a execução da aplicação, o tempo total de execução de uma aplicação engloba não somente o tempo gasto com a computação, mas também o tempo gasto com as reconfigurações. Assim, a reconfiguração deve ser realizada da forma mais rápida e eficiente possível, garantindo que o *overhead* da reconfiguração não cancele o ganho de desempenho obtido com a especialização do hardware. Diversas abordagens de reconfiguração rápida têm sido avaliadas, incluindo mecanismos em hardware e software [21].

Dispositivos reconfiguráveis com múltiplos contextos armazenam várias configurações (ativadas em diferentes momentos) e permitem uma rápida troca de contexto, reduzindo o tempo de reconfiguração [26]. Dispositivos parcialmente reconfiguráveis podem ser programados seletivamente, permitindo assim uma redução no tempo de reconfiguração [11, 55]. A busca antecipada da configuração é usada para sobrepor a reconfiguração do componente com a execução de instruções no processador hospedeiro, evitando assim que o processador fique bloqueado esperando a reconfiguração [40]. A compressão das configurações é usada para reduzir o seu tamanho (em bits), e em conseqüência o tempo necessário de sua transferência para cada bloco lógico [44]. Além disso, memórias *cache* podem ser usadas para armazenar as configurações, acelerando assim a transferência dos dados durante a reconfiguração [54]. Também têm sido estudadas técnicas para limitar a quantidade de informação que precisa ser modificada durante as reconfigurações, através por exemplo da identificação de similaridades entre configurações [71].

2.3.2 Mapeamento da Aplicação

Para que computação reconfigurável seja mais amplamente adotada como um modelo de Computação e aplicada a uma maior gama de domínios, é necessário o desenvolvimento de ferramentas para mapear as aplicações no hardware reconfigurável. Estas ferramentas devem ser capazes de, a partir de descrição da aplicação em alguma linguagem de programação de alto nível (como por exemplo C e C++), sintetizar configurações de hardware, permitindo ganhos de desempenho sem que o projetista precise manipular complexas configurações lógicas de baixo nível [42].

Para isso, estas ferramentas precisam combinar técnicas de compilação e de síntese de alto nível (*High-Level Synthesis* – HLS). É necessário aplicar otimizações baseadas em transformações de laços (como *loop unrolling*, etc), análise de fluxo de dados (como *operator strength reduction*, eliminação de código morto, eliminação de sub-expressões comuns, propagação de constantes, etc) e técnicas como *function inlining* e *tree-height reduction*, para se atingir um melhor desempenho [15].

Infelizmente, diversos projetos de computação reconfigurável ainda são baseados em ferramentas de HLS tradicionalmente usadas no projeto de ASICs. Tais ferramentas não são adequadas para arquiteturas reconfiguráveis, pois não exploram o potencial de reconfiguração, além de possuírem um tempo de compilação muito longo. Esta última característica é inaceitável no contexto de computação reconfigurável, pois inviabiliza a modificação dinâmica de configurações (por exemplo, especializar um bloco para um determinado valor de um dado de entrada, só conhecido durante a execução). Ademais, na geração de configurações para um componente reconfigurável, não há a liberdade de *layout* disponível nos ASICs [6].

Sintetizar hardware a partir de linguagens de programação de alto nível é mais difícil do que a partir de linguagens de descrição de hardware (HDLs – *Hardware Description Languages*), pois o hardware é inerentemente paralelo e as HDLs possuem um modelo de execução que facilmente expressa concorrência, enquanto que as linguagens de programação possuem, em geral, um modelo seqüencial de execução. Além disso, programas escritos em linguagens de programação trabalham com tipos de dados padrão (com determinadas larguras em número de bits), mesmo em situações em que poderiam ser usadas larguras menores, que resultariam na geração de hardware com menor custo, área, etc [42].

Alguns projetos de pesquisa utilizam ferramentas que transformam programas em

linguagens de alto nível ou em linguagem de montagem, em configurações de hardware, mas por outro lado restringem as construções da linguagem que podem ser usadas. Outros projetos propõem novas linguagens e impõem o seu uso ao programador.

Implementar toda uma aplicação em hardware só é viável para determinadas aplicações, pois em geral o número de operações em um programa típico pode resultar em uma grande área de hardware. Além disso, na maioria das aplicações há uma grande parte do código que é raramente executada. Pode-se então utilizar uma arquitetura híbrida, onde as partes executadas mais freqüentemente ou que consomem a maior parte do tempo de execução são implementadas em hardware, no componente reconfigurável, e as operações não implementáveis ou ineficientes em hardware ficam no software, executado pelo processador hospedeiro. O desafio desta estratégia é como selecionar quais partes da aplicação serão mapeadas em hardware, isto é, como fazer o particionamento hardware/software. Em geral, dados de *profiling* da aplicação são utilizados juntamente com a seleção manual por parte do projetista [42].

Assim, são necessárias ferramentas automatizadas de desenvolvimento, que permitam que o projetista analise o espaço de soluções e investigue as possibilidades de particionamento hardware/software, avaliando critérios como área, desempenho, etc [39, 81].

2.3.3 *Memory Wall*

Da mesma maneira que os GPPs, as arquiteturas reconfiguráveis também enfrentam o problema de descompasso entre a largura de banda da memória e a velocidade de processamento da unidade de execução, conhecido como *memory wall* [29, 42]. Arquiteturas reconfiguráveis reduzem os acessos à memória resultantes da busca de instruções, porém precisam de uma maior largura de banda para acesso aos dados, pois exploram mais paralelismo. Assim, o mecanismo de comunicação entre o processador hospedeiro (ou o sistema de memória) e o componente reconfigurável impõe limites no volume de informação que pode ser trocada entre eles, e em conseqüência no volume de computação que pode ser executada no componente reconfigurável.

Por exemplo, nas arquiteturas em que o componente reconfigurável é utilizado como um coprocessador, a taxa de transferência da comunicação entre o componente reconfigurável e o processador hospedeiro limita o ganho de desempenho que pode ser obtido. Nas arquiteturas em que o componente reconfigurável é uma unidade funcional

no *datapath* do processador hospedeiro, o número de operandos das instruções específicas é limitado em geral pelo número de portas do conjunto de registradores do processador hospedeiro, que, portanto, também restringe o paralelismo em potencial da aplicação.

2.3.4 Avaliação

Como mencionado, os sistemas reconfiguráveis propostos até hoje diferenciam-se bastante entre si, na sua arquitetura, no modelo de reconfiguração, no método de mapeamento da aplicação, etc. Existe portanto a necessidade de estabelecer critérios e definir *benchmarks* para a avaliação e comparação destes sistemas [50].

Capítulo 3

Modelo de Arquitetura Utilizado

Este capítulo apresenta a plataforma para o projeto de sistemas dedicados, que foi utilizada neste trabalho, baseada em uma arquitetura reconfigurável. Na Seção 3.1 a abordagem de especialização do conjunto de instruções é descrita. A Seção 3.2 apresenta o fluxo de projeto adotado, no qual uma das etapas é a alocação de recursos para um *datapath* reconfigurável, tema central deste trabalho. O modelo de arquitetura reconfigurável utilizado é apresentado na Seção 3.3. A Seção 3.4 apresenta uma avaliação preliminar da abordagem de especialização do conjunto de instruções. Por fim, a utilização do *datapath* reconfigurável é exemplificada na Seção 3.5.

3.1 Especialização do Conjunto de Instruções

GPPs freqüentemente não oferecem o desempenho e baixo custo exigidos por sistemas dedicados, pois o conjunto de instruções destes processadores é projetado para produzir um desempenho razoável em uma grande variedade de aplicações. Como um sistema dedicado usualmente é destinado a um domínio de aplicação restrito, uma melhor razão custo/desempenho pode ser obtida através da especialização do sistema para a(s) aplicação(ões) que ele precisa executar.

Os primeiros trabalhos sobre o projeto de processadores com conjunto de instruções específico para uma aplicação (ASIPs – *Application-Specific Instruction Set Processors*) envolvem a geração de um conjunto completo de instruções para um determinado domínio. Esta abordagem, utilizada em [46, 82], é todavia muito custosa e demanda um longo tempo de projeto.

Uma abordagem diferente é a geração de extensões do conjunto de instruções, onde um processador já disponível (por exemplo, um GPP) é utilizado e seu conjunto de instruções é estendido com instruções específicas para o domínio ou aplicação. O objetivo é otimizar o desempenho da aplicação, porém mantendo baixos o custo e o tempo de projeto.

O problema de determinar extensões especializadas do conjunto de instruções consiste em detectar na aplicação, clusters de operações primitivas que, quando implementadas em hardware, maximizem o desempenho da aplicação. Estes clusters são implementados como instruções especializadas, executadas em unidades funcionais específicas, adicionadas ao hardware do GPP.

Dada uma aplicação, o número de possíveis instruções especializadas cresce exponencialmente com o tamanho do programa, enquanto que o ganho de desempenho obtido com elas varia significantemente. Portanto, o processo de especialização deve ser o mais eficiente e automatizado possível.

Alguns trabalhos de extensão do conjunto de instruções são [3, 4, 8, 18, 19, 20, 37, 49, 51, 77]. Em vários deles, as operações que formam uma instrução especializada não podem ultrapassar os limites de blocos básicos, isto é, não podem conter construções de controle, tais como *"if"*, *"if-else"* e *"switch"* [20, 37, 49]. Outros trabalhos possuem um tempo de processamento exponencial [4] ou algumas etapas manuais [18, 51], isto é, não automatizadas. Tais fatores podem restringir a aplicabilidade das técnicas apresentadas e o desempenho alcançado por elas.

3.2 Fluxo do Projeto

Esta seção descreve uma plataforma experimental para projeto de ASIPs, apresentada em [7, 16] e utilizada neste trabalho. Os objetivos são: (a) encontrar, de forma automatizada e eficiente, extensões do conjunto de instruções de um processador base, dadas as aplicações a serem executadas; (b) projetar uma unidade funcional específica, sob a forma de um *datapath* reconfigurável, para executar as instruções especializadas. A plataforma permite a exploração do espaço de projeto, buscando o melhor desempenho para as aplicações e respeitando restrições arquiteturais. A Figura 3.1 mostra as principais etapas do fluxo do projeto utilizando esta plataforma.

Inicialmente, dado o código fonte da aplicação, ele é compilado para o processador base, isto é, considerando-se apenas o conjunto de instruções original. Utilizando uma carga de trabalho representativa, a aplicação é então executada em um modelo do processador base, obtendo-se seu desempenho e informações de *profile* (etapa 1 da Figura 3.1).

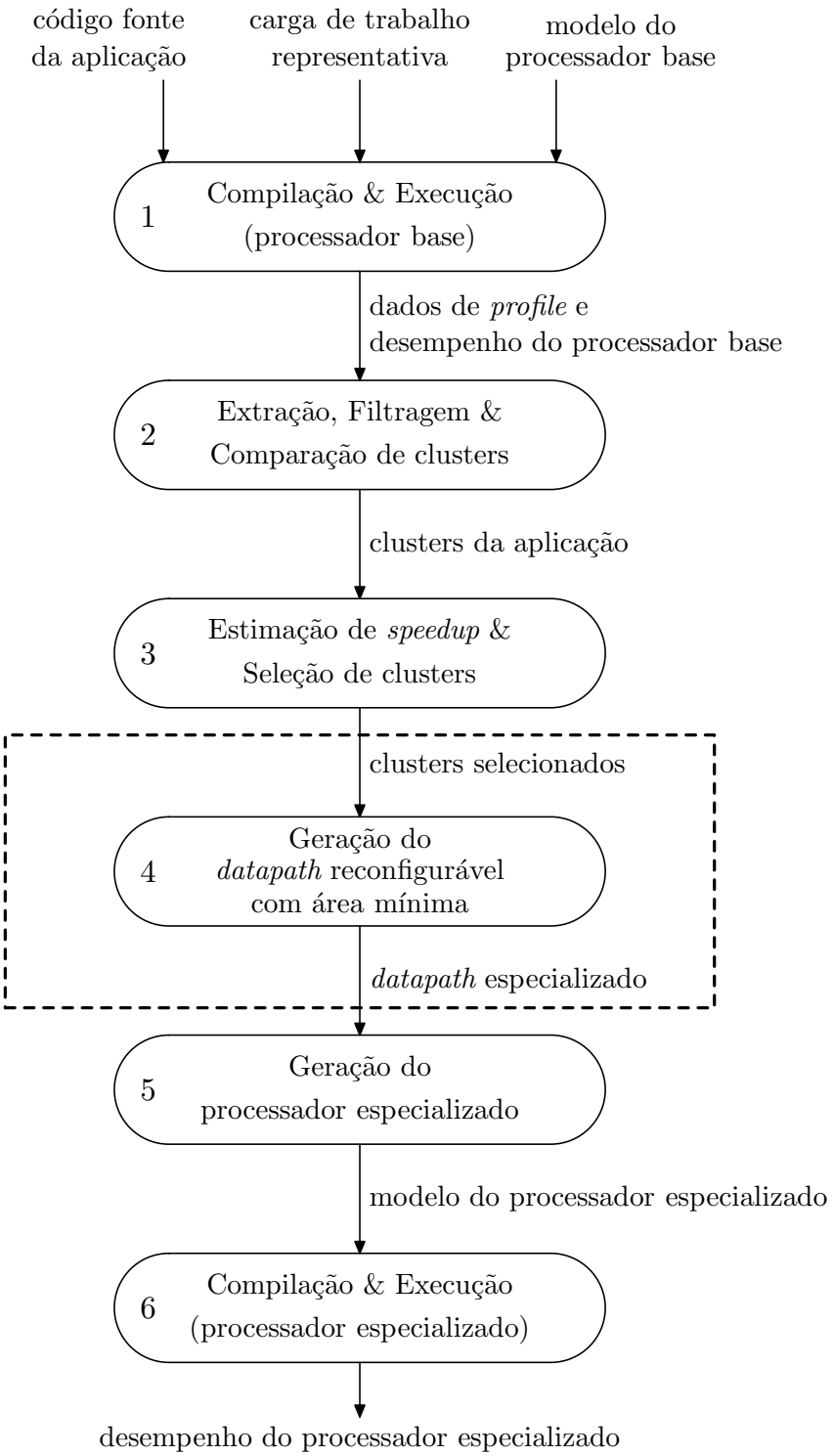


Figura 3.1: Passos do fluxo de projeto

Com as informações de *profile* é possível identificar as seções da aplicação que consomem a maior parte do tempo de execução (em geral são laços internos e funções). Todos os clusters válidos das seções identificadas são extraídos. O trecho de código da aplicação correspondente a um cluster não está limitado a um bloco básico e pode conter construções de controle. É possível também comparar estes clusters e identificar aqueles que aparecem repetidamente em diferentes partes da aplicação ou em diferentes aplicações (etapa 2 da Figura 3.1). Estes clusters são então representados como CDFGs.

Após identificar um conjunto de clusters, estima-se o *speedup* obtido ao substituí-los na aplicação por instruções especializadas. Com estas informações, selecionam-se aqueles clusters que proporcionam os maiores *speedups* (etapa 3 da Figura 3.1).

O cálculo da estimativa do *speedup* alcançado com um determinado cluster é realizado usando o número de ciclos consumido pelas instruções do cluster quando a aplicação foi executada no processador base (obtido dos dados de *profile*). Determina-se também o número de ciclos consumidos pela implementação em hardware do cluster, através do escalonamento de seu CDFG.

Um *datapath* especializado é gerado, de forma que possa ser reconfigurado dinamicamente para realizar a computação correspondente a cada cluster selecionado. Para isso, uma etapa de alocação de recursos é necessária (etapa 4 da Figura 3.1). Idealmente, este *datapath* deve possuir uma área mínima, considerando blocos lógicos e interconexões. A alocação de recursos determina o número e os tipos de blocos lógicos e interconexões a serem utilizados no *datapath* e como os clusters são mapeados nestes blocos lógicos e interconexões. Para projetar o *datapath* reconfigurável de área mínima é necessário compartilhar recursos do hardware entre os vários clusters. Esta etapa, destacada na figura, é o tema central deste trabalho.

Um modelo para o processador especializado é gerado, a partir dos modelos do processador base e do *datapath* especializado (etapa 5 da Figura 3.1). O processador especializado consiste do processador base com o *datapath* reconfigurável acoplado a ele. Diferentes níveis de abstração podem ser usados para modelar os processadores base e especializado. Por exemplo, os processadores podem ser modelados utilizando uma linguagem de descrição de arquiteturas (ADL – *Architecture Description Language*), permitindo a geração de simuladores dos mesmos. É possível ainda utilizar uma linguagem de descrição de hardware (HDL – *Hardware Description Language*) para desenvolver protótipos dos processadores, e sintetizá-los em uma FPGA.

Finalmente, dados os clusters selecionados e modelo do processador especializado,

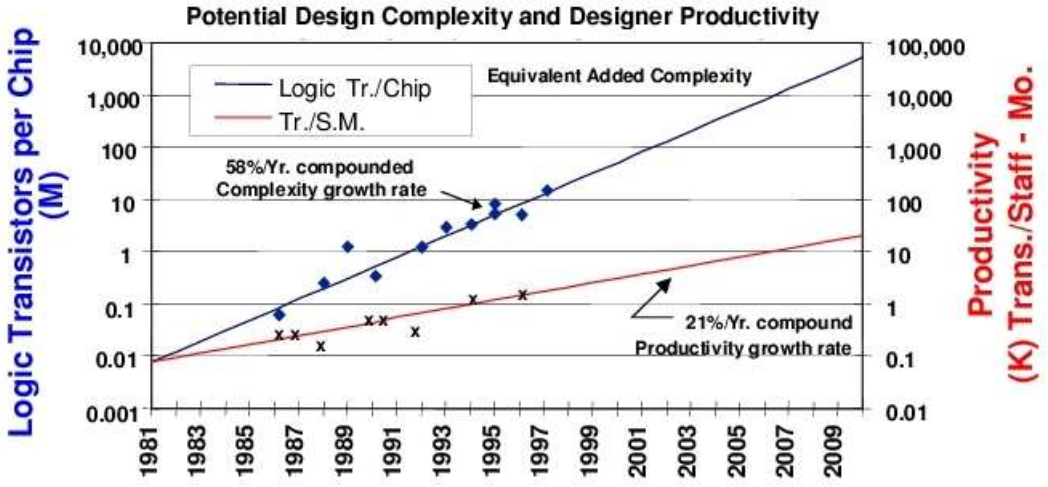


Figura 3.2: *Design productivity gap* [69]

a aplicação é compilada novamente, gerando código para o processador especializado. Neste processo, o compilador substitui as operações dos clusters pelas novas instruções especializadas, que serão executadas no *datapath* reconfigurável. A aplicação é executada novamente, porém agora no processador especializado, obtendo-se seu desempenho (etapa 6 da Figura 3.1). Com isto, calcula-se com maior precisão o *speedup* obtido com a implementação em hardware dos clusters selecionados.

3.3 Modelo de Arquitetura Reconfigurável

Com o advento de novas tecnologias de fabricação e integração, circuitos integrados com mais de 100 milhões de transistores já são fabricados. De fato, a Lei de Moore observa que a capacidade dos chips dobra a cada 18 meses [70]. Embora a previsão seja de que esta tendência não será mantida por muito tempo [70], já ocorre o que é denominado de *design productivity gap* (Figura 3.2): o crescimento da produtividade dos projetistas de circuitos não tem acompanhado o crescimento da capacidade dos circuitos, devido ao aumento da complexidade destes [69, 80]. Além disso, há cada vez mais imposições de se reduzir o tempo de desenvolvimento e os custos dos projetos de circuitos, devido à concorrência entre fabricantes.

Essa conjuntura requer que novas abordagens arquiteturais sejam investigadas, para permitir o projeto de arquiteturas reconfiguráveis com milhões de transistores em um chip, em um tempo aceitável e a um custo razoável [79]. Com a crescente capacidade

dos circuitos, componentes reconfiguráveis de granularidade fina (como FPGAs) trazem mais dificuldades em relação ao projeto e mais limitações de desempenho, devido principalmente à complexidade da rede de interconexões entre os blocos lógicos [41]. Arquiteturas reconfiguráveis de granularidade mais grossa representam uma alternativa viável, pois possuem uma organização mais modular e de mais alto nível [39].

Como conseqüência, o problema de alocação de recursos volta a ser estudado, porém em um contexto diferente daquele de HLS. No projeto de um *datapath* reconfigurável para uma arquitetura de granularidade grossa não há a liberdade de *layout* existente no projeto de ASICs ou mesmo quando FPGAs são utilizadas. A arquitetura pode já possuir uma determinada quantidade de cada tipo de recurso, e a alocação deve ser realizada respeitando estas restrições.

Além disso, é muito importante que os algoritmos de alocação de recursos sejam eficientes em termos de tempo de processamento, pois o longo tempo de síntese das ferramentas de HLS pode inviabilizar a busca de uma solução.

Por fim, com o crescimento da capacidade dos chips, a rede de interconexões entre os blocos lógicos torna-se mais complexa, contribuindo cada vez mais para a área total do componente reconfigurável. Portanto, é fundamental que as técnicas de alocação não utilizem apenas estimativas da área de interconexões.

3.3.1 Componente Reconfigurável

O modelo de arquitetura utilizado neste trabalho consiste de um componente reconfigurável de granularidade grossa que contém um conjunto de blocos lógicos (unidades funcionais – FUs – e registradores) interligados por uma rede de interconexão, formando um *datapath* programável, como mostrado na Figura 3.3. A rede de interconexão é baseada em um conjunto de multiplexadores (MUXes) que selecionam os operandos de entrada dos blocos lógicos. O *datapath* possui blocos lógicos fixos (*hardwired*) e apenas a rede de interconexão é programável. Cada bloco lógico é selecionado de uma biblioteca de componentes contendo desde unidades funcionais que realizam uma única operação, como somadores e multiplicadores, até unidades funcionais multi-função, como unidades lógico-aritméticas capazes de realizar soma, subtração e operações lógicas. Restrições de área e recursos podem ser impostas durante a construção do *datapath*.

À medida que a computação progride, o sistema reconfigura parcialmente o *datapath* (ajustando os MUXes da rede de interconexões), de forma que os clusters da aplicação

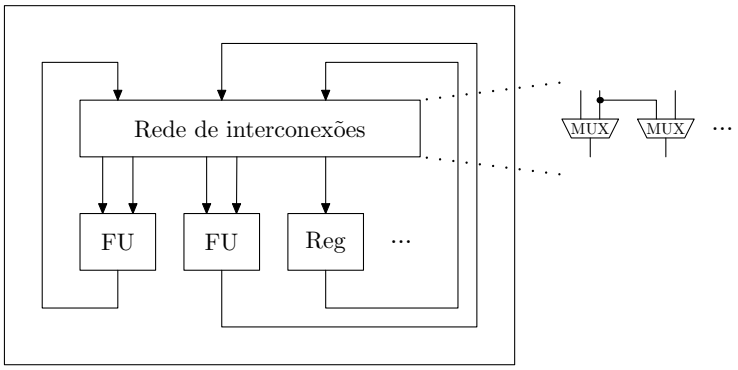


Figura 3.3: Modelo da arquitetura do componente reconfigurável

sejam mapeados nele. Dada a granularidade grossa dos blocos lógicos, menos bits são necessários para reconfigurar o *datapath* (em comparação com arquiteturas de granularidade fina), diminuindo portanto o que é denominado contexto de reconfiguração. Além disso, em geral, quanto menor o tamanho do contexto, menor o tempo necessário para reconfiguração. O tempo de reconfiguração é uma questão crítica em arquiteturas reconfiguráveis, dado que desempenho final é determinado pela soma dos tempos de computação e reconfiguração.

O componente reconfigurável pode assumir a forma de uma RFU integrada ao *datapath* do processador hospedeiro, como mostrado na Figura 3.4, formando um sistema fortemente acoplado. Neste caso, os operandos de uma instrução especializada executada na RFU são provenientes do conjunto de registradores do processador, para onde também são encaminhados os resultados produzidos. Portanto, o número máximo de registradores que a instrução especializada pode ler e escrever a cada ciclo do *clock* é determinado pela organização do *datapath* do processador hospedeiro e pelo número de portas de leitura e escrita do seu conjunto de registradores.

O fato da RFU estar integrada ao *datapath* do processador também pode impor limites às suas capacidade computacional e complexidade. Tal limitação, aliada à restrição acima em relação aos operandos de entrada e saída, faz com que, em geral, os clusters implementados em uma RFU possuam um número reduzido de operações. Como clusters menores consomem usualmente um tempo menor de execução, é necessário que o *overhead* de reconfiguração seja também reduzido, para haver ganho de desempenho em relação à implementação em software do cluster.

Assim, uma RFU é especialmente adequada para a implementação de clusters que representem operações simples, muito utilizadas pela aplicação, porém não presentes no

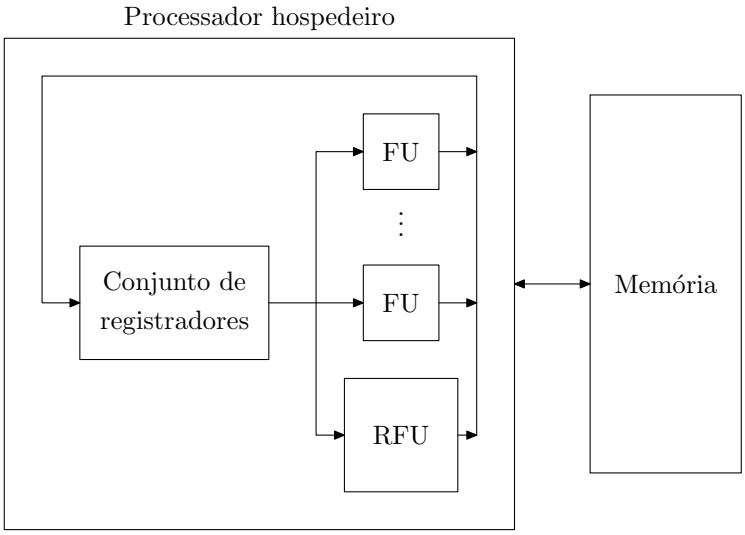


Figura 3.4: Componente reconfigurável como RFU

conjunto de instruções base do processador hospedeiro. O ganho de desempenho obtido com tais clusters deve-se principalmente à grande freqüência com que eles ocorrem na execução da aplicação.

Alternativamente, o componente reconfigurável pode ser implementado na forma de um coprocessador reconfigurável ligado ao processador hospedeiro, formando um sistema de acoplamento médio, mostrado na Figura 3.5. Neste caso, os operandos de entrada e saída de uma instrução especializada podem ser oriundos dos ou se destinar aos registradores do processador hospedeiro, respectivamente. O processador se encarrega de realizar as transferências de dados entre ele e o coprocessador. É também possível que o coprocessador tenha acesso à hierarquia de memórias sem necessitar a intervenção do processador, e assim obtenha seus operandos diretamente da memória.

Devido ao acoplamento mais fraco entre o processador hospedeiro e o componente reconfigurável, este pode ter uma maior capacidade computacional, e em conseqüência, implementar clusters maiores. Por outro lado, o fraco acoplamento aumenta o custo da comunicação entre processador e componente reconfigurável. Para haver ganho de desempenho, o *datapath* deve ser reconfigurado menos freqüentemente, ou seja, a execução do cluster deve consumir um tempo maior.

Portanto, um coprocessador reconfigurável é adequado para implementar clusters que englobam um grande número de operações, como por exemplo laços internos e funções. O ganho de desempenho obtido com tais clusters deve-se ao fato dos mesmos, quando

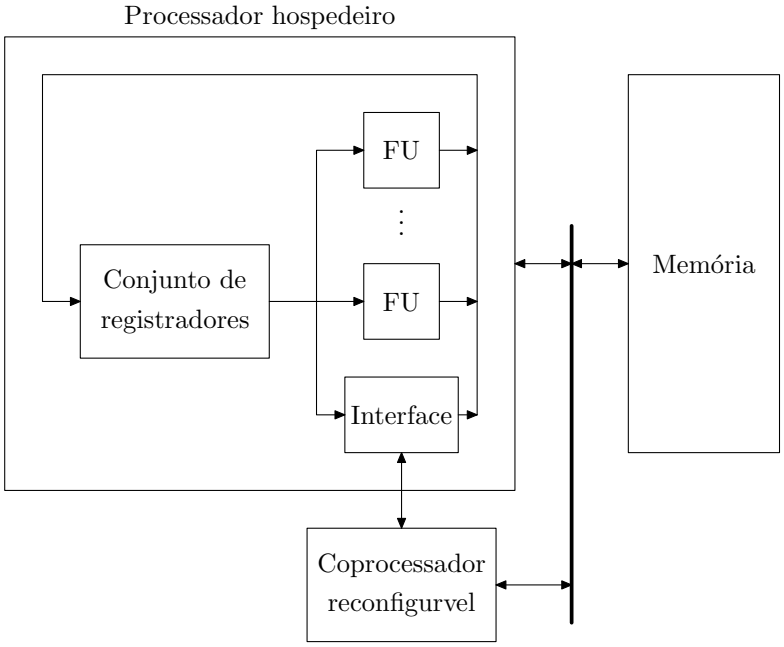


Figura 3.5: Componente reconfigurável como coprocessador

executados no coprocessador, consumirem bem menos ciclos que a sua implementação em software. Além disso, possivelmente o cluster é executado várias vezes na aplicação.

Em linhas gerais, as duas alternativas de implementação do componente reconfigurável, como uma RFU ou um coprocessador reconfigurável, representam um compromisso entre maior simplicidade do hardware (RFU) ou maior ganho de desempenho (coprocessador).

3.4 Avaliação da Especialização do Conjunto de Instruções

Antes de elaborar técnicas para compartilhamento de recursos no *datapath* reconfigurável, é importante avaliar se as aplicações realmente possuem clusters de operações que podem trazer ganhos de desempenho quando implementados em hardware.

3.4.1 Ganho de Desempenho com Cluster

Um exemplo de um cluster de operações simples pode ser obtido a partir da aplicação ADPCM, que implementa algoritmos de compressão e descompressão de voz, do *benchmark* MediaBench [52]. A Figura 3.6 mostra partes do código fonte de uma função com um único laço de cada um dos programas “ADPCM encoder” e “decoder”.

```
void adpcm_coder(...)
{
    ⋮
    for ( ; len > 0 ; len--)
    {
        ⋮
        if (valpred > 32767)
            valpred = 32767;
        else if (valpred < -32768)
            valpred = -32768;
        ⋮
        if (index < 0) index = 0;
        if (index > 88) index = 88;
        ⋮
    }
    ⋮
}
```

```
void adpcm_decoder(...)
{
    ⋮
    for ( ; len > 0 ; len--)
    {
        ⋮
        if (index < 0) index = 0;
        if (index > 88) index = 88;
        ⋮
        if (valpred > 32767)
            valpred = 32767;
        else if (valpred < -32768)
            valpred = -32768;
        ⋮
    }
    ⋮
}
```

Figura 3.6: Partes do código fonte dos programas “ADPCM encoder” e “decoder”

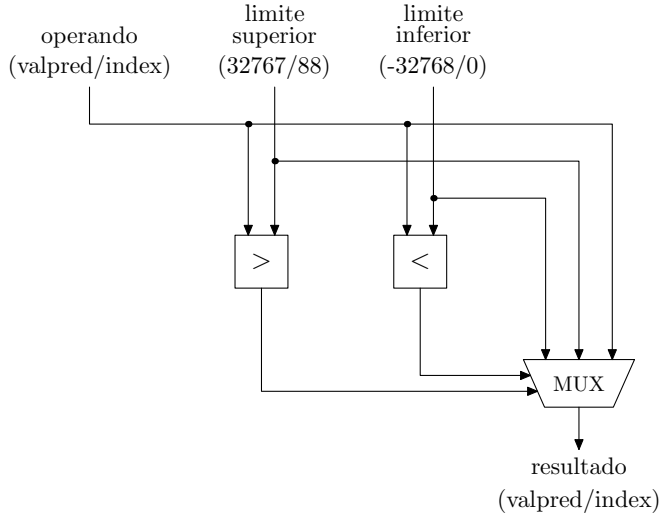


Figura 3.7: Unidade funcional especializada para clusters do ADPCM

Considerando o número de vezes que a função `adpcm_coder` é invocada e o número de repetições de cada execução do seu laço, o número total de vezes que o corpo deste laço é executado é linearmente proporcional ao tamanho (em bytes) da entrada de dados da aplicação. O mesmo ocorre com o laço da função `adpcm_decoder`. Por exemplo, para as entradas de dados fornecidas juntamente com o *benchmark*, o corpo de cada um dos laços é executado 147.520 vezes.

Na Figura 3.6 estão destacados dois trechos de código que ocorrem em ambos os programas. Estes trechos produzem dois clusters bastante similares que podem ser implementados como uma instrução específica executada em uma FU especializada. De forma simplificada, esta FU especializada pode ter a estrutura mostrada na Figura 3.7.

O cluster que manipula a variável `index`, quando executado em software em um processador hospedeiro SPARC v8 [75], consome exatamente 5 ciclos do *clock*, independentemente de o resultado das comparações ser verdadeiro ou falso. O outro cluster consome de 4 a 9 ciclos, dependendo do valor da variável `valpred`. A instrução específica pode ser implementada na FU especializada de forma a consumir apenas um ciclo. Com isso, a implementação em hardware dos clusters proporciona uma redução de 17,8% no número de ciclos gastos no laço. Este resultado mostra que mesmo clusters pequenos, porém com uma grande freqüência de execução, podem proporcionar ganhos de desempenho.

3.4.2 Repetição de Clusters

Um experimento apresentado em [16] utilizou a plataforma descrita na Seção 3.2 para procurar clusters de operações que ocorram repetidamente em diferentes aplicações e portanto, sejam candidatos à implementação como instruções especializadas. O modelo de hardware consistiu de um GPP (um processador SPARC v8) com unidades funcionais especializadas inseridas em seu *datapath*. Foram utilizadas 15 aplicações dos *benchmarks* MediaBench [52] e MiBench [38].

Para cada aplicação, um conjunto de clusters foi extraído. Inicialmente, estes clusters foram filtrados para descartar aqueles com menos de três operações. Tais clusters, por serem muito pequenos, proporcionariam pouco ou nenhum ganho de desempenho. Em seguida, os clusters foram comparados para detectar ocorrências repetidas de um mesmo cluster. Os clusters repetidos encontrados foram reunidos em grupos isomorfos. Alguns destes grupos apresentaram apenas pequenas diferenças, como por exemplo uma

Categoria	Número de aplicações	Número de grupos	Número de clusters
ZOL	15	11	765
Bit test	5	2	169
Or-Plus	7	3	16
Mul 6	4	1	59
Mem Copy	4	1	10
16to32	3	1	30
Mem Set-ZOL	3	1	7

Tabela 3.1: Exemplos de categorias de clusters [16]

comparação *"not equal to"* ao invés de *"equal to"*. Supondo que estes grupos de clusters poderiam se beneficiar da mesma implementação em hardware, eles foram reunidos em categorias.

Algumas das categorias encontradas que revelaram um potencial de ganho de desempenho com a implementação em hardware (na forma de instruções especializadas), são descritas em [16] e a seguir:

- *ZOL*: implementa o contador e o teste de término de um laço.
- *Bit test*: testa se um determinado bit de uma palavra é zero.
- *Or-Plus*: uma operação *or* com uma constante, seguida de uma soma.
- *16to32*: conversão de um dado de 16 bits para 32 bits.
- *Mul 6*: uma multiplicação por 6 usando deslocamentos e soma.
- *Mem Copy*: cópia de uma região da memória para outra.
- *Mem Set-ZOL*: equivalente à categoria *ZOL* acrescida da inicialização de uma região da memória.

A Tabela 3.1 mostra, para cada categoria descrita, o número de aplicações em que os clusters foram encontrados, o número de grupos isomorfos que pertencem àquela categoria e o número total de clusters encontrados, apresentados em [16].

Os resultados mostraram que vários clusters pequenos ocorrem com grande freqüência (em uma mesma aplicação e em várias aplicações). Tais clusters, se implementados em hardware gerariam FUs especializadas simples e consumiriam um número menor de ciclos do *clock*, em comparação com a implementação original em software. O ganho de desempenho da aplicação pode ser razoável, devido à freqüência de ocorrência dos clusters, principalmente quando eles estão dentro de laços.

3.4.3 *Speedup* de ASIPs

Um outro conjunto de experimentos, apresentado em [7], também utilizou a plataforma descrita na Seção 3.2 para estender o conjunto de instruções de um processador base, com a inclusão de instruções especializadas. O objetivo foi selecionar, para uma determinada aplicação, os clusters que maximizassem o desempenho da mesma, respeitando restrições arquiteturais. Neste caso, as instruções especializadas foram geradas a partir de clusters maiores, alguns inclusive correspondendo a todo o corpo de laços internos.

Da mesma forma, o modelo de hardware consistiu de um GPP (um processador SPARC v8) com uma FU especializada que implementou as instruções específicas. Esta FU foi inserida no *datapath* do processador e seus operandos e resultados eram lidos e escritos, respectivamente, no conjunto de registradores do processador base. Cada instrução especializada poderia consumir um único ciclo ou múltiplos ciclos do *clock*.

Ambos os processadores base e especializado foram simulados usando modelos desenvolvidos na ADL ArchC [66]. Tais modelos, por serem com precisão de ciclos, permitiram a medição do desempenho das aplicações.

Novamente, algumas aplicações dos *benchmarks* MediaBench [52] e MiBench [38] foram utilizadas, e calculou-se o *speedup* proporcionado por cada cluster extraído de cada aplicação, com diferentes restrições de recursos de hardware. Estas restrições limitaram os recursos disponíveis para a FU especializada (unidades aritméticas, unidades lógicas e de deslocamento, multiplicadores, MUXes, portas de leitura e escrita do conjunto de registradores e portas de leitura e escrita da memória). O número máximo de registradores (do processador base) que poderiam ser lidos e escritos a cada ciclo foi fixado em 2 e 1, respectivamente.

A Tabela 3.2 mostra os *speedups* obtidos em [7] com a inclusão de uma única instrução especializada correspondente ao melhor cluster, para cada aplicação. Se dois clusters de uma aplicação são totalmente disjuntos, ambos podem ser implementados como instruções especializadas, melhorando ainda mais o desempenho.

Os resultados mostraram que as aplicações possuem alguns clusters grandes que, quando implementados em hardware, permitem ganhos de desempenho consideráveis do processador especializado em relação ao processador base (de propósito geral).

Aplicação	*Speedup*
ADPCM decoder	3,60
ADPCM encoder	2,63
Rijndael decoder	1,88
Rijndael encoder	1,91
JPEG decoder	1,23
GSM decoder	1,66
Blowfish decoder	1,34

Tabela 3.2: *Speedup* obtido com melhor cluster para cada aplicação [7]

3.5 Exemplo de *Datapath* Reconfigurável

Nesta seção, um exemplo é utilizado para ilustrar o uso de um *datapath* reconfigurável para implementação de partes de uma aplicação. Suponha que esta aplicação possui dois laços internos, como mostrado na Figura 3.8, e que o corpo de cada laço será transformado em uma instrução especializada.

É necessário então gerar um *datapath* reconfigurável para executar o corpo dos laços, de forma multiplexada no tempo. Para isso, o corpo de cada laço é representado como um CDFG. Os CDFGs G_1 e G_2, mostrados na Figura 3.9, resultam dos laços 1 e 2 respectivamente. Estes CDFGs são combinados, de forma que blocos lógicos e interconexões sejam compartilhados. Assim, o *datapath* reconfigurável correspondente, também mostrado na Figura 3.9, é gerado. Conforme será apresentado na Seção 4.1, este *datapath* é ótimo, isto é, possui uma área total mínima.

A Figura 3.10(a) mostra a mesma aplicação da Figura 3.8, porém agora substituindo o corpo dos laços por instruções especializadas. A função `config_datapath` reconfigura o *datapath*, enquanto que a função `exec_datapath` transfere o controle para o componente reconfigurável que então executa alguma das instruções especializadas.

Para executar o corpo de um dos laços, o *datapath* precisa ser reconfigurado de forma a realizar a computação representada pelo CDFG correspondente ao laço. O sinal de controle *ctl* do *datapath* seleciona a operação a ser realizada pela FU somador/subtrador e também o operando esquerdo do deslocador. Atribuindo 0 ou 1 para *ctl*, o *datapath* é reconfigurado para executar o corpo do laço 1 ou 2, respectivamente, como mostrado na Figura 3.10. Nesta figura, as interconexões em negrito mostram o fluxo de dados na execução do corpo de cada laço.

```
⋮
for (i = 0; i < n; i ++)        /* Laço 1 */
{
   t1 = i * a;
   t2 = i + b;
   t3 = i + c;
   a = (t1 & t2) >> t3 ;
}
⋮
for (j = 0; j < n; j ++)        /* Laço 2 */
{
   t4 = j * d;
   t5 = j - e;
   d = t4 >> t5 ;
}
⋮
```

Figura 3.8: Código fonte de uma aplicação exemplo

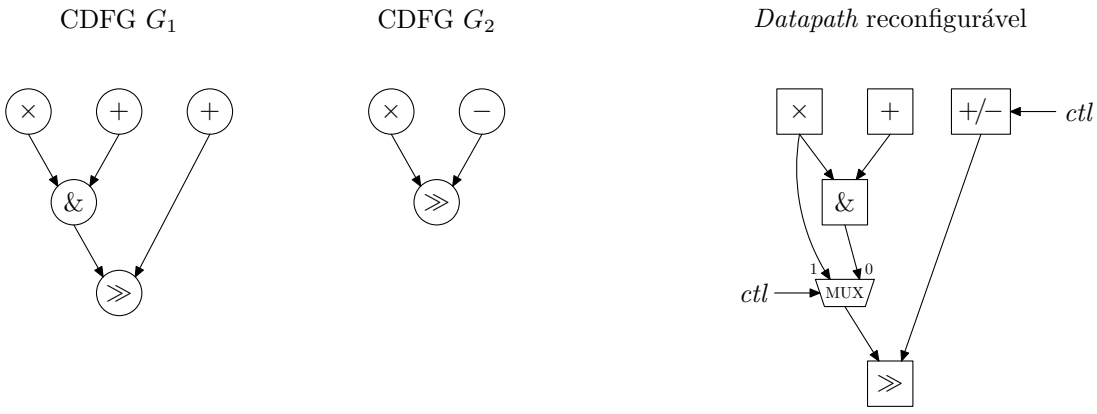
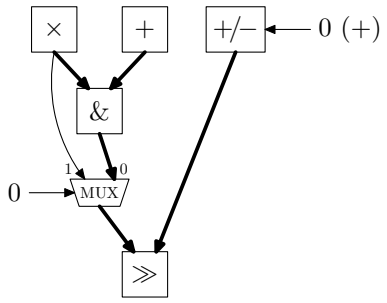


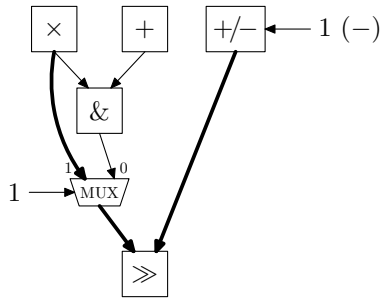
Figura 3.9: CDFGs correspondentes aos corpos dos laços da Figura 3.8 e *datapath* reconfigurável gerado

```
⋮
config_datapath(0);
for (i = 0; i < n; i ++)      /* Laço 1 */
{
   exec_datapath(...) ;
}
⋮
⋮
⋮
⋮
⋮
config_datapath(1);
for (j = 0; j < n; j ++)      /* Laço 2 */
{
   exec_datapath(...) ;
}
⋮
```

(a) Código da aplicação utilizando *datapath* reconfigurável



(b) Execução do corpo do laço 1



(c) Execução do corpo do laço 2

Figura 3.10: *Datapath* configurado para executar corpo dos laços modificados

Por exemplo, quando a função `config_datapath(0)` é executada, o valor 0 é atribuído ao sinal de controle *ctl* do *datapath*. Assim, quando a função `exec_datapath(...)` for executada, a FU somador/subtrador realizará soma e o MUX selecionará a sua entrada 0 para ser encaminhada para o deslocador, implementando assim o corpo do laço 1.

Como nessa aplicação há apenas uma instrução especializada dentro dos laços, é possível reconfigurar o *datapath* uma única vez antes do início de cada laço, ao invés de reconfigurá-lo a cada iteração do laço. Com isso, o *overhead* de configuração pode ser bastante reduzido.

Neste trabalho nós propomos algoritmos para o problema de compartilhamento de recursos na geração do *datapath* reconfigurável (Capítulos 5 e 6). Estes algoritmos combinam os CDFGs correspondentes aos clusters e buscam minimizar a área do *datapath*. Nos experimentos realizados para a avaliação de nossas técnicas, adotamos o modelo em que o componente reconfigurável é um coprocessador com blocos lógicos de granularidade

grossa e acoplado a um processador hospedeiro. O coprocessador tem acesso à memória principal e pode implementar clusters com um número razoável de operações. Porém, os algoritmos propostos não são limitados a este modelo e podem também ser aplicados para o modelo de RFU.

Capítulo 4

Compartilhamento de Recursos

Como mencionado nos capítulos anteriores, a computação reconfigurável permite o projeto de *datapaths* especializados que podem ser configurados dinamicamente para realizar diferentes computações, de acordo com a aplicação ou parte da aplicação que está sendo executada. Estas partes consistem de seqüências de instruções, módulos ou laços internos, selecionados por consumirem grande parte do tempo de execução. Durante a execução, quando cada parte selecionada inicia, o sistema reconfigura o *datapath* para realizar a computação correspondente.

O *datapath* reconfigurável deve possuir o menor número possível de blocos lógicos (unidades funcionais e registradores) e interconexões (multiplexadores e fios) para reduzir seu custo, área e consumo de potência. Deve também atender a requisitos de desempenho, como minimizar o tempo de execução ou garantir que um *throughput* mínimo estabelecido seja alcançado. Para reduzir a área e o custo do *datapath*, seus blocos lógicos e suas interconexões devem ser reutilizados ao máximo nas diferentes partes da(s) aplicação(ões). Este compartilhamento de recursos tem também um grande impacto na redução do *overhead* de reconfiguração do sistema, tanto em tempo quanto em espaço.

Cada cluster selecionado da aplicação é representado por um CDFG, e para projetar o *datapath* reconfigurável, esses CDFGs são combinados. O processo de combinação de CDFGs permite a reutilização de blocos lógicos e interconexões, identificando similaridades entre os CDFGs, e produzindo um *datapath* reconfigurável, que pode ser configurado dinamicamente para realizar a computação representada por cada CDFG. Idealmente, este *datapath* deve ter uma área mínima, considerando a área dos blocos lógicos e interconexões. Encontrar a solução para a combinação de CDFGs que minimiza a área dos blocos e interconexões é o tema central deste trabalho.

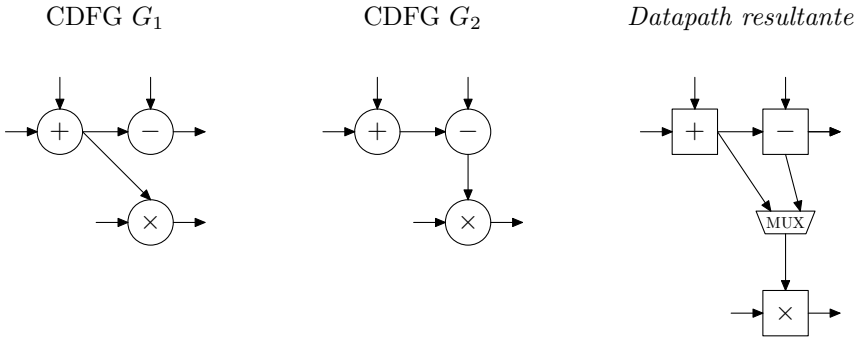


Figura 4.1: Combinação de CDFGs

A Figura 4.1 ilustra o conceito de combinação de CDFGs. Quando os CDFGs G_1 e G_2 são combinados, o *datapath* resultante é obtido. Note que neste *datapath* há interconexões originadas a partir de apenas um CDFG (por exemplo, a interconexão $(+,\times)$ do CDFG G_1) e interconexões compartilhadas por ambos os CDFGs (por exemplo, a interconexão $(+,-)$).

O problema de compartilhamento de recursos é bem conhecido na área de HLS [32, 25]. Neste contexto, um *datapath* é sintetizado a partir de uma especificação comportamental da aplicação, representada por um CDFG, otimizando um determinado objetivo (por exemplo, o custo do circuito). O processo de síntese consiste basicamente das tarefas de escalonamento e alocação, onde a alocação pode ser ainda dividida em duas sub-tarefas principais: seleção de unidades (*unit selection*) e atribuição de unidades (*unit binding*). A seleção de unidades consiste em determinar o número e os tipos de blocos lógicos (unidades funcionais, registradores, etc) e interconexões a serem utilizados no circuito. A atribuição de unidades envolve o mapeamento das operações, variáveis e transferências de dados do CDFG nos blocos lógicos e interconexões alocados [32].

No problema de combinação de CDFGs estudado neste trabalho, o *datapath* reconfigurável sintetizado deve ser capaz de realizar a computação de várias especificações comportamentais (representadas por vários CDFGs), de forma multiplexada no tempo, enquanto que em HLS o *datapath* resultante corresponde a apenas um CDFG de entrada. O problema de combinação de CDFGs é principalmente focado em prover compartilhamento de recursos entre CDFGs, enquanto que a alocação em HLS permite o compartilhamento de recursos intra-CDFG. Tradicionalmente, em HLS a alocação de blocos lógicos é realizada utilizando custos estimados das interconexões, e somente após este passo, quando os requisitos de interconexões tornam-se claros, a alocação das mesmas

é realizada. Esta não é uma boa estratégia para resolver o problema de combinação de CDFGs, pois a área das interconexões resultante da atribuição de interconexões é altamente dependente da atribuição dos blocos lógicos. Na combinação de CDFGs deve-se portanto mapear blocos lógicos e interconexões de forma conjunta.

A Seção 4.1 formaliza o problema de combinação de CDFGs para o projeto de um *datapath*, e uma prova de que este problema é NP-completo é apresentada na Seção 4.2. A Seção 4.3 descreve diversas soluções propostas para este problema, encontradas na literatura. Por fim, as soluções propostas para a combinação global e incremental de CDFGs são discutidas na Seção 4.4.

4.1 O Problema de Combinação de CDFGs

Nesta seção o problema de combinação de CDFGs é formulado formalmente. Deseja-se combinar vários CDFGs, correspondentes a diferentes clusters de uma aplicação (ou de diferentes aplicações), para construir um *datapath* reconfigurável, capaz de realizar a computação de cada cluster, de forma multiplexada no tempo. Além disso, a área dos blocos lógicos (unidades funcionais e registradores) e interconexões deste *datapath* deve ser mínima. Cada cluster i, $i = 1 \ldots n$ da aplicação é modelado como um CDFG G_i, definido a seguir.

Definição 1 *Um grafo de fluxo de dados e controle (Control/Data-Flow Graph – CDFG) é um grafo direcionado $G = (V, E)$, tal que:*

- *Um vértice $v \in V$ representa uma operação ou uma variável e possui um conjunto de portas de entrada $p = 1 \ldots n_v$ e atributos especificando seu tipo e largura (em bits).*
- *Um arco $e = (u, v, p) \in E$ indica uma transferência de dados do vértice u para a porta de entrada p do vértice v.*

Dado um vértice de um CDFG, podem existir na biblioteca de componentes (ou disponíveis na arquitetura) vários blocos lógicos onde ele pode ser executado.

Definição 2 *O conjunto de blocos lógicos $B(v)$ de um vértice v de um CDFG contém os blocos lógicos da biblioteca de componentes que podem realizar a computação representada por v.*

O *datapath* reconfigurável resultante é modelado como um grafo combinado $\bar{G}$, definido a seguir. O grafo combinado $\bar{G}$ pode ser visto como a sobreposição de todos os CDFGs G_i, $i = 1 \ldots n$, tal que somente vértices que podem ser implementados por um mesmo bloco lógico são sobrepostos.

Definição 3 *Um grafo combinado, correspondente aos CDFGs G_i, $i = 1 \ldots n$, é um grafo direcionado $\bar{G} = (\bar{V}, \bar{E})$, tal que:*

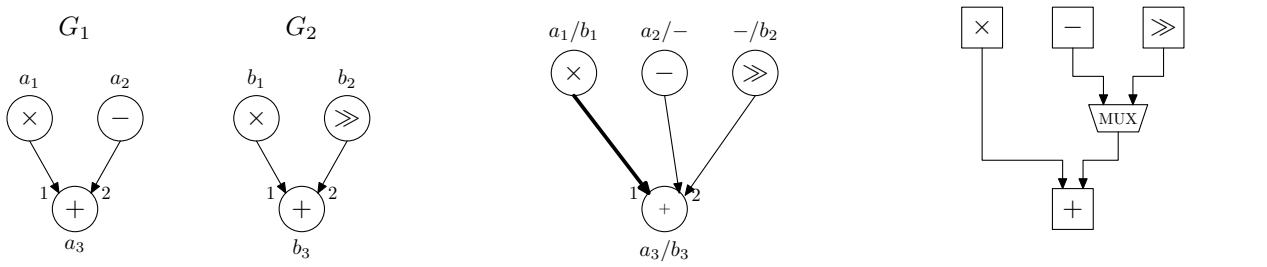
- *Um vértice $\bar{v} \in \bar{V}$ representa um mapeamento de $m_{\bar{v}}$ vértices v_i, $1 \leq m_{\bar{v}} \leq n$, cada v_i pertencendo a um V_i diferente, tal que $\bigcap_{\forall v_i \text{ mapeado em } \bar{v}} B(v_i) \neq \emptyset$.*
- *Um arco $\bar{e} = (\bar{u}, \bar{v}, \bar{p}) \in \bar{E}$ representa um mapeamento de $m_{\bar{e}}$ arcos $e_i = (u_i, v_i, p_i)$, $1 \leq m_{\bar{e}} \leq n$, cada e_i pertencendo a um E_i diferente, tal que todos os u_i foram mapeados em $\bar{u}$, todos os v_i foram mapeados em $\bar{v}$, e todas as portas de entrada p_i são equivalentes*[1]*.*

O *datapath* reconfigurável terá um bloco lógico correspondendo a cada vértice $\bar{v} \in \bar{V}$. Este bloco lógico é selecionado de forma que seja capaz de realizar a computação representada por cada vértice v_i mapeado em $\bar{v}$. Para cada arco $\bar{e} = (\bar{u}, \bar{v}, \bar{p}) \in \bar{E}$, haverá no *datapath* um caminho conectando os blocos lógicos correspondentes a $\bar{u}$ e $\bar{v}$, indo da saída do primeiro para a porta de entrada $\bar{p}$ do segundo. Além disso, para cada porta de entrada $\bar{p}$ de cada vértice $\bar{v}$ que possui mais de um arco incidente $(*, \bar{v}, \bar{p})$, o *datapath* reconfigurável terá um MUX selecionando o operando de entrada.

O mapeamento de vértices representado por $\bar{v} \in \bar{V}$ é descrito por uma n-tupla $v_1/v_2/\ldots/v_n$ enumerando os vértices v_i mapeados em $\bar{v}$. Se não há nenhum vértice de um dado V_i mapeado em $\bar{v}$, o elemento correspondente na n-tupla é vazio, como por exemplo em $-/v_2/\ldots/v_n$. Analogamente, o mapeamento de arcos representado por $\bar{e} \in \bar{E}$ é descrito por uma n-tupla $e_1/e_2/\ldots/e_n$ enumerando os arcos e_i mapeados em $\bar{e}$, com elementos vazios quando necessário.

Dado um conjunto de n CDFGs G_i, é possível construir vários grafos combinados $\bar{G}$ diferentes correspondendo a eles. A solução ótima para $\bar{G}$ é aquela que produz o *datapath* reconfigurável com área mínima, considerando os blocos lógicos e as interconexões. A área do *datapath* reconfigurável gerado a partir de um grafo combinado $\bar{G}$ é a soma da área dos blocos lógicos com a área das interconexões. A área dos blocos lógicos do *datapath*

[1] O significado de portas de entrada equivalentes será elaborado na Subseção 4.1.1.



(a) CDFGs G_1 e G_2 (b) Grafo combinado $\bar{G}$ (c) *Datapath* correspondente a $\bar{G}$

Figura 4.2: Mapeamento de arcos e número de entradas nos MUXes

é a soma da área de cada bloco lógico do *datapath*. Cada bloco lógico (correspondente a um vértice $\bar{v} \in \bar{V}$) é selecionado de acordo com os vértices v_i mapeados em $\bar{v}$.

Como a rede de interconexões é baseada em MUXes, a área das interconexões é proporcional ao número de entradas nos MUXes. Para cada arco $\bar{e} = (\bar{u}, \bar{v}, \bar{p}) \in \bar{E}$ que é um mapeamento de $m_{\bar{e}}$ arcos, o MUX (se existir) na porta de entrada $\bar{p}$ do bloco lógico $\bar{v}$ possui $m_{\bar{e}} - 1$ menos entradas do que ele teria se nenhum arco fosse sobreposto. Por exemplo, na Figura 4.2 os CDFGs G_1 e G_2 são combinados, obtendo-se o grafo $\bar{G}$. O vértice a_3/b_3 de $\bar{G}$ possui dois arcos incidindo em sua porta de entrada 2. Portanto, o *datapath* terá um MUX de duas entradas na porta direita do somador. Como o arco $(a_1/b_1, a_3/b_3)$ de $\bar{G}$ (em negrito) corresponde ao mapeamento de dois arcos de G_1 e G_2, respectivamente, o MUX na porta esquerda do somador precisaria de apenas uma entrada (ao invés de duas), sendo portanto eliminado.

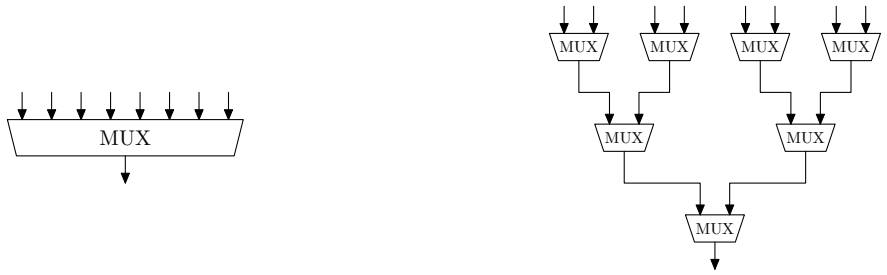
Modelando cada MUX como uma árvore de MUXes de duas entradas (como ilustrado na Figura 4.3), existe uma dependência linear entre o número de MUXes de duas entradas e o número de interconexões do *datapath*. Logo, a área das interconexões pode ser expressa em termos do número de interconexões (e portanto, em termos do número de arcos de $\bar{G}$). A área do *datapath* reconfigurável é definida a seguir.

Definição 4 *A área total $A(\bar{G})$ do datapath reconfigurável correspondente ao grafo combinado $\bar{G} = (\bar{V}, \bar{E})$ é:*

$$A(\bar{G}) = A_b(\bar{G}) + A_i(\bar{G})$$

onde $A_b(\bar{G}) = \sum_{\forall \bar{v} \in \bar{V}} A_b(\bar{v})$ e $A_i(\bar{G}) = |\bar{E}| \times A_{mux}$ são, respectivamente, as áreas dos blocos lógicos e das interconexões do datapath. $A_b(\bar{v})$ é a área do bloco lógico alocado para $\bar{v} \in \bar{V}$, e A_{mux} a área equivalente a uma entrada de MUX da largura adequada.

O problema de combinação de CDFGs é então definido a seguir.



(a) MUX de 8 entradas

(b) Árvore de MUXes de 2 entradas

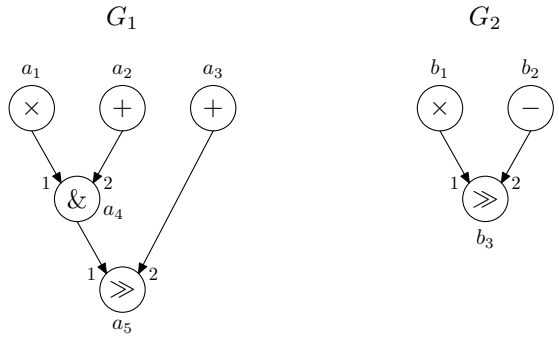
Figura 4.3: MUX de 8 entradas modelado como árvore de MUXes de 2 entradas

Definição 5 *Dados n CDFGs* $G_i = (V_i, E_i)$, $i = 1 \ldots n$, *e uma biblioteca de componentes, o problema de combinação de CDFGs consiste em encontrar um grafo combinado* $\bar{G} = (\bar{V}, \bar{E})$ *correspondente, tal que* $A(\bar{G})$ *seja mínimo.*

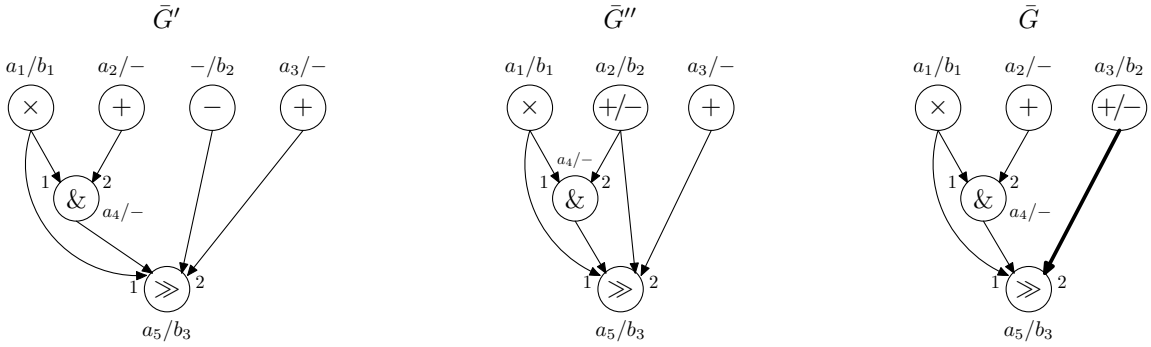
Encontrar um mapeamento de vértices dos CDFGs de forma a minimizar área dos blocos lógicos não é uma tarefa difícil. Ela pode ser modelada como um problema de emparelhamento de peso máximo em um grafo bipartido, para o qual um algoritmo exato com complexidade polinomial é conhecido. Por outro lado, mapear os arcos dos CDFGs de forma a minimizar a área das interconexões é um problema difícil (mais precisamente, é um problema NP-completo, como será visto na Seção 4.2) pois o mapeamento dos arcos depende do mapeamento dos vértices adjacentes a eles. Isto é, dois arcos de dois CDFGs só podem ser mapeados se seus vértices origem estão mapeados, assim como seus vértices destino. Logo, se o mapeamento de vértices é feito ignorando os custos de interconexões ou usando apenas estimativas, pode-se obter uma solução onde a área das interconexões não é mínima, e conseqüentemente a área total também não é ótima.

Esses conceitos são ilustrados com um exemplo. Dados os CDFGs G_1 e G_2 da Figura 4.4(a), obtidos a partir dos laços do exemplo da Seção 3.5, é possível construir três grafos combinados $\bar{G}'$, $\bar{G}''$ e $\bar{G}$ diferentes[2] (mostrados na Figura 4.4(b)). Em $\bar{G}'$, os vértices a_2 de G_1 e b_2 de G_2 não estão mapeados, então o *datapath* reconfigurável correspondente a $\bar{G}'$ terá seis blocos lógicos (um multiplicador, dois somadores, um subtrador, um *and* e um deslocador). Em $\bar{G}''$, estes vértices estão mapeados, resultando em um *datapath* com cinco blocos lógicos (um multiplicador, um somador/subtrador, um somador, um *and* e um deslocador). Como resultado, $A_b(\bar{G}')$ é maior que $A_b(\bar{G}'')$ pois apesar da área

[2]Por simplicidade, os multiplexadores que selecionam as entradas do deslocador em $\bar{G}'$, $\bar{G}''$ e $\bar{G}$ não são mostrados na figura.



(a) CDFGs G_1 e G_2



(b) Grafos combinados $\bar{G}'$ e $\bar{G}''$ e grafo combinado ótimo $\bar{G}$

Figura 4.4: Diferentes grafos combinados: $A_b(\bar{G}') > A_b(\bar{G}'') = A_b(\bar{G})$ e $A_i(\bar{G}'') > A_i(\bar{G})$

de um somador/subtrador ser maior que a área de apenas um somador, ela certamente é menor que as áreas de um somador e um subtrador adicionadas. Conseqüentemente, $A(\bar{G}')$ é também maior que $A(\bar{G}'')$, dado que $A_i(\bar{G}')$ é igual a $A_i(\bar{G}'')$ pois nenhum arco foi sobreposto em $\bar{G}'$ ou $\bar{G}''$.

Ainda na Figura 4.4, $\bar{G}''$ e $\bar{G}$ representam diferentes mapeamentos de vértices. Em $\bar{G}''$ o vértice b_2 de G_2 está mapeado no vértice a_2 de G_1, enquanto que em $\bar{G}$ ele está mapeado em a_3. Os mapeamentos de vértices representados por $\bar{G}''$ e $\bar{G}$ podem parecer equivalentes, pois ambos geram *datapaths* com os mesmos blocos lógicos. De fato, $A_b(\bar{G}'')$ é igual a $A_b(\bar{G})$, porém os mapeamentos de vértices de $\bar{G}''$ e $\bar{G}$ permitem diferentes mapeamentos de arcos. Em $\bar{G}''$, nenhum arco é sobreposto, logo dois MUXes são necessários nas duas portas de entrada do vértice a_5/b_3. Em $\bar{G}$, os arcos (a_3, a_5) e (b_2, b_3) estão mapeados (em negrito na figura), eliminando assim a necessidade de um dos MUXes. Como resultado, $A_i(\bar{G}'')$ é maior que $A_i(\bar{G})$ e, conseqüentemente, $A(\bar{G}'')$ também é maior que $A(\bar{G})$.

Para obter a solução ótima para $\bar{G}$ é necessário determinar qual o mapeamento de vértices, dentre as várias possibilidades, permite o melhor mapeamento de arcos, isto é, minimiza a área total.

4.1.1 Portas de Entrada e Comutatividade

Outro importante aspecto do problema de combinação de CDFGs diz respeito às portas de entrada e à propriedade de comutatividade. Diversas operações de dois operandos são comutativas, logo a troca apropriada dos operandos destas operações pode permitir mapeamentos de arcos que não seriam possíveis caso contrário. Assim, interconexões e MUXes são eliminados, reduzindo a área das interconexões do *datapath* reconfigurável.

Cada operação possui um conjunto de portas de entrada que representam os operandos que ela pressupõe. Por exemplo, uma adição possui duas portas de entrada 1 e 2. Pela Definição 3, dois arcos $(u_1, v_1, p_1) \in G_1$ e $(u_2, v_2, p_2) \in G_2$ podem ser mapeados se u_1 e u_2 podem ser mapeados, assim como v_1 e v_2, e se p_1 e p_2 são equivalentes. A equivalência de portas de entrada é definida a seguir.

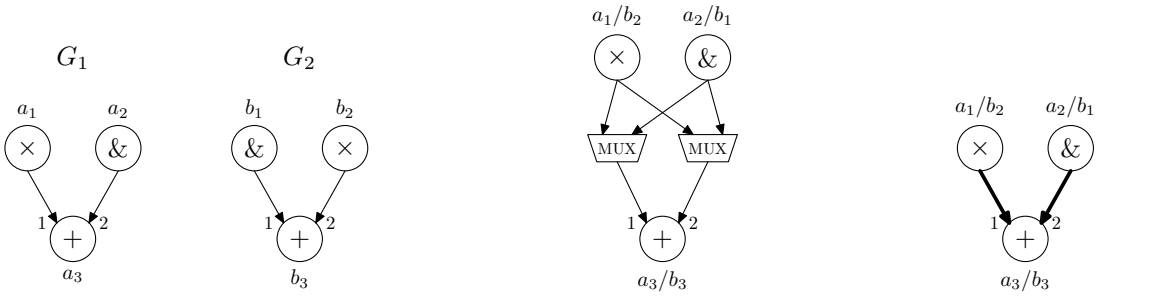
Definição 6 *Dados dois arcos (u_1, v_1, p_1) e (u_2, v_2, p_2) pertencentes aos CDFGs G_1 e G_2, respectivamente, as portas de entrada p_1 e p_2 são equivalentes se:*

- *$p_1 = p_2$; ou*
- *v_1 e/ou v_2 representam operações comutativas de dois operandos.*

Considere por exemplo, os CDFGs G_1 e G_2 da Figura 4.5, onde as portas de entrada 1 e 2 das operações de adição são mostradas. Ao combinar G_1 e G_2, há três possíveis mapeamentos de vértices (a_1/b_2, a_2/b_1 e a_3/b_3) e dois possíveis mapeamentos de arcos $(a_1, a_3, 1)/(b_2, b_3, 2)$ e $(a_2, a_3, 2)/(b_1, b_3, 1)$. Estes mapeamentos de arcos são obtidos apenas com a exploração da comutatividade da operação de adição, pois as portas de entrada dos arcos mapeados são diferentes. Como resultado, o grafo combinado $\bar{G}$ possui menos interconexões e dois MUXes a menos do que $\bar{G}'$, que é o grafo combinado obtido sem a comutação dos operandos.

4.2 Prova de NP-Completude

Nesta seção nós provamos que o problema de combinação de CDFGs é NP-completo [59]. Na demonstração, a versão de decisão do problema, definida a seguir, é usada.



(a) CDFGs G_1 e G_2

(b) Grafo combinado $\bar{G}'$ (sem comutação)

(c) Grafo combinado $\bar{G}$ (com comutação)

Figura 4.5: Compartilhamento de interconexões através da comutação de operandos

Definição 7 *Dados n CDFGs $G_i = (V_i, E_i)$, $i = 1 \ldots n$, um inteiro k e uma biblioteca de componentes com áreas inteiras, o problema de decisão de combinação de CDFGs (COMB) consiste em determinar se há um grafo combinado $\bar{G} = (\bar{V}, \bar{E})$ correspondente, tal que $A(\bar{G}) \leq k$.*

Para provar que COMB é NP-completo, é necessário mostrar que COMB pertence a NP e a NP-difícil. Nós provamos que COMB é NP-difícil através de uma redução a partir do problema de isomorfismo de subgrafos (aplicado a grafos direcionados), que é NP-completo [33, GT48]. O problema de isomorfismo de subgrafos é definido a seguir.

Definição 8 *Dados os grafos direcionados $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, o problema de isomorfismo de subgrafos (ISO) consiste em determinar se G_1 contém um subgrafo direcionado $H = (V_H, E_H)$ isomorfo a G_2, i.e., se existe uma função bijetiva $f : V_2 \rightarrow V_H$, tal que $(u, v) \in E_2 \Leftrightarrow (f(u), f(v)) \in E_H$.*

A seguir, apresentamos o teorema e sua demonstração.

Teorema 1 *O problema COMB é NP-completo.*

Prova: COMB pertence a NP pois é possível construir, de forma não-determinística, e verificar o grafo combinado $\bar{G} = (\bar{V}, \bar{E})$, em tempo polinomial. Na fase de construção, determina-se não-deterministicamente, para cada $v \in V_i$, o vértice $\bar{v} \in \bar{V}$ no qual ele está mapeado, e, para cada $e \in E_i$, o arco $\bar{e}$ no qual ele está mapeado, para todo $i = 1, \ldots, n$. A fase de verificação checa, para cada $\bar{v} \in \bar{V}$, se o mapeamento de vértices que ele representa é valido, isto é, se $\bigcap_{\forall v \text{ mapeado em } \bar{v}} B(v) \neq \emptyset$. Similarmente, verifica-se, para cada $\bar{e} \in \bar{E}$, se

o mapeamento de arcos que ele representa é válido, isto é, se as portas de entrada p são equivalentes, $\forall e = (u, v, p)$ mapeado em $\bar{e}$. Finalmente, checa se $A(\bar{G}) \leq k$.

Para provar que COMB é NP-difícil, uma instância arbitrária de ISO, $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, é transformada em uma instância de COMB. Estabelecendo $n = 2$, constrói-se os CDFGs $G'_1 = (V'_1, E'_1)$ e $G'_2 = (V'_2, E'_2)$ correspondentes a G_1 e G_2, respectivamente, e determina-se k e a biblioteca de componentes, tal que:

- $\forall v \in V_i$, existe um vértice $v' \in V'_i$, $\forall i = 1, 2$, representando uma operação comutativa, com número de portas de entrada $n_{v'}$ igual ao grau de entrada de v;
- $\forall e = (u, v) \in E_i$, existe um arco $e' = (u', v', p) \in E'_i$, $\forall i = 1, 2$, sendo as portas de entrada p atribuídas seqüencialmente de 1 a $n_{v'}$ para todos os arcos da forma $(*, v) \in E_i$;
- A biblioteca de componentes possui apenas um bloco lógico com área 1, que pode realizar a computação representada por todos $v' \in V'_i, \forall i = 1, 2$. Além disso, $A_{mux} = 1$.
- $k = A(G'_1) = |V_1| + |E_1|$.

É necessário mostrar que G_1 possui um subgrafo isomorfo a G_2 se e somente se existe um grafo combinado $\bar{G} = (\bar{V}, \bar{E})$ correspondente a G'_1 e G'_2, tal que $A(\bar{G}) \leq k$. Suponha que $H = (V_H, E_H)$ é um subgrafo de G_1 isomorfo a G_2, e que $f : V_2 \rightarrow V_H$ é a função bijetiva de isomorfismo. Seja $\bar{V}$ um conjunto de vértices tal que, para cada $v_1 \in V_H$ e $v_2 \in V_2$ com $f(v_2) = v_1$, existe um vértice $\bar{v} = v'_1/v'_2 \in \bar{V}$. Além disso, para cada $v_1 \in V_1 - V_H$, existe um vértice $\bar{v} = v'_1/- \in \bar{V}$. Analogamente, seja $\bar{E}$ um conjunto de arcos tal que, para cada $e_1 = (u_1, v_1) \in E_H$ e $e_2 = (u_2, v_2) \in E_2$ com $f(u_2) = u_1$ e $f(v_2) = v_1$, existe um arco $\bar{e} = e'_1/e'_2 \in \bar{E}$. Além disso, para cada $e_1 \in E_1 - E_H$, existe um arco $\bar{e} = e'_1/- \in \bar{E}$. Como H é isomorfo a G_2 e todas as operações em G'_1 e G'_2 são comutativas e implementadas pelo mesmo (único) bloco lógico da biblioteca de componentes, $\bar{V}$ e $\bar{E}$ definem o grafo combinado $\bar{G} = (\bar{V}, \bar{E})$. Mais ainda, $A(\bar{G}) = k$ pois $|\bar{V}| = |V_1|$, $|\bar{E}| = |E_1|$, $A_b(\bar{v}) = 1, \forall \bar{v} \in \bar{V}$, e $A_{mux} = 1$.

Reciprocamente, suponha que $\bar{G} = (\bar{V}, \bar{E})$ é um grafo combinado correspondente a G'_1 e G'_2 tal que $A(\bar{G}) \leq k$. Como $k = A(G'_1)$, $\bar{G}$ possui todos os vértices e arcos de G'_2 mapeados em vértices e arcos de G'_1. Seja V_H o conjunto de vértices $v_1 \in V_1$ com

$\bar{v} = v'_1/v'_2 \in \bar{V}$, e seja $f : V_2 \rightarrow V_H$ uma função tal que $f(v_2) = v_1$. Analogamente, seja E_H o conjunto de arcos $e_1 \in E_1$ com $\bar{e} = e'_1/e'_2 \in \bar{E}$. V_H e E_H definem o subgrafo $H = (V_H, E_H)$ de G_1 isomorfo a G_2, e f é a função de isomorfismo.

Esta redução pode ser realizada em tempo polinomial pois requer apenas a construção de G'_1 e G'_2 a partir de G_1 e G_2, e a computação de k, para transformar uma instância de ISO em uma instância de COMB. Além disso, o resultado de COMB é trivialmente transformado no resultado de ISO. □

4.3 Trabalhos Relacionados

O problema de combinação de CDFGs é usado em diferentes contextos para modelar problemas variados. Nesta seção, alguns trabalhos relacionados a este problema são descritos.

4.3.1 Emparelhamento de Peso Máximo em Grafo Bipartido

O método mais comumente utilizado para o problema de combinação de CDFGs combina dois CDFGs a cada iteração e é baseado no problema de encontrar um emparelhamento de peso máximo em um grafo bipartido [34, 71, 47]. As duas partições de vértices do grafo bipartido correspondem aos vértices dos dois CDFGs sendo combinados, respectivamente. Há uma aresta conectando dois vértices no grafo bipartido se eles podem ser mapeados (sobrepostos). O peso desta aresta representa o ganho obtido se os dois vértices forem mapeados, em relação à função objetivo desejada (por exemplo, a redução de área obtida com aquele mapeamento). Como cada vértice de um CDFG deve ser mapeado em no máximo um vértice do outro, o emparelhamento de peso máximo do grafo bipartido fornece os mapeamentos de vértices que maximizam a função objetivo. A partir destes mapeamentos, o grafo combinado é construído.

Em [34] duas soluções são apresentadas para sintetizar uma unidade funcional específica para uma aplicação. Esta unidade é capaz de executar em diferentes modos, cada um correspondendo a um grupo de operações de um CDFG não escalonado que representa a aplicação. As soluções resolvem os problemas de seleção e atribuição de unidades e visam minimizar a área da unidade resultante. Dados dois CDFGs G_1 e G_2, o peso de uma aresta (v_1, v_2) do grafo bipartido representa o ganho em área obtido se os vértices v_1 e v_2 são mapeados. Este peso inclui o ganho em área dos blocos lógicos e das

interconexões. No entanto, o ganho em área das interconexões é apenas estimado como sendo o limite superior da redução da área das interconexões obtida com o mapeamento de v_1 e v_2. Isto é, supõe-se que os vértices predecessores (sucessores) de v_1 em G_1 serão também mapeados nos vértices predecessores (sucessores) de v_2 em G_2, o que pode não acontecer.

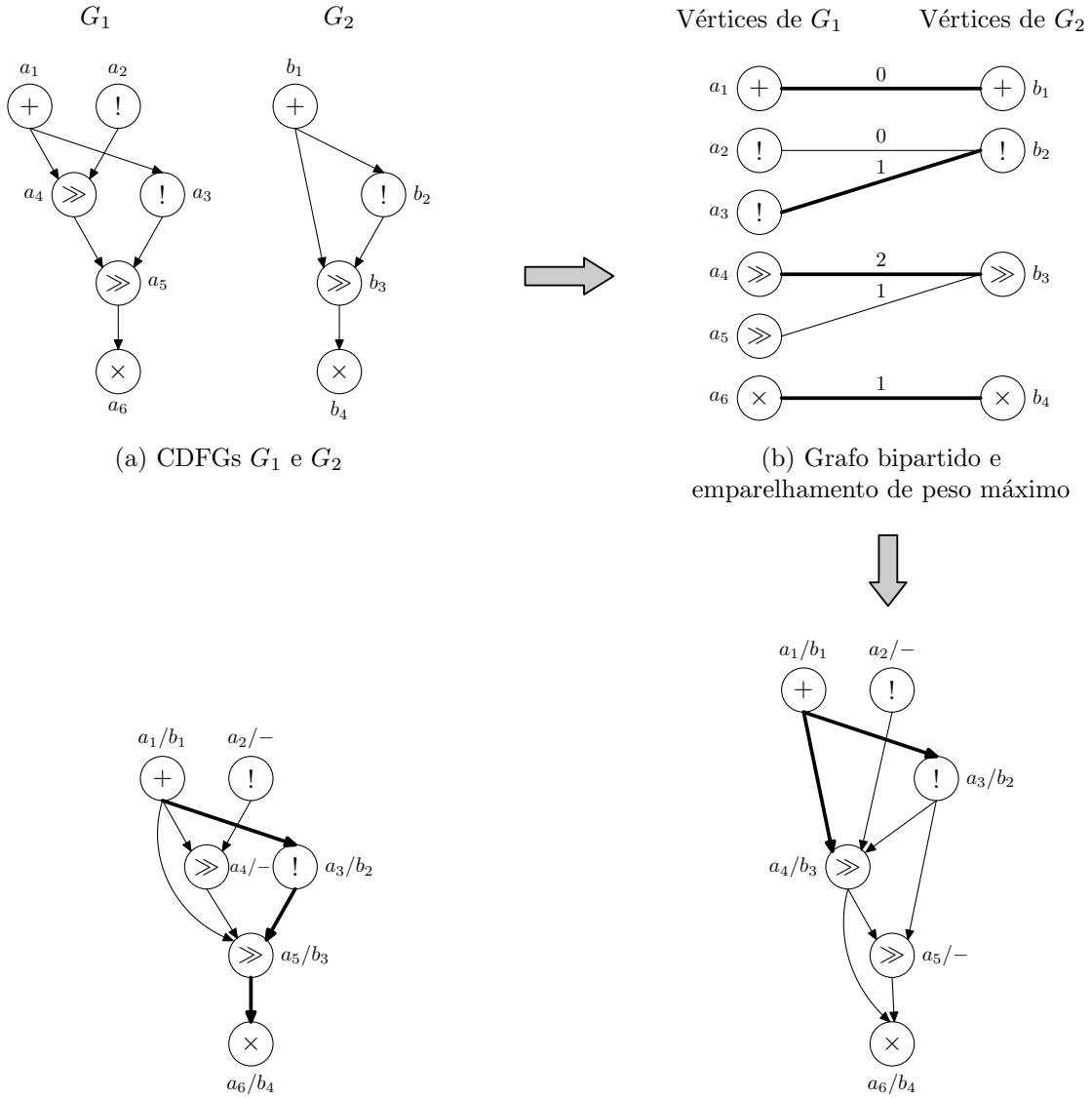
A segunda solução apresentada em [34] utiliza uma abordagem sutilmente diferente para tratar situações em que os dois CDFGs sendo combinados exibem um alto grau de similaridade e regularidade. A posição relativa dos vértices no CDFG é usada como critério de desempate quando várias arestas do grafo bipartido possuem o mesmo peso.

Em [71] os autores combinam dois ou mais circuitos em um circuito reconfigurável, baseados na identificação e mapeamento de componentes comuns aos circuitos. O objetivo é produzir configurações dinâmicas e minimizar o tempo de reconfiguração. Os pesos das arestas do grafo bipartido são definidos usando critérios como tipos dos componentes, posição dos componentes na FPGA, etc.

A técnica de emparelhamento de peso máximo em grafo bipartido também é utilizada em [47] para projetar um *datapath* reconfigurável capaz de realizar a computação correspondente aos laços internos que consomem grande parte do tempo de execução de uma aplicação. O peso da aresta (v_1, v_2) do grafo bipartido representa apenas uma estimativa do número de interconexões que serão mapeadas se os vértices v_1 de G_1 e v_2 de G_2 forem mapeados, supondo que os vértices predecessores de v_1 e v_2 serão também mapeados.

A Figura 4.6 mostra CDFGs G_1 e G_2 e o grafo bipartido obtido a partir deles, com pesos das arestas atribuídos de acordo com a estimativa descrita em [47]. O emparelhamento de peso máximo é mostrado em negrito na figura e produz o grafo combinado $\bar{G}'$ mostrado na Figura 4.6(c). Como a aresta (a_4, b_3) tem peso maior que (a_5, b_3), os vértices a_4 e b_3 são mapeados em $\bar{G}'$. Porém o peso 2 de (a_4, b_3) é obtido supondo que os predecessores de a_4 e b_3 serão mapeados. De fato, os vértices a_1 e b_1 são mapeados, porém a_2 e b_1 não o são. Se o mapeamento a_5/b_3 fosse escolhido, o grafo combinado $\bar{G}$ (Figura 4.6(d)) que possui uma área das interconexões menor, seria obtido.

Esse método é uma abordagem eficiente em termos de tempo de processamento, pois o problema de emparelhamento de peso máximo em um grafo bipartido pode ser resolvido por um algoritmo exato de complexidade temporal polinomial (o método húngaro descrito em [64]). Porém, como os vértices são mapeados ignorando ou utilizando apenas estimativas sobre o mapeamento das interconexões, a solução produzida pode estar muito aquém da solução ótima.



(a) CDFGs G_1 e G_2

(b) Grafo bipartido e emparelhamento de peso máximo

(d) Grafo combinado ótimo $\bar{G}$

(c) Grafo combinado $\bar{G}'$

Figura 4.6: Combinação de CDFGs por emparelhamento de peso máximo em um grafo bipartido

4.3.2 Busca no Espaço de Soluções

Outro grupo de métodos [84, 34] baseia-se em mecanismos de busca para explorar o espaço de soluções, procurando a solução ótima.

Em [84] uma técnica de combinação de CDFGs é proposta para sintetizar uma unidade de processamento multi-função, na qual cada função corresponde a um grupo de operações

na especificação da aplicação. O método mapeia operações nos operadores da unidade de processamento, considerando que estes operadores foram selecionados previamente. A abordagem é baseada nos algoritmos de busca local por melhorias sucessivas (*iterative improvement*) e *simulated annealing* [1] e busca reduzir o custo da rede de interconexões da unidade de processamento. Ambos os algoritmos começam com uma solução inicial aleatória, que consiste de um mapeamento arbitrário dos CDFGs. Eles então exploram o espaço de soluções, continuamente mudando da solução corrente para uma solução vizinha a ela (escolhida aleatoriamente), de acordo com um critério de aceitação de custo.

O conceito de vizinhança é definido usando um mecanismo de troca, onde os mapeamentos de duas operações de um mesmo CDFG são trocados. Por exemplo, dados os CDFGs G_1 e G_2, suponha que em uma solução (isto é, em um grafo combinado $\bar{G}'$), há os mapeamentos de vértices a_1/b_1 e a_2/b_2. Um exemplo de solução vizinha a esta é um grafo combinado $\bar{G}''$, com os mapeamentos a_1/b_2 e a_2/b_1 e todos os demais mapeamentos de vértices iguais a $\bar{G}'$. Assim, define-se a vizinhança da solução corrente como o conjunto de soluções que podem ser alcançadas, a partir dela, realizando-se apenas uma troca. No pior caso, supondo apenas dois CDFGs de entrada, cada solução $\bar{G} = (\bar{V}, \bar{E})$ pode ter $O(|\bar{V}|^2)$ soluções vizinhas.

A solução obtida pelo algoritmo de melhorias sucessivas é o primeiro mínimo local encontrado, que pode ser muito inferior que a solução ótima. O método de *simulated annealing* aceita em sua busca, transições que pioram (de forma limitada) o custo da solução, com o objetivo de escapar de mínimos locais. Conseqüentemente, possui um tempo de execução mais longo.

Uma solução descrita em [34] também utiliza um mecanismo de busca local por melhorias sucessivas, baseado no algoritmo apresentado em [84]. Porém, esta solução não exige que os operadores sejam selecionados previamente, e começa com uma solução inicial diferente. Novamente o espaço de soluções é explorado mudando-se da solução corrente para uma das soluções vizinhas, alcançadas por uma troca no mapeamento de duas operações. O critério de aceitação não permite transições que piorem o custo da solução, mas aceita transições com diferença de custo nula, isto é, que não melhoram o custo da solução.

Como o espaço de soluções do problema de combinação de CDFGs possui um tamanho exponencial e é explorado aleatoriamente, o tempo de execução desses métodos de busca pode não ser limitado polinomialmente. Todavia, na prática eles algumas vezes comportam-se razoavelmente bem.

4.3.3 Modelo de Programação Inteira

Em outra solução proposta em [34], que também combina dois CDFGs a cada passo, um modelo de programação inteira é formulado para representar uma modelagem global da área das interconexões. Neste modelo, o ganho em área das interconexões correspondente ao mapeamento dos vértices v_1 de um CDFG G_1 e v_2 de G_2 é dependente do mapeamento de seus vértices predecessores nos CDFGs. Este modelo provê uma solução quasi-ótima para o problema de combinar apenas dois CDFGs (pois não explora a comutatividade), porém de tempo exponencial. Esta solução pode ser utilizada como uma aproximação para combinar vários CDFGs iterativamente.

4.3.4 Outras Soluções

Nenhum dos trabalhos descritos até aqui explora a comutatividade das operações para melhorar o compartilhamento de interconexões. Tradicionalmente, a comutatividade é explorada em passos de pré- ou pós-processamento. No primeiro caso, quando operandos são comutados antes da atribuição de unidades, utilizam-se apenas estimativas sobre o mapeamento de blocos lógicos e interconexões, limitando-se as possibilidades de mapeamento na fase de atribuição. Por outro lado, se a comutatividade é explorada quando a atribuição de unidades já foi realizada, soluções melhores para o mapeamento de blocos lógicos e interconexões podem já ter sido descartadas e a comutação é utilizada de forma mais restrita. Portanto, idealmente, a comutação de operandos deve ser considerada durante a seleção e atribuição de unidades.

Um modelo que permite o mapeamento conjunto de operações, variáveis e transferências de dados é apresentado em [17]. O problema é formulado como um problema de *clustering* em grafos e alguns algoritmos simples são descritos. O mapeamento de unidades funcionais, registradores e interconexões é realizado concomitantemente para permitir a busca da melhor solução global. Porém, quando a rede de interconexões é baseada em MUXes, cada transferência de dados é agrupada com a sua operação ou variável predecessora. Ou seja, o mapeamento das interconexões é realizado considerando somente o mapeamento dos seus blocos lógicos de origem. Para minimizar a rede de interconexões deveriam ser considerados também os mapeamentos dos blocos sucessores. Neste trabalho, a comutatividade das operações é explorada em um passo de pós-processamento.

Em [9] uma solução heurística para o problema de combinação de grafos de fluxo de

dados (*Data-Flow Graphs* – DFGs) é apresentada. O objetivo é construir um *datapath* em FPGA, capaz de executar um conjunto de instruções especializadas para uma determinada aplicação. Cada instrução é representada por um DFG e o compartilhamento de recursos é explorado para minimizar a área do *datapath*. Esta abordagem é baseada no problema de encontrar a subseqüência comum mais longa de duas seqüências. O passo inicial do algoritmo consiste em enumerar todos os caminhos de cada DFG. Em seguida, os caminhos são sobrepostos iterativamente, a partir da determinação da subseqüência comum mais longa entre cada par de caminhos que não se originam de um mesmo DFG. Para o problema de subseqüência comum mais longa, um algoritmo de complexidade de tempo polinomial é conhecido (utilizando programação dinâmica, como descrito em [22]). Assim, o algoritmo proposto é polinomial em relação ao número de caminhos em todos os DFGs de entrada. No entanto, o número de caminhos de um DFG é exponencial em relação ao seu número de vértices. Logo, a solução proposta possui uma complexidade de tempo exponencial em relação à sua entrada.

4.4 Combinação Global vs. Incremental de CDFGs

Várias das técnicas de combinação de CDFGs descritas são incrementais, isto é, a cada passo dois CDFGs são combinados, formando um grafo combinado parcial. Ao combinar dois CDFGs G_1 e G_2, alguns mapeamentos já são estabelecidos, com base em informações locais de G_1 e G_2. Estes mapeamentos não podem ser desfeitos ao combinar o resultado com outro CDFG G_3. Desta forma, ao combinar os CDFGs iterativamente, estes algoritmos tornam-se uma estratégia gulosa. Porém, as decisões tomadas localmente podem se revelar inadequadas em uma iteração posterior, e não levar ao melhor resultado final. Para que a solução exata para a combinação de n CDFGs, $n > 2$, seja obtida, é necessário combinar todos os CDFGs ao mesmo tempo, isto é, globalmente. Porém, dado que o problema de combinar apenas dois CDFGs é NP-difícil (conforme demonstrado na Seção 4.2), a construção de uma solução global resultaria em um espaço de soluções ainda maior e em um tempo de processamento excessivo.

Adotada uma estratégia incremental para combinar os CDFGs, a ordem em que soluções parciais são combinadas também interfere no resultado. Isto é, combinar G_1 com G_2, e depois combinar o resultado com G_3, pode produzir um grafo combinado diferente daquele obtido ao combinar G_1 com G_3 e o resultado com G_2. Porém, o número de ordenações diferentes dos n CDFGs de entrada é exponencial em função de n. Logo,

alguma ordenação dos CDFGs é em geral adotada.

Em [34] algumas diferentes ordenações de CDFGs são avaliadas:

- combinar os CDFGs na ordem em que foram fornecidos na entrada de dados;
- a cada iteração, combinar os dois CDFGs menores (em relação ao número de vértices);
- a cada iteração, combinar os dois CDFGs maiores;
- a cada iteração, combinar os dois CDFGs mais similares (de acordo com uma função de similaridade calculada para cada par de CDFGs);
- a cada iteração, combinar os dois CDFGs mais dissimilares.

Os resultados obtidos em [34] mostram que não há diferenças significativas na área da unidade funcional específica produzida, em relação aos diferentes esquemas de ordenação. As três primeiras ordenações acima produziram, em vários casos, as melhores soluções, mas não de forma geral. Por outro lado, as duas ordenações baseadas na função de similaridade requerem um maior tempo de processamento devido ao seu cálculo.

Capítulo 5

Otimização da Área de Interconexões

Neste capítulo nós propomos e avaliamos uma heurística para resolver o problema de combinação de CDFGs. Este algoritmo é apresentado em [58] e utilizado em [48] no projeto de um *datapath* reconfigurável que atua como um coprocessador de uma plataforma específica para uma aplicação. A heurística realiza a tarefa de atribuição de unidades, supondo que a seleção de unidades foi realizada previamente. A atribuição de blocos lógicos e interconexões é feita simultaneamente, com o objetivo de encontrar uma solução que minimize a área da rede de interconexões $A_i(\bar{G})$ do *datapath* resultante $\bar{G}$.

A heurística proposta combina dois CDFGs, e para combinar vários CDFGs, ela é aplicada iterativamente. Primeiro, dois CDFGs de entrada são combinados, então o grafo resultante é combinado com outro CDFG de entrada, e assim por diante.

As Seções 5.1 e 5.2 descrevem as estruturas auxiliares utilizadas no algoritmo proposto, que é apresentado na Seção 5.3. A Seção 5.4 ilustra como este algoritmo é capaz de explorar a comutatividade das operações para minimizar ainda mais a área de interconexões do *datapath*. Algumas questões relacionadas com a ordem de combinação dos CDFGs são discutidas na Seção 5.5. Finalmente, a Seção 5.6 apresenta uma avaliação experimental do algoritmo proposto.

5.1 Grafo de Compatibilidade

Inicialmente, dados dois CDFGs de entrada G_1 e G_2, um grafo de compatibilidade é construído para representar todos os possíveis mapeamentos de arcos entre eles,

e a coerência entre estes mapeamentos. Pela Definição 3, dois arcos de G_1 e G_2, respectivamente, podem ser mapeados se os seus vértices origem podem ser mapeados, assim como seus vértices destino, e se suas portas de entrada são equivalentes. O grafo de compatibilidade possui então um vértice para cada possível mapeamento de arcos. Dadas as condições para a equivalência de portas de entrada descritas na Definição 6, este grafo também contém mapeamentos de arcos obtidos através da comutação de operandos. O grafo de compatibilidade é definido a seguir.

Definição 9 *O grafo de compatibilidade, correspondente aos CDFGs* $G_1 = (V_1, E_1)$ *e* $G_2 = (V_2, E_2)$*, é um grafo não direcionado* $G_c = (V_c, E_c)$*, tal que:*

- *Cada vértice* $v_c \in V_c$ *corresponde a um possível mapeamento* $(u_1, v_1, p_1)/(u_2, v_2, p_2)$ *de arcos* $(u_1, v_1, p_1) \in E_1$ *e* $(u_2, v_2, p_2) \in E_2$*, tal que* $B(u_1) \cap B(u_2) \neq \emptyset$*,* $B(v_1) \cap B(v_2) \neq \emptyset$*, e* p_1 *e* p_2 *são equivalentes.*

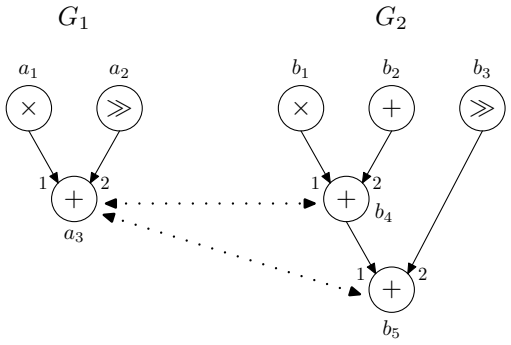
- *Existe uma aresta* $e_c = (u_c, v_c) \in E_c$ *se os mapeamentos representados por* u_c *e* v_c *são compatíveis.*

Para construir o grafo de compatibilidade, é necessário também definir o conceito de compatibilidade entre mapeamentos.

Definição 10 *Dois mapeamentos de arcos dos CDFGs* $G_1 = (V_1, E_1)$ *e* $G_2 = (V_2, E_2)$ *<u>não</u> são compatíveis se eles mapeiam o mesmo vértice de* V_1 *em dois vértices distintos de* V_2*, ou vice-versa.*

Esse critério de compatibilidade é ilustrado na Figura 5.1. Dados os CDFGs G_1 e G_2, existem dois possíveis mapeamentos de arcos entre eles, mostrados na Figura 5.1(b). Estes dois mapeamentos não são compatíveis pois eles mapeiam o mesmo vértice a_3 de G_1 em dois vértices distintos, b_4 e b_5, de G_2.

Usando as definições apresentadas acima, o grafo de compatibilidade pode ser construído. Note que o termo grafo de compatibilidade utilizado neste trabalho possui um significado diferente de seu uso convencional em HLS. Em HLS, cada vértice do grafo de compatibilidade corresponde a uma operação do CDFG de entrada. Uma aresta no grafo de compatibilidade representa que aquelas duas operações podem ser atribuídas a um mesmo recurso. Neste trabalho, cada vértice do grafo de compatibilidade representa um possível mapeamento de duas transferências de dados de dois CDFGs de entrada, isto é,



(a) CDFGs G_1 e G_2

$(a_1, a_3, 1)/(b_1, b_4, 1)$

$(a_2, a_3, 2)/(b_3, b_5, 2)$

(b) Mapeamentos de arcos

Figura 5.1: Mapeamentos de arcos incompatíveis: a_3 é mapeado em b_4 e b_5

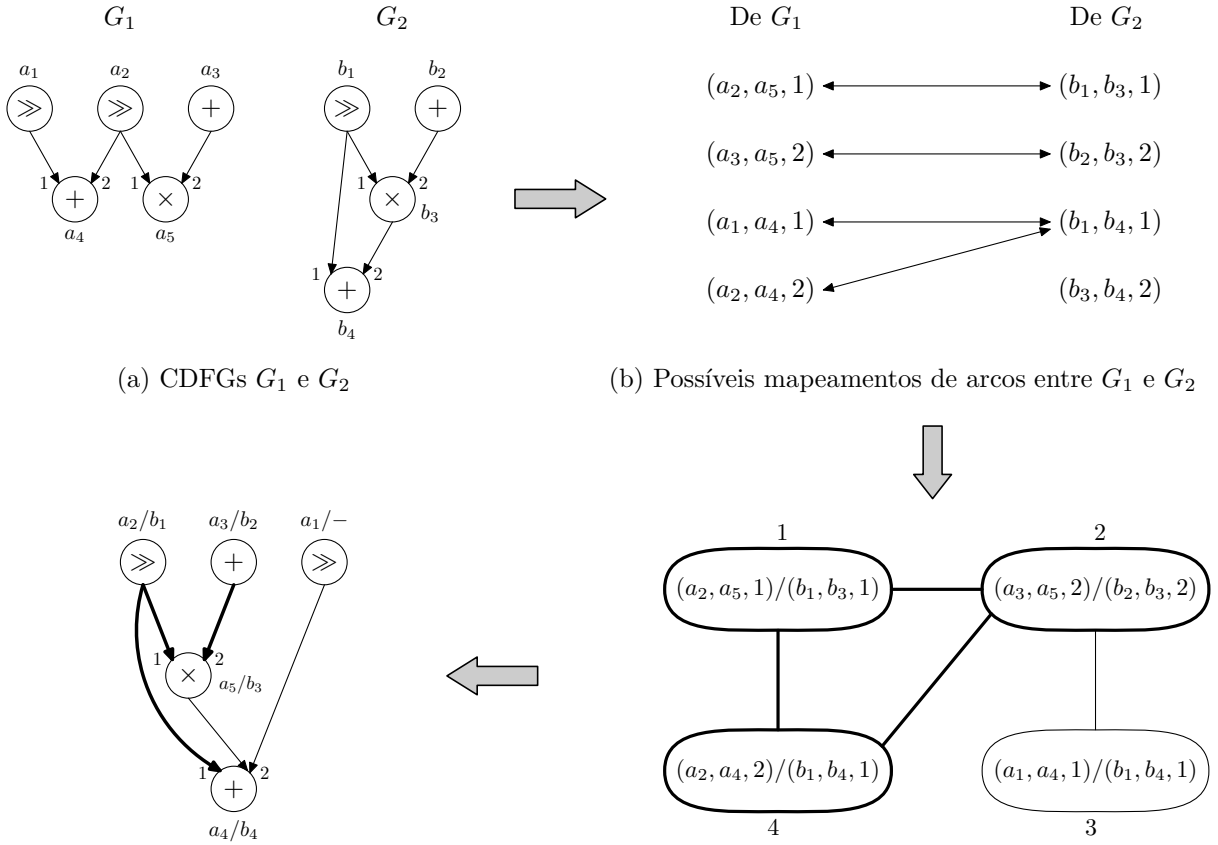
duas transferências que podem ser atribuídas a um mesmo recurso. Uma aresta no grafo de compatibilidade representa dois mapeamentos que podem ser usados em conjunto.

A Figura 5.2 ilustra todos os passos do algoritmo proposto, inclusive a construção do grafo de compatibilidade. Dados os CDFGs G_1 e G_2 da Figura 5.2(a), os arcos de G_1 e G_2 que podem ser sobrepostos são listados na Figura 5.2(b). Cada possível mapeamento de arcos é representado por uma seta dupla ligando os dois arcos mapeados. A Figura 5.2(c) mostra o grafo de compatibilidade G_c, construído a partir de G_1 e G_2, que representa os possíveis mapeamentos de arcos entre os dois CDFGs (listados na Figura 5.2(b)). Considere, por exemplo, os mapeamentos de arcos $(a_2, a_4, 2)/(b_1, b_4, 1)$ (vértice 4 de G_c) e $(a_2, a_5, 1)/(b_1, b_3, 1)$ (vértice 1 de G_c). Nestes mapeamentos, nenhum vértice de G_1 é mapeado em dois vértices distintos de G_2, ou vice-versa. Como resultado, os dois mapeamentos são compatíveis, e a aresta (1,4) é inserida em G_c. Por outro lado, não há aresta em G_c entre os vértices 3 e 4. A razão é que os mapeamentos representados por eles são incompatíveis, pois b_1 de G_2 é mapeado tanto em a_1 quanto em a_2 de G_1.

5.2 Clique Máximo

Nem todos os possíveis mapeamentos de arcos representados no grafo de compatibilidade G_c podem ser utilizados. Para encontrar o grafo combinado $\bar{G}$ tal que $A_i(\bar{G})$ seja mínimo, é necessário determinar o maior conjunto de mapeamentos de arcos compatíveis entre si. Isto é obtido encontrando-se um clique máximo de G_c, definido a seguir.

Definição 11 *Um clique máximo de um grafo $G_c = (V_c, E_c)$ é um conjunto de vértices*



(a) CDFGs G_1 e G_2

(b) Possíveis mapeamentos de arcos entre G_1 e G_2

(d) Grafo combinado $\bar{G}$

(c) Clique máximo do grafo de compatibilidade G_c

Figura 5.2: Passos do algoritmo de combinação de CDFGs

$C \subseteq V_c$ tal que, para todos os vértices $u_c, v_c \in C$, a aresta $(u_c, v_c) \in E_c$, e $|C|$ é máximo.

Os vértices do clique máximo C do grafo de compatibilidade G_c fornecem os mapeamentos de arcos que devem ser utilizados na construção do grafo combinado $\bar{G}$. Além disso, estes mapeamentos de arcos fornecem também alguns mapeamentos de vértices (os vértices origem e destino dos arcos mapeados) que devem ser utilizados. Assim, blocos lógicos e interconexões são mapeados simultaneamente.

Se dois vértices, v_1 de G_1 e v_2 de G_2, são mapeados, o *datapath* resultante possuirá apenas um bloco lógico capaz de realizar a computação representada por ambos os vértices, ao invés de dois blocos lógicos para realizar cada computação, respectivamente. Se dois arcos de G_1 e G_2 são mapeados, ao invés de haver no *datapath* resultante duas interconexões, haverá apenas uma, e o MUX (se houver), na porta de entrada correspondente do bloco lógico destino, terá uma entrada a menos. Portanto, dado que o

clique máximo representa o maior número possível de mapeamentos de arcos compatíveis entre si, o grafo combinado G produzido é tal que $A_i(\bar{G})$ é mínimo.

O clique máximo do grafo de compatibilidade G_c da Figura 5.2(c) possui os vértices 1, 2 e 4, e é mostrado em negrito na figura. Por exemplo, o vértice 1 do clique indica que os arcos $(a_2, a_5, 1)$ de G_1 e $(b_1, b_3, 1)$ de G_2 devem ser mapeados. Indica também que os vértices a_2 de G_1 e b_1 de G_2 devem ser mapeados, assim como os vértices a_5 e b_3.

O grafo combinado $\bar{G}$ é então construído em uma seqüência de passos, a partir de um grafo vazio, com base nos mapeamentos indicados por C. Inicialmente, para cada vértice $v_c \in C$ representando o mapeamento dos arcos (u_1, v_1, p_1) de G_1 e (u_2, v_2, p_2) de G_2 , dois vértices $\bar{u}$ e $\bar{v}$ são inseridos em $\bar{G}$ (se ainda não o foram), correspondendo aos mapeamentos de vértices u_1/u_2 e v_1/v_2, respectivamente. Além disso, o arco $\bar{e} = (\bar{u}, \bar{v}, \bar{p})$ é também inserido em $\bar{G}$, correspondendo aos arcos mapeados representados por v_c. Se v_c representa um mapeamento de arcos obtido através da comutação de operandos, as portas de entrada de $\bar{v}$ em $\bar{G}$ são modificadas apropriadamente.

O clique máximo indica todos os mapeamentos de arcos de $\bar{G}$, porém não necessariamente fornece todos os mapeamentos de vértices. Assim, para cada vértice v_1 de G_1 que ainda não foi mapeado, é inserido em $\bar{G}$ um vértice $\bar{v}$, correspondendo ao mapeamento de vértices $v_1/-$. Em seguida, para cada vértice v_2 de G_2 ainda não mapeado, se existe um vértice $\bar{v}$ em $\bar{G}$, tal que $\bar{v}$ corresponde a um mapeamento de vértices $v_1/-$ e $B(v_1) \cap B(v_2) \neq \emptyset$, então v_1 e v_2 são mapeados e $\bar{v}$ passa a corresponder ao mapeamento de vértices v_1/v_2. Caso contrário, é inserido em $\bar{G}$ um novo vértice $\bar{v}$, correspondendo ao mapeamento de vértices $-/v_2$. Por fim, todos os arcos de G_1 e G_2 que não pertencem a nenhum mapeamento em C, são também inseridos em $\bar{G}$ como arcos não mapeados, conectando os vértices correspondentes.

Por exemplo, utilizando o clique máximo da Figura 5.2(c), o grafo combinado $\bar{G}$, mostrado na Figura 5.2(d), é obtido. Se apenas mapeamentos de vértices fossem considerados, vários grafos combinados diferentes poderiam ser produzidos, todos com uma área de interconexões maior que $A_i(\bar{G})$.

5.3 Algoritmo de Combinação de CDFGs

A heurística proposta por nós para a combinação de CDFGs é apresentada no Algoritmo 5.1. Os passos de construção do grafo de compatibilidade e do grafo combinado são detalhados nos Algoritmos 5.2 e 5.3. Por simplicidade, algumas construções e

```
CombinaCDFGs
    ENTRADA: n CDFGs G_i = (V_i, E_i), i = 1...n
    SAÍDA: grafo combinado Ḡ = (V̄, Ē),
        tal que A_i(Ḡ) é mínimo
    /* Inicialmente, Ḡ é o CDFG de entrada G_1 */
    Ḡ ← G_1;
    /* Iterativamente combina Ḡ com CDFG G_i */
    PARA i ← 2 ATÉ n FAÇA
        G_c ← ConstróiGrafoCompatibilidade(Ḡ, G_i);
        C ← EncontraCliqueMáximo(G_c);
        Ḡ ← ReconstróiGrafoCombinado(Ḡ, G_i, C);
```

Algoritmo 5.1: Combinação de CDFGs para minimizar área de interconexões

estruturas de dados utilizadas para tornar o algoritmo mais eficiente são omitidas.

Na combinação de apenas dois CDFGs, esse algoritmo produz um *datapath* com área de interconexões mínima, desde que um algoritmo exato para o problema de clique máximo seja utilizado. Porém, o problema de clique máximo também é NP-completo [33, GT19]. Logo, uma heurística de tempo polinomial é utilizada para encontrar o clique máximo, e assim o algoritmo apresentado torna-se também uma heurística com complexidade de tempo polinomial.

Para combinar dois CDFGs $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, a complexidade de tempo do passo "ConstróiGrafoCompatibilidade" é, no pior caso, $O(|E_1|^2 \times |E_2|^2)$, pois $|V_c|$ é no máximo $O(|E_1| \times |E_2|)$. O algoritmo usado no passo "EncontraCliqueMáximo" é baseado em um método de busca local [5]. Esta heurística é ajustada de forma que seu tempo de execução seja limitado polinomialmente por $|V_c|$, isto é, a busca é interrompida quando um determinado limite de tempo é atingido. Finalmente, a complexidade de tempo do passo "ReconstróiGrafoCombinado" é, no pior caso, $O(|E_1| \times |E_2|)$. Esta situação de pior caso acontece quando todos os vértices de G_1 e G_2 representam uma mesma operação comutativa (o que na prática nunca acontece), e em conseqüência, todos os arcos de G_1 são passíveis de serem mapeados em todos os arcos de G_2.

```
ConstróiGrafoCompatibilidade(G_i, G_j)
    /* Inicialmente, G_c é vazio */
    V_c ← ∅; E_c ← ∅;
    /* Determina vértices de G_c */
    PARA cada e_i = (u_i, v_i, p_i) ∈ E_i FAÇA
        PARA cada e_j = (u_j, v_j, p_j) ∈ E_j FAÇA
            SE B(u_i) ∩ B(u_j) ≠ ∅ E B(v_i) ∩ B(v_j) ≠ ∅ E p_i e p_i são equivalentes ENTÃO
                V_c ← V_c ∪ {v_c}, onde v_c = e_i/e_j;
    /* Determina arestas de G_c */
    PARA cada u_c e v_c ∈ V_c, u_c ≠ v_c FAÇA
        SE u_c e v_c são compatíveis ENTÃO
            E_c ← E_c ∪ {e_c}, onde e_c = (u_c, v_c);
    RETORNA G_c = (V_c, E_c);
```

Algoritmo 5.2: Construção do grafo de compatibilidade G_c

5.4 Explorando a Comutatividade

O algoritmo de combinação de CDFGs apresentado explora a comutatividade das operações durante a tarefa de atribuição de unidades. Como a atribuição de blocos lógicos e interconexões é realizada simultaneamente, as decisões de comutação de operandos é realizada usando dados precisos sobre a área de interconexões.

Durante a construção do grafo de compatibilidade, as possíveis comutações de operandos já são consideradas, isto é, todos os possíveis mapeamentos de arcos são enumerados, inclusive aqueles obtidos com a comutatividade. Assim, o clique máximo do grafo de compatibilidade indica os mapeamentos de vértices e arcos que resultam na maior redução da área de interconexões do *datapath*, mapeamentos estes obtidos com a comutação ou não de operandos.

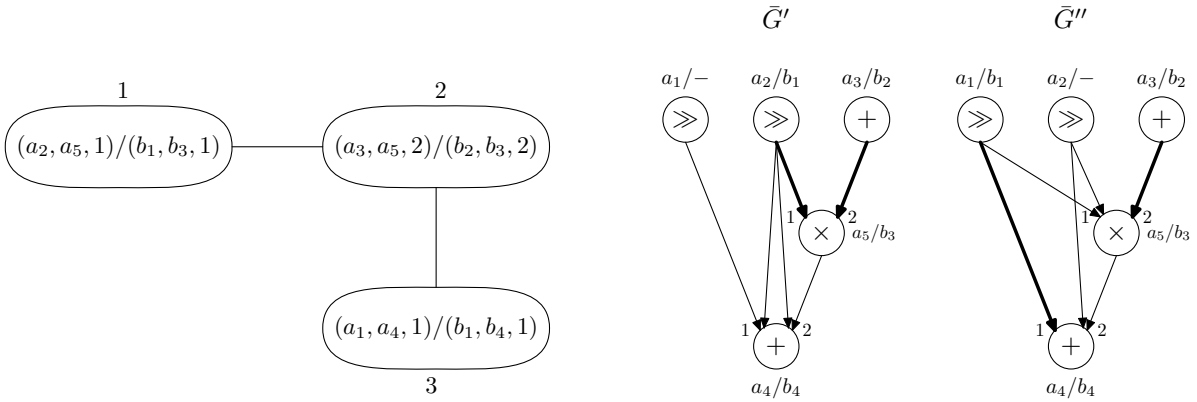
Por exemplo, dados os CDFGs G_1 e G_2 da Figura 5.2(a), o vértice 4 do grafo de compatibilidade G_c correspondente (mostrado na Figura 5.2(c)) representa mapeamento de arcos $(a_2, a_4, 2)/(b_1, b_4, 1)$, obtido com a comutação dos operandos da operação de adição, pois as portas de entrada dos arcos mapeados são diferentes. Porém, se a comutatividade não é explorada, o grafo de compatibilidade obtido é o grafo G'_c, mostrado na Figura 5.3(a). Neste grafo, há dois clique máximos C' e C'', compostos pelos vértices $\{1, 2\}$ e $\{2, 3\}$, respectivamente. Caso o clique máximo C' seja escolhido, obtém-se o grafo combinado $\bar{G}'$, mostrado na Figura 5.3(b). Caso contrário, se é escolhido o clique máximo

ReconstróiGrafoCombinado(G_i, G_j, C)

`AUXILIAR:` funções de mapeamento de vértices $\mathcal{V}_i : V_i \rightarrow \bar{V} \cup \{-1\}$ e $\mathcal{V}_j : V_j \rightarrow \bar{V} \cup \{-1\}$;
funções de mapeamento de arcos $\mathcal{E}_i : E_i \rightarrow \{0,1\}$ e $\mathcal{E}_j : E_j \rightarrow \{0,1\}$

/* Inicialmente, $\bar{G}$ é vazio e todos os vértices e arcos de G_i e G_j não estão mapeados em $\bar{G}$ */
$\bar{V} \leftarrow \emptyset$; $\bar{E} \leftarrow \emptyset$;
$\mathcal{V}_i(v_i) \leftarrow -1, \forall v_i \in V_i$; $\mathcal{V}_j(v_j) \leftarrow -1, \forall v_j \in V_j$;
$\mathcal{E}_i(e_i) \leftarrow 0, \forall e_i \in E_i$; $\mathcal{E}_j(e_j) \leftarrow 0, \forall e_j \in E_j$;
/* Trata mapeamentos de vértices e arcos indicados em C */
`PARA` cada $v_c = e_i/e_j \in V_c, e_i = (u_i, v_i, p_i)$ e $e_j = (u_j, v_j, p_j)$ `FAÇA`
  `SE` $\mathcal{V}_i(u_i) = -1$ `ENTÃO`
    $\bar{V} \leftarrow \bar{V} \cup \{\bar{u}\}$, onde $\bar{u} = u_i/u_j$;
    $\mathcal{V}_i(u_i) \leftarrow \bar{u}$; $\mathcal{V}_j(u_j) \leftarrow \bar{u}$;
  `SE` $\mathcal{V}_i(v_i) = -1$ `ENTÃO`
    $\bar{V} \leftarrow \bar{V} \cup \{\bar{v}\}$, onde $\bar{v} = v_i/v_j$;
    $\mathcal{V}_i(v_i) \leftarrow \bar{v}$; $\mathcal{V}_j(v_j) \leftarrow \bar{v}$;
  $\bar{E} \leftarrow \bar{E} \cup \{\bar{e}\}$, onde $\bar{e} = e_i/e_j = (\bar{u}, \bar{v}, \bar{p})$, com $\bar{p}$ definido apropriadamente;
  $\mathcal{E}_i(e_i) \leftarrow 1$; $\mathcal{E}_j(e_j) \leftarrow 1$;
/* Trata vértices não mapeados */
`PARA` cada $v_i \in V_i$ `FAÇA`
  `SE` $\mathcal{V}_i(v_i) = -1$ `ENTÃO`
    $\bar{V} \leftarrow \bar{V} \cup \{\bar{v}\}$, onde $\bar{v} = v_i/-$;
    $\mathcal{V}_i(v_i) \leftarrow \bar{v}$;
`PARA` cada $v_j \in V_j$ `FAÇA`
  `SE` $\mathcal{V}_j(v_j) = -1$ `ENTÃO`
    `SE` $\exists\, \bar{v} = v_i/- \in \bar{V}$, tal que $B(v_i) \cap B(v_j) \neq \emptyset$ `ENTÃO`
      $\mathcal{V}_j(v_j) \leftarrow \bar{v}$, onde $\bar{v} = v_i/v_j$;
    `SENÃO`
      $\bar{V} \leftarrow \bar{V} \cup \{\bar{v}\}$, onde $\bar{v} = -/v_j$;
      $\mathcal{V}_j(v_j) \leftarrow \bar{v}$;
/* Trata arcos não mapeados */
`PARA` cada $e_i = (u_i, v_i, p_i) \in E_i$ `FAÇA`
  `SE` $\mathcal{E}_i(e_i) = 0$ `ENTÃO`
    $\bar{E} \leftarrow \bar{E} \cup \{\bar{e}\}$, onde $\bar{e} = e_i/- = (\mathcal{V}_i(u_i), \mathcal{V}_i(v_i), \bar{p})$, com $\bar{p}$ definido apropriadamente;
    $\mathcal{E}_i(e_i) \leftarrow 1$
`PARA` cada $e_j = (u_j, v_j, p_j) \in E_j$ `FAÇA`
  `SE` $\mathcal{E}_j(e_j) = 0$ `ENTÃO`
    $\bar{E} \leftarrow \bar{E} \cup \{\bar{e}\}$, onde $\bar{e} = -/e_j = (\mathcal{V}_j(u_j), \mathcal{V}_j(v_j), \bar{p})$, com $\bar{p}$ definido apropriadamente;
    $\mathcal{E}_j(e_j) \leftarrow 1$
`RETORNA` $\bar{G} = (\bar{V}, \bar{E})$;

Algoritmo 5.3: Construção do grafo combinado $\bar{G}$



(a) Grafo de compatibilidade G'_c

(b) Grafos combinados $\bar{G}'$ e $\bar{G}''$

Figura 5.3: Combinação sem comutação dos CDFGs G_1 e G_2 da Figura 5.2

C'', o grafo combinado produzido é $\bar{G}''$, também mostrado na Figura 5.3(b). Tanto $\bar{G}'$ quanto $\bar{G}''$ possuem uma interconexão e um MUX a mais que o grafo combinado $\bar{G}$ da Figura 5.2, obtido com a comutação dos operandos da operação de adição a_4 de G_1. Logo, $A_i(\bar{G}) < A_i(\bar{G}')$ e $A_i(\bar{G}) < A_i(\bar{G}'')$.

No grafo combinado $\bar{G}'$, um passo de pós-processamento ainda pode identificar a oportunidade de comutação dos operandos de a_4, modificar alguns mapeamentos, e assim obter o grafo $\bar{G}$. Porém, no grafo $\bar{G}''$ a atribuição de unidades realizada elimina esta oportunidade. Isto ilustra a necessidade de explorar a comutatividade durante a atribuição das unidades, e não apenas após esta atribuição.

5.5 Ordem de Combinação dos CDFGs

A técnica de combinação de CDFGs proposta neste capítulo é incremental. Logo alguma ordenação dos CDFGs precisa ser adotada. Assim, o algoritmo é uma heurística para a combinação de n CDFGs, $n > 2$, mesmo que um algoritmo exato para problema do clique máximo seja utilizado.

Outra questão relacionada à combinação incremental dos CDFGs diz respeito à escolha do clique máximo do grafo de compatibilidade. Na combinação de dois CDFGs G_1 e G_2, o grafo de compatibilidade construído pode possuir vários cliques máximos, por exemplo C e C'. Com respeito apenas à combinação de G_1 e G_2, C e C' são equivalentes, isto é, produzem grafos combinados $\bar{G}$ e $\bar{G}'$, respectivamente, com a mesma área mínima de

interconexões. Porém, nas iterações posteriores, isto é, ao combinar o grafo combinado parcial com outro CDFG G_3, os mapeamentos obtidos com C podem ser mais proveitosos que os obtidos com C', e assim, podem levar a um grafo combinado final com área de interconexões menor.

5.6 Resultados Experimentais

Com o objetivo de avaliar o nosso algoritmo de combinação de CDFGs, nós realizamos uma série de experimentos utilizando um conjunto de aplicações do *benchmark* MediaBench [52]. Os CDFGs foram gerados, a partir do código fonte das aplicações (na linguagem C ou C++), utilizando a metodologia e as ferramentas apresentadas em [7].

Cada aplicação foi compilada usando o compilador GCC e informações de *profile* foram extraídas para determinar quais laços internos contribuem mais para o tempo de execução da aplicação. Tais laços foram então considerados para mapeamento em um *datapath* reconfigurável. Para cada laço, um CDFG foi gerado a partir do código RTL (*Register Transfer Language*, representação intermediária do GCC [76]) do corpo do laço. Usar RTL ao invés do código de máquina gerado pelo compilador permite que o código do laço seja extraído após a maioria das otimizações de código independentes de máquina, porém antes da alocação de registradores e das otimizações dependentes de máquina. Sempre que possível, a otimização de *inlining* [61] das funções foi aplicada.

Na construção de cada CDFG, seus vértices representaram as operações da seção de código, e as dependências de dados entre as operações foram obtidas a partir das listas de uso-definição [61] mantidas pelo GCC. O grafo de dependências de controle (*Control-Dependence Graph* – CDG) [61] foi utilizado para tratar as dependências de controle. A técnica de *if-conversion* [2] para predicação foi aplicada e seletores foram inseridos no CDFG. A predicação foi utilizada apenas para operações de acesso à memória.

Assim, a seção de código da aplicação correspondente a um CDFG não está limitada a apenas um bloco básico e pode conter construções de controle, tais como *"if"*, *"if-else"* e *"switch"*, podendo ser inclusive não-estruturada (com *"goto"*). Além disso, os CDFGs podem representar operações com sub-expressões comuns (isto é, não estão limitados a dependências em forma de árvore). Por simplicidade, laços aninhados não são tratados. As informações sobre os CDFGs gerados são mostradas na Tabela 5.1.

A nossa heurística de combinação por clique máximo em grafo de compatibilidade, referida nesta seção como **Clique Máximo**, foi aplicada aos CDFGs de cada aplicação

Aplicação	n	G_1		G_2		G_3		G_4	
		$\|V_1\|$	$\|E_1\|$	$\|V_2\|$	$\|E_2\|$	$\|V_3\|$	$\|E_3\|$	$\|V_4\|$	$\|E_4\|$
ADPCM	2	97	138	78	103	—	—	—	—
EPIC decoder	4	16	17	16	15	15	13	12	11
EPIC encoder	4	36	43	16	17	13	13	11	10
G721	2	57	66	19	18	—	—	—	—
GSM decoder	4	91	102	65	69	20	20	19	18
GSM encoder	4	46	57	41	48	20	21	19	17
JPEG decoder	2	101	111	61	68	—	—	—	—
JPEG encoder	3	46	55	31	32	28	32	—	—
MPEG2 decoder	4	32	31	24	29	15	15	11	10
MPEG2 encoder	4	31	39	30	37	20	22	18	16
PEGWIT	3	40	46	27	28	27	30	—	—

Tabela 5.1: CDFGs $G_i = (V_i, E_i), i = 1, \ldots, n$, de cada aplicação

para produzir o *datapath* resultante. Os CDFGs foram combinados iterativamente, do maior para o menor CDFG (em relação ao número de vértices). Alguns experimentos foram realizados com diferentes ordenações dos CDFGs e escolhendo diferentes cliques máximos. Embora a combinação em ordem decrescente de tamanho tenha produzido melhores resultados em muitos casos, não foram obtidas diferenças significativas no resultado para as diferentes ordenações nem para os diferentes cliques.

5.6.1 Comparação com Emparelhamento Máximo

Em um primeiro experimento, o algoritmo de Clique Máximo foi comparado com a técnica de combinação de CDFGs mais comumente utilizada. O método de combinação por emparelhamento de peso máximo em grafo bipartido (referido nesta seção como **Emparelhamento Máximo** e descrito em [47] e na Subseção 4.3.1) foi implementado e também aplicado iterativamente aos CDFGs em ordem decrescente de tamanho. Os resultados foram comparados em relação tanto à área de interconexões do *datapath* produzido quanto ao tempo de execução dos algoritmos. A Tabela 5.2 mostra o número de vértices e arcos do grafo combinado produzido e o tempo de execução das duas heurísticas. Mostra também a porcentagem de redução de interconexões obtida com o algoritmo de Clique Máximo em relação ao método de Emparelhamento Máximo.

A técnica de Clique Máximo produziu resultados melhores para todas as aplicações, com grafos combinados com área de interconexões até 35,8% menor que o método de

Aplicação	Emparelhamento Máximo $\bar{G}'$		Tempo de execução	Clique Máximo $\bar{G}$		Tempo de execução	Redução de interconexões
	$\|\bar{V}'\|$	$\|\bar{E}'\|$		$\|\bar{V}\|$	$\|\bar{E}\|$		
ADPCM	108	221	0,3 s	108	168	1,5 s	24,0%
EPIC decoder	24	48	0,8 s	24	33	3,9 s	31,3%
EPIC encoder	39	71	0,8 s	39	61	4,0 s	14,1%
G721	57	84	0,7 s	57	70	1,3 s	16,7%
GSM decoder	92	187	1,1 s	92	120	4,2 s	35,8%
GSM encoder	48	108	1,2 s	48	72	3,9 s	33,3%
JPEG decoder	104	171	0,3 s	104	137	1,7 s	19,9%
JPEG encoder	47	95	0,6 s	47	71	2,5 s	25,3%
MPEG2 decoder	34	70	0,8 s	34	51	4,0 s	27,1%
MPEG2 encoder	32	80	1,1 s	32	53	15,8 s	33,8%
PEGWIT	41	82	0,6 s	41	60	2,3 s	26,8%
Média			0,8 s			4,1 s	26,2%

Tabela 5.2: Comparação com técnica de Emparelhamento Máximo

Emparelhamento Máximo. Em média a redução da área de interconexões foi de 26,2%. Para construir estas soluções, o algoritmo de Clique Máximo gastou menos de 5 segundos para cada aplicação, com exceção da aplicação "MPEG encoder", que consumiu 15,8 segundos. O tempo de execução médio da heurística de Emparelhamento Máximo foi de 0,8 segundos, em contraste com o tempo médio de 4,1 segundos da abordagem por Clique Máximo. Ambas as abordagens mostraram-se eficientes em termos de tempo de execução, apesar do método de Emparelhamento Máximo ter sido mais rápido. Porém, ele produziu resultados muito piores em relação à área de interconexões, logo não é uma boa escolha quando há restrições de área para o *datapath*.

5.6.2 Análise de Erro das Soluções

O objetivo do segundo experimento foi comparar o número de interconexões $|\bar{E}|$ do grafo combinado $\bar{G}$ produzido pelo nosso algoritmo com seus limites inferior e superior. O limite inferior de $|\bar{E}|$ é definido como o número máximo de interconexões de um CDFG, dentre todos os CDFGs de entrada. O limite superior de $|\bar{E}|$ é definido como a soma do número de interconexões de todos os CDFGs. Assim, para uma aplicação com n CDFGs de entrada $G_i = (V_i, E_i), i = 1, \ldots, n$, tem-se:

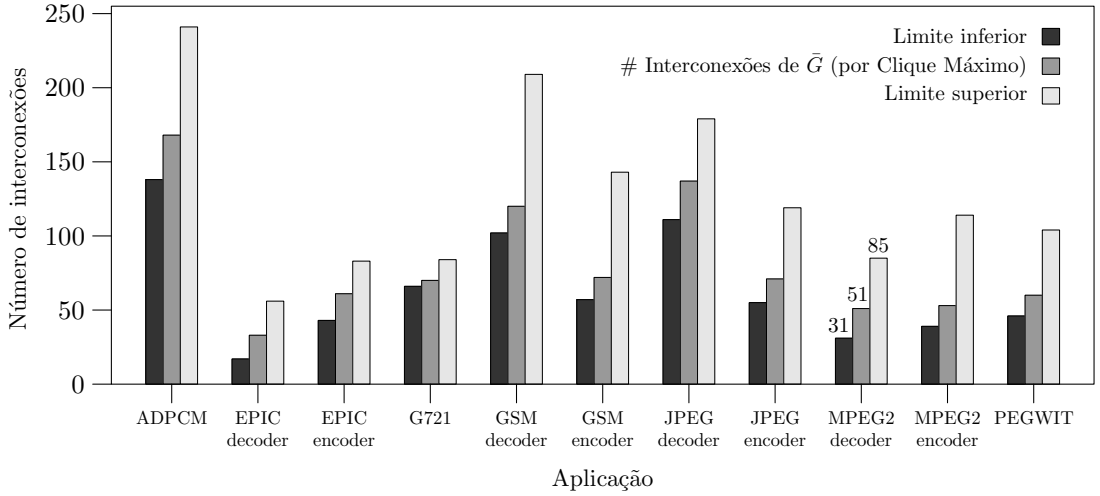


Figura 5.4: Número de interconexões de $\bar{G}$ e seus limites inferior e superior

$$\textit{Limite inferior de } |\bar{E}| = \max_{i=1}^{n} |E_i| \tag{5.1}$$

$$\textit{Limite superior de } |\bar{E}| = \sum_{i=1}^{n} |E_i| \tag{5.2}$$

Note que o limite inferior é altamente otimista, enquanto que o limite superior é muito pessimista. O limite inferior representa uma combinação em que todos os CDFGs são totalmente sobrepostos a um deles, isto é, nenhuma interconexão extra é necessária. O limite superior representa a situação em que os CDFGs são combinados sem nenhum compartilhamento de interconexões. A Figura 5.4 mostra o número de interconexões dos grafos combinados produzidos pela heurística de Clique Máximo, assim como os limites inferior e superior destas interconexões, para as aplicações do MediaBench. Para todas as aplicações, o número de interconexões do grafo combinado aproximou-se mais do seu limite inferior do que do limite superior.

Um modelo de programação inteira (PI) para o problema de combinação de CDFGs é apresentado em [73, 74] e tem como objetivo minimizar o número de interconexões do grafo combinado. Este modelo trata globalmente a combinação de n CDFGs, $n \geq 2$, e permite a obtenção de um limite inferior (mais preciso do que aquele da Equação 5.1) para o número de interconexões. Para aplicações com poucos e pequenos CDFGs, o modelo também permite encontrar soluções ótimas. Usando estes limites inferiores, o erro máximo das soluções construídas pelo nosso algoritmo de Clique Máximo é calculado,

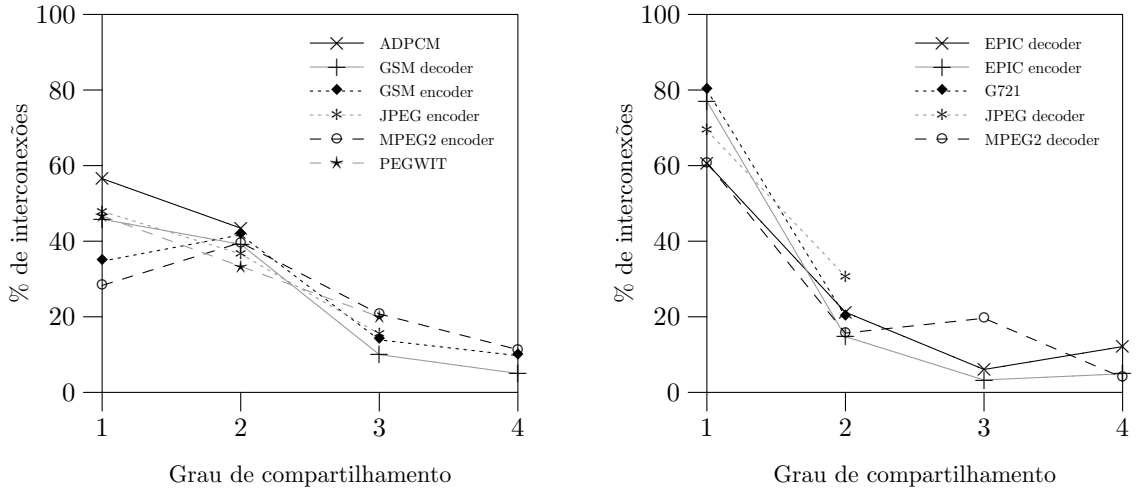
Aplicação	Clique Máximo $\lvert\bar{E}\rvert$	Prog. Inteira Limite inferior	Erro máximo
ADPCM	168	168	0,0%
EPIC decoder	33	33	0,0%
EPIC encoder	61	59	3,4%
G721	70	70	0,0%
GSM decoder	120	113	6,2%
GSM encoder	72	69	4,3%
JPEG decoder	137	127	7,9%
JPEG encoder	71	65	9,2%
MPEG2 decoder	51	48	6,3%
MPEG2 encoder	53	49	8,2%
PEGWIT	60	60	0,0%
Média			4,1%

Tabela 5.3: Limite inferior pelo modelo de PI e erro máximo do Clique Máximo

como mostrado na Tabela 5.3. É importante ressaltar que a análise deste modelo de PI consome um tempo de execução exponencial.

Os resultados da Tabela 5.3 mostram que a nossa heurística produziu soluções muito boas. Para 4 das 11 aplicações, a solução ótima foi encontrada, isto é, o grafo combinado produzido possui um número mínimo de interconexões (erro máximo de 0,0%). Para todas as demais aplicações, o erro máximo foi inferior a 10%, sendo em média 4,1%. Dentre estas 7 aplicações, a solução ótima foi obtida pelo modelo de PI apenas para a aplicação "EPIC encoder", e o erro de apenas 3,4% do algoritmo de Clique Máximo é exato. Para as aplicações restantes, o limite inferior do modelo de PI forneceu apenas o erro máximo. Por exemplo, para a aplicação "JPEG encoder", os resultados do modelo de PI informaram que não existe, para esta aplicação, grafo combinado com menos de 65 interconexões. Porém, eles não garantiram que existe um grafo combinado com este número de interconexões. Ou seja, a solução produzida pelo algoritmo proposto possui no máximo 9,2% mais interconexões que a solução ótima.

Esses resultados também reforçam a observação de que os limites inferiores da Figura 5.4 (calculados através da Equação 5.1) são extremamente otimistas. Por exemplo, pela Equação 5.1, o limite inferior para a aplicação "MPEG decoder" é 31, porém pelo modelo de PI é 48.



(a) (b)

Figura 5.5: Grau de compartilhamento das interconexões de $\bar{G}$

5.6.3 Grau de Compartilhamento das Interconexões

Em um último experimento foi avaliada a eficiência do algoritmo de Clique Máximo em maximizar o compartilhamento de interconexões entre os CDFGs de uma aplicação. Dado o grafo combinado obtido com a combinação de n CDFGs, determinou-se a porcentagem das interconexões deste grafo que resultam da sobreposição de q interconexões, para cada $q = 1, \ldots, n$. Esta medida reflete não apenas a habilidade do algoritmo em maximizar o compartilhamento de interconexões, mas também o quanto similar a estrutura dos CDFGs são. A Figura 5.5 mostra os resultados.

Para as aplicações da Figura 5.5(b), 60% a 80% das interconexões do grafo combinado são resultado de nenhuma sobreposição, isto é, não são compartilhadas. E à medida que o grau de compartilhamento (eixo horizontal na figura) cresce, a porcentagem de interconexões do grafo combinado com aquele grau de compartilhamento diminui quase que exponencialmente.

Note que pela Figura 5.5(b), 80% das interconexões do grafo combinado da aplicação "G.721" não são compartilhadas. Porém, pela Tabela 5.3 este grafo combinado é ótimo, isto é, ele possui o número mínimo de interconexões. Portanto, a eficiência de compartilhamento para esta aplicação foi baixa porque seus CDFGs são muito diferentes estruturalmente.

Capítulo 6

Otimização da Área Total

Neste capítulo nós estendemos a heurística para o problema de combinação de CDFGs apresentada no Capítulo 5, de modo a considerar tanto a área de blocos lógicos quanto de interconexões do *datapath* resultante. Este novo algoritmo, apresentado em [60], tem portanto o objetivo de minimizar a área total $A(\bar{G})$ do grafo combinado $\bar{G}$. O algoritmo realiza ambas as tarefas de seleção e atribuição de unidades e a atribuição de blocos lógicos e interconexões é feita simultaneamente. Ele também combina dois CDFGs a cada iteração e é aplicado de forma incremental para combinar vários CDFGs.

A Seção 6.1 descreve as estruturas auxiliares utilizadas no algoritmo proposto, que é apresentado na Seção 6.2. A Seção 6.3 apresenta uma avaliação experimental deste algoritmo.

6.1 Grafo de Compatibilidade e Clique de Peso Máximo

Inicialmente, dados dois CDFGs de entrada G_1 e G_2, o grafo de compatibilidade é construído. Porém, neste novo algoritmo ele representa todos os possíveis mapeamentos de vértices e arcos entre os CDFGs, além da coerência entre estes mapeamentos. Assim, o grafo de compatibilidade e o conceito de compatibilidade entre mapeamentos são redefinidos a seguir, de forma a incluir o mapeamento de vértices.

Definição 12 *O grafo de compatibilidade, correspondente aos CDFGs* $G_1 = (V_1, E_1)$ *e* $G_2 = (V_2, E_2)$*, é um grafo não direcionado* $G_c = (V_c, E_c)$ *com pesos nos vértices, tal que:*

- *Cada vértice $v_c \in V_c$ com peso w_c corresponde a:*
  - *um possível mapeamento v_1/v_2 de vértices $v_1 \in V_1$ e $v_2 \in V_2$, tal que $B(v_1) \cap B(v_2) \neq \emptyset$; ou*
  - *um possível mapeamento $(u_1, v_1, p_1)/(u_2, v_2, p_2)$ de arcos $(u_1, v_1, p_1) \in E_1$ e $(u_2, v_2, p_2) \in E_2$, tal que $B(u_1) \cap B(u_2) \neq \emptyset$, $B(v_1) \cap B(v_2) \neq \emptyset$, e p_1 e p_2 são equivalentes.*
- *Existe uma aresta $e_c = (u_c, v_c) \in E_c$ se os mapeamentos representados por u_c e v_c são compatíveis.*

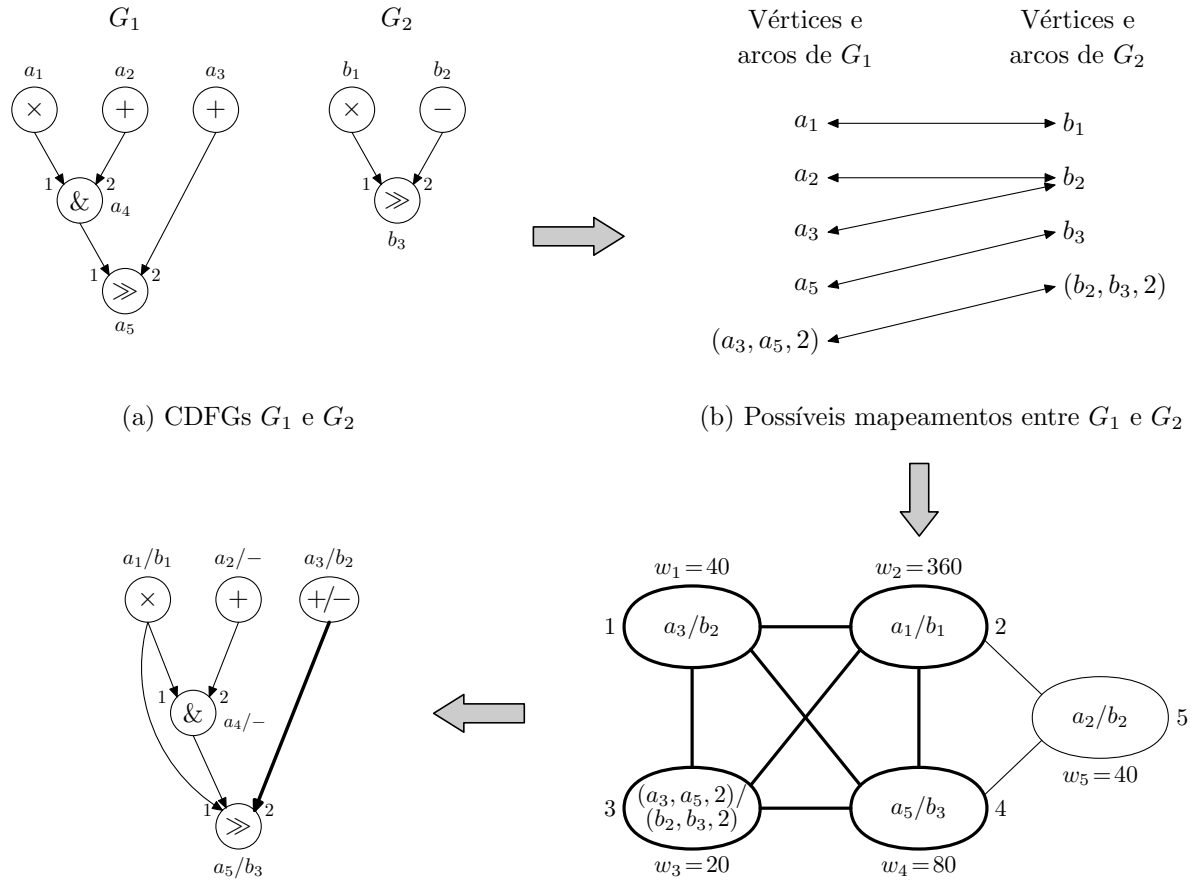
Definição 13 *Dois mapeamentos de vértices ou arcos dos CDFGs $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ não são compatíveis se eles mapeiam o mesmo vértice de V_1 em dois vértices distintos de V_2, ou vice-versa.*

Os pesos dos vértices do grafo de compatibilidade são determinados utilizando o conceito de redução de área, conforme a definição abaixo. Se dois vértices v_1 e v_2 são mapeados, haverá no *datapath* resultante apenas um bloco lógico capaz de realizar as operações representadas por ambos v_1 e v_2, ao invés de haver dois blocos lógicos para realizar cada operação, respectivamente. A redução de área proporcionada pelo mapeamento v_1/v_2 reflete esta diferença. Quando dois arcos são mapeados, ao invés de haver no *datapath* resultante duas interconexões, haverá apenas uma e não será necessário um MUX na porta de entrada do bloco lógico destino. A redução de área obtida com este mapeamento corresponde à área equivalente a uma entrada de MUX da largura adequada.

Definição 14 *O peso w_c do vértice $v_c \in V_c$ é a redução de área obtida com o mapeamento representado por v_c, onde:*

- *se v_c representa um mapeamento de vértices v_1/v_2, então $w_c = A_b(v_1) + A_b(v_2) - A_b(v_1/v_2)$;*
- *se v_c representa um mapeamento de arcos $(u_1, v_1, p_1)/(u_2, v_2, p_2)$, então $w_c = A_{mux}$.*

O grafo de compatibilidade é então construído com base nas definições apresentadas acima. A Figura 6.1 ilustra todos os passos deste segundo algoritmo proposto. Dados os CDFGs G_1 e G_2 da Figura 6.1(a), os vértices e arcos de G_1 e G_2 que podem ser sobrepostos são listados na Figura 6.1(b). A Figura 6.1(c) mostra o grafo de compatibilidade G_c



(a) CDFGs G_1 e G_2

(b) Possíveis mapeamentos entre G_1 e G_2

(d) Grafo combinado $\bar{G}$

(c) Clique de peso máximo do grafo de compatibilidade G_c

Figura 6.1: Passos do segundo algoritmo de combinação de CDFGs

obtido, representando os possíveis mapeamentos de vértices e arcos entre os dois CDFGs. O peso dos vértices de G_c representa a redução de área dos mapeamentos correspondentes. Por exemplo, o peso w_1 do mapeamento de vértices a_3/b_2 corresponde a redução de área obtida por se ter apenas um bloco lógico somador/subtrador, ao invés de um somador e um subtrador. Este algoritmo de combinação de CDFGs também explora a comutatividade das operações, durante a construção do grafo de compatibilidade.

Para encontrar o grafo combinado com área total mínima, é necessário determinar o conjunto de mapeamentos de vértices e arcos que são compatíveis entre si e somam o maior peso. Isto é obtido encontrando-se um clique de peso máximo de G_c, definido a seguir.

Definição 15 *Um clique de peso máximo de um grafo $G_c = (V_c, E_c)$, onde cada $v_c \in V_c$*

possui peso w_c, é um conjunto de vértices $C \subseteq V_c$ tal que, para todos os vértices $u_c, v_c \in C$, a aresta $(u_c, v_c) \in E_c$, e $\sum_{\forall v_c \in C} w_c$ é máximo.

Os vértices do clique de peso máximo C do grafo de compatibilidade G_c fornecem os mapeamentos de vértices e arcos que devem ser utilizados na construção do grafo combinado $\bar{G}$. Assim, blocos lógicos e interconexões são mapeados simultaneamente. Dado que o clique de peso máximo representa os mapeamentos compatíveis entre si que fornecem a maior redução em área, o grafo combinado $\bar{G}$ produzido é tal que $A(\bar{G})$ é mínimo. Por exemplo, o clique de peso máximo do grafo de compatibilidade da Figura 6.1(c) possui os vértices 1, 2, 3 e 4, e é mostrado em negrito na figura.

O grafo combinado $\bar{G}$ é então construído em uma seqüência de passos, a partir de um grafo vazio, com base nos mapeamentos indicados por C. Inicialmente, para cada vértice $v_c \in C$ representando o mapeamento dos vértices v_1 de G_1 e v_2 de G_2 , um vértice $\bar{v}$ é inserido em $\bar{G}$ (se ainda não o foi), correspondendo ao mapeamento de vértices v_1/v_2. Para cada vértice $v_c \in C$ representando o mapeamento dos arcos (u_1, v_1, p_1) de G_1 e (u_2, v_2, p_2) de G_2 , dois vértices $\bar{u}$ e $\bar{v}$ são inseridos em $\bar{G}$ (se ainda não o foram), correspondendo aos mapeamentos de vértices u_1/u_2 e v_1/v_2, respectivamente. Além disso, o arco $\bar{e} = (\bar{u}, \bar{v}, \bar{p})$ é também inserido em $\bar{G}$, correspondendo aos arcos mapeados representados por v_c. Se v_c representa um mapeamento de arcos obtido através da comutação de operandos, as portas de entrada de $\bar{v}$ em $\bar{G}$ são modificadas apropriadamente.

Por fim, como o clique de peso máximo indica todos os mapeamentos de vértices e arcos de $\bar{G}$, todos os vértices de G_1 e G_2 que ainda não foram mapeados, são inseridos em $\bar{G}$ como vértices não mapeados. Analogamente, todos os arcos de G_1 e G_2 que não pertencem a nenhum mapeamento em C, são também inseridos em $\bar{G}$ como arcos não mapeados, conectando os vértices correspondentes.

Por exemplo, utilizando o clique de peso máximo da Figura 6.1(c), o grafo combinado $\bar{G}$, mostrado na Figura 6.1(d), é obtido. Note que se o mapeamento de arcos $(a_3, a_5, 2)/(b_2, b_3, 2)$ não fosse incluído no grafo de compatibilidade e apenas mapeamentos de vértices fossem considerados, o clique de peso máximo poderia ser o conjunto de vértices 2, 4 e 5 de G_c, resultando no grafo combinado $\bar{G}''$ mostrado na Figura 4.4 que possui uma área total maior que $\bar{G}$.

```
CombinaCDFGs
    ENTRADA: n CDFGs G_i = (V_i, E_i), i = 1...n
    SAÍDA: grafo combinado Ḡ = (V̄, Ē),
        tal que A_i(Ḡ) é mínimo
    /* Inicialmente, Ḡ é o CDFG de entrada G_1 */
    Ḡ ← G_1;
    /* Iterativamente combina Ḡ com CDFG G_i */
    PARA i ← 2 ATÉ n FAÇA
        G_c ← ConstróiGrafoCompatibilidade(Ḡ, G_i);
        C ← EncontraCliquePesoMáximo(G_c);
        Ḡ ← ReconstróiGrafoCombinado(Ḡ, G_i, C);
```

Algoritmo 6.1: Combinação de CDFGs para minimizar área total

6.2 Algoritmo de Combinação de CDFGs

Nossa segunda heurística para combinação de CDFGs é apresentada no Algoritmo 6.1. Os passos de construção do grafo de compatibilidade e do grafo combinado são detalhados nos Algoritmos 6.2 e 6.3.

Como o problema de clique de peso máximo também é NP-completo [33], uma heurística de tempo polinomial é aplicada. O algoritmo utilizado para encontrar o clique de peso máximo é baseado em uma técnica de *branch-and-bound* [63, 62] e ajustado de forma que seu tempo de execução seja limitado polinomialmente por $|V_c|$. A complexidade temporal dos passos do algoritmo de combinação de CDFGs proposto neste capítulo é a mesma do algoritmo apresentado no Capítulo 5.

6.3 Resultados Experimentais

Nós realizamos um conjunto de experimentos com o objetivo de avaliar a abordagem de combinação de CDFGs e a segunda heurística que propusemos. Novamente, um conjunto de programas do *benchmark* MediaBench foi utilizado. Os CDFGs foram gerados de forma semelhante à descrita na Seção 5.6, sendo que para cada aplicação foram considerados até oito laços internos (se disponíveis). Para cada aplicação, os CDFGs foram combinados iterativamente, do maior para o menor, em relação ao seu número de vértices. As informações sobre os CDFGs gerados são mostradas na Tabela 6.1.

```
ConstróiGrafoCompatibilidade(G_i, G_j)
    /* Inicialmente, G_c é vazio */
    V_c ← ∅; E_c ← ∅;
    /* Determina vértices de G_c que são mapeamentos de vértices */
    PARA cada v_i ∈ V_i FAÇA
        PARA cada v_j ∈ V_j FAÇA
            SE B(v_i) ∩ B(v_j) ≠ ∅ ENTÃO
                V_c ← V_c ∪ {v_c}, onde v_c = v_i/v_j;
    /* Determina vértices de G_c que são mapeamentos de arcos */
    PARA cada e_i = (u_i, v_i, p_i) ∈ E_i FAÇA
        PARA cada e_j = (u_j, v_j, p_j) ∈ E_j FAÇA
            SE B(u_i) ∩ B(u_j) ≠ ∅ E B(v_i) ∩ B(v_j) ≠ ∅ E p_i e p_i são equivalentes ENTÃO
                V_c ← V_c ∪ {v_c}, onde v_c = e_i/e_j;
    /* Determina arestas de G_c */
    PARA cada u_c e v_c ∈ V_c, u_c ≠ v_c FAÇA
        SE u_c e v_c são compatíveis ENTÃO
            E_c ← E_c ∪ {e_c}, onde e_c = (u_c, v_c);
    RETORNA G_c = (V_c, E_c);
```

Algoritmo 6.2: Construção do grafo de compatibilidade G_c

6.3.1 Comparação das Técnicas de Combinação de CDFGs

Para avaliar o nosso algoritmo em relação a outros métodos de combinação de CDFGs, três técnicas baseadas em trabalhos relacionados foram implementadas. As soluções obtidas com estas técnicas foram comparadas com as produzidas pela nossa heurística, em relação à área total do *datapath* resultante e ao tempo de execução do algoritmo. Nos experimentos descritos nesta subseção, as informações de área utilizadas foram obtidas de uma biblioteca de componentes.

Os métodos implementados foram o emparelhamento de peso máximo em grafo bipartido, na forma descrita em [34] (referido nesta subseção como **Emparelhamento Máximo**), a busca local por melhorias sucessivas [84, 34] (referida como **Melhorias Sucessivas**) e o modelo de programação inteira descrito em [34] (referido como **Programação Inteira**), todos eles descritos na Seção 4.3. Nossa heurística, baseada no clique de peso máximo do grafo de compatibilidade, é referida como **Clique Compatível**.

ReconstróiGrafoCombinado(G_i, G_j, C)

`AUXILIAR:` funções de mapeamento de vértices $\mathcal{V}_i : V_i \rightarrow \bar{V} \cup \{-1\}$ e $\mathcal{V}_j : V_j \rightarrow \bar{V} \cup \{-1\}$;
funções de mapeamento de arcos $\mathcal{E}_i : E_i \rightarrow \{0, 1\}$ e $\mathcal{E}_j : E_j \rightarrow \{0, 1\}$

/* Inicialmente, $\bar{G}$ é vazio e todos os vértices e arcos de G_i e G_j não estão mapeados em $\bar{G}$ */
$\bar{V} \leftarrow \emptyset$; $\bar{E} \leftarrow \emptyset$;
$\mathcal{V}_i(v_i) \leftarrow -1, \forall v_i \in V_i$; $\mathcal{V}_j(v_j) \leftarrow -1, \forall v_j \in V_j$;
$\mathcal{E}_i(e_i) \leftarrow 0, \forall e_i \in E_i$; $\mathcal{E}_j(e_j) \leftarrow 0, \forall e_j \in E_j$;
/* Trata mapeamentos indicados em C */
`PARA` cada $v_c \in V_c$ `FAÇA`
  /* Se v_c representa um mapeamento de vértices */
  `SE` $v_c = v_i/v_j, v_i \in V_i$ e $v_j \in V_j$ `FAÇA`
    `SE` $\mathcal{V}_i(v_i) = -1$ `ENTÃO`
      $\bar{V} \leftarrow \bar{V} \cup \{\bar{v}\}$, onde $\bar{v} = v_i/v_j$; $\mathcal{V}_i(v_i) \leftarrow \bar{v}$; $\mathcal{V}_j(v_j) \leftarrow \bar{v}$;
  /* Senão, se v_c representa um mapeamento de arcos */
  `SENÃO SE` $v_c = e_i/e_j, e_i = (u_i, v_i, p_i) \in E_i$ e $e_j = (u_j, v_j, p_j) \in E_j$ `FAÇA`
    `SE` $\mathcal{V}_i(u_i) = -1$ `ENTÃO`
      $\bar{V} \leftarrow \bar{V} \cup \{\bar{u}\}$, onde $\bar{u} = u_i/u_j$; $\mathcal{V}_i(u_i) \leftarrow \bar{u}$; $\mathcal{V}_j(u_j) \leftarrow \bar{u}$;
    `SE` $\mathcal{V}_i(v_i) = -1$ `ENTÃO`
      $\bar{V} \leftarrow \bar{V} \cup \{\bar{v}\}$, onde $\bar{v} = v_i/v_j$; $\mathcal{V}_i(v_i) \leftarrow \bar{v}$; $\mathcal{V}_j(v_j) \leftarrow \bar{v}$;
    $\bar{E} \leftarrow \bar{E} \cup \{\bar{e}\}$, onde $\bar{e} = e_i/e_j = (\bar{u}, \bar{v}, \bar{p})$, com $\bar{p}$ definido apropriadamente;
    $\mathcal{E}_i(e_i) \leftarrow 1$; $\mathcal{E}_j(e_j) \leftarrow 1$;
/* Trata vértices não mapeados */
`PARA` cada $v_i \in V_i$ `FAÇA`
  `SE` $\mathcal{V}_i(v_i) = -1$ `ENTÃO`
    $\bar{V} \leftarrow \bar{V} \cup \{\bar{v}\}$, onde $\bar{v} = v_i/-$; $\mathcal{V}_i(v_i) \leftarrow \bar{v}$;
`PARA` cada $v_j \in V_j$ `FAÇA`
  `SE` $\mathcal{V}_j(v_j) = -1$ `ENTÃO`
    $\bar{V} \leftarrow \bar{V} \cup \{\bar{v}\}$, onde $\bar{v} = -/v_j$; $\mathcal{V}_j(v_j) \leftarrow \bar{v}$;
/* Trata arcos não mapeados */
`PARA` cada $e_i = (u_i, v_i, p_i) \in E_i$ `FAÇA`
  `SE` $\mathcal{E}_i(e_i) = 0$ `ENTÃO`
    $\bar{E} \leftarrow \bar{E} \cup \{\bar{e}\}$, onde $\bar{e} = e_i/- = (\mathcal{V}_i(u_i), \mathcal{V}_i(v_i), \bar{p})$, com $\bar{p}$ definido apropriadamente;
    $\mathcal{E}_i(e_i) \leftarrow 1$
`PARA` cada $e_j = (u_j, v_j, p_j) \in E_j$ `FAÇA`
  `SE` $\mathcal{E}_j(e_j) = 0$ `ENTÃO`
    $\bar{E} \leftarrow \bar{E} \cup \{\bar{e}\}$, onde $\bar{e} = -/e_j = (\mathcal{V}_j(u_j), \mathcal{V}_j(v_j), \bar{p})$, com $\bar{p}$ definido apropriadamente;
    $\mathcal{E}_j(e_j) \leftarrow 1$
`RETORNA` $\bar{G} = (\bar{V}, \bar{E})$;

Algoritmo 6.3: Construção do grafo combinado $\bar{G}$

Aplicação	n	G_1		G_2		G_3		G_4		G_5		G_6		G_7		G_8	
		$\|V_1\|$	$\|E_1\|$	$\|V_2\|$	$\|E_2\|$	$\|V_3\|$	$\|E_3\|$	$\|V_4\|$	$\|E_4\|$	$\|V_5\|$	$\|E_5\|$	$\|V_6\|$	$\|E_6\|$	$\|V_7\|$	$\|E_7\|$	$\|V_8\|$	$\|E_8\|$
EPIC decoder	3	16	17	15	16	12	12	—	—	—	—	—	—	—	—	—	—
EPIC encoder	2	13	15	11	12	—	—	—	—	—	—	—	—	—	—	—	—
G721	2	57	68	19	20	—	—	—	—	—	—	—	—	—	—	—	—
GSM encoder	8	171	195	112	131	90	115	49	54	46	59	41	50	20	22	19	21
JPEG decoder	2	101	113	61	74	—	—	—	—	—	—	—	—	—	—	—	—
JPEG encoder	7	109	127	53	68	53	63	53	60	46	58	31	35	28	35	—	—
MPEG2 decoder	6	40	47	34	35	32	34	24	31	15	16	11	12	—	—	—	—
MPEG2 encoder	6	45	57	35	38	31	41	30	39	20	23	18	21	—	—	—	—
PEGWIT	7	75	85	75	85	75	85	75	85	58	66	40	47	27	31	—	—

Tabela 6.1: CDFGs $G_i = (V_i, E_i), i = 1, \ldots, n$, de cada aplicação

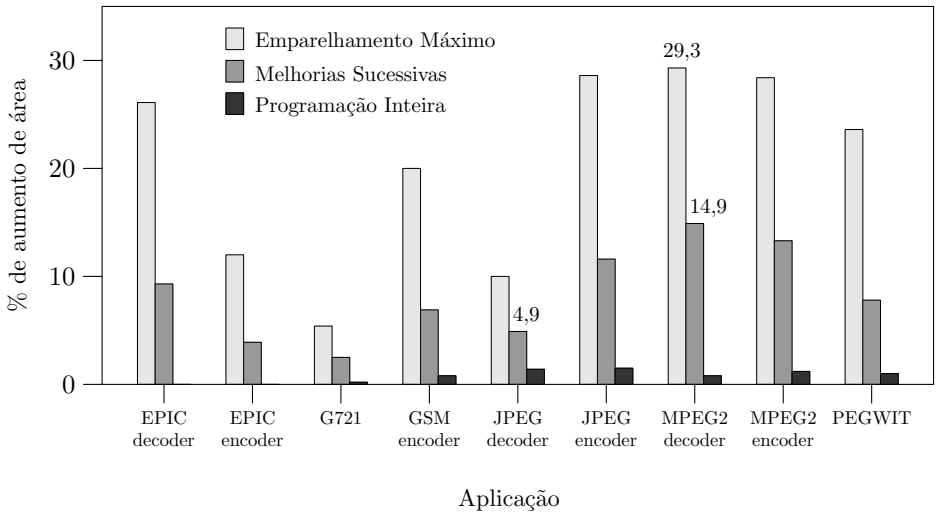


Figura 6.2: Aumento de área das demais técnicas em relação a Clique Compatível (barras ausentes representam um aumento de área de 0%)

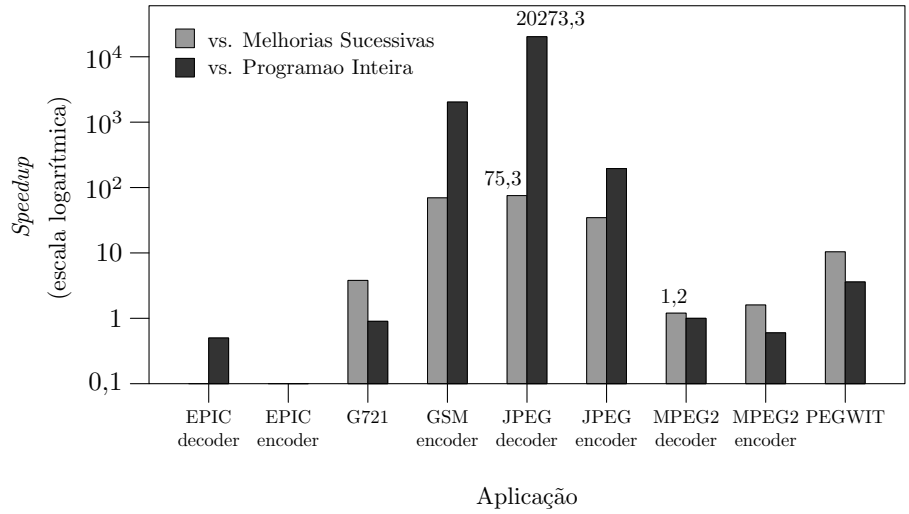


Figura 6.3: *Speedup* de Clique Compatível (barras ausentes representam *speedup* de 0.1)

A Figura 6.2 mostra, para cada aplicação, a porcentagem de aumento de área do grafo combinado produzido pelas demais técnicas, em relação à heurística de Clique Compatível. Além disso, o *speedup* do tempo de execução obtido com este método, em comparação ao demais, é mostrado na Figura 6.3 (note a escala logarítmica).

Algoritmo	Aumento de área médio vs. Clique Compatível	*Speedup* médio de Clique Compatível
Emparelhamento Máximo	20,4%	0,1
Melhorias Sucessivas	8,4%	21,8
Programação Inteira	0,8%	2500,4

Tabela 6.2: Comparação do método de Clique Compatível com outros algoritmos

Note que o Emparelhamento Máximo produziu grafos combinados com área até 30% maior do que o nosso algoritmo. O método de Melhorias Sucessivas produziu um aumento de área de 2,5% a 14,9%, quando comparado com o Clique Compatível, e apresenta um tempo de execução bem maior que este algoritmo. Por exemplo, para a aplicação "JPEG decoder", o método de Melhorias Sucessivas produziu um aumento de área de apenas 4,9%, porém levou 75,3 vezes mais tempo para executar. Para a aplicação "MPEG2 decoder", seu tempo de execução foi apenas 20% maior que o Clique Compatível, mas o aumento de área foi de 14,9%. A técnica de Programação Inteira foi quase equivalente à nossa heurística no que diz respeito à área, mas seu tempo de execução foi até 20 mil vezes maior que o algoritmo de Clique Compatível. Para algumas aplicações, o Clique Compatível produziu inclusive grafos combinados com área menor do que aqueles da técnica de Programação Inteira. Embora esta técnica forneça a solução ótima para a combinação de dois CDFGs, ela é apenas uma heurística para a combinação incremental de vários CDFGs. Além disso, este método não explora a comutatividade das operações, enquanto que o algoritmo de Clique Compatível sim.

A Tabela 6.2 resume os resultados das Figuras 6.2 e 6.3 e mostra o aumento médio de área dos demais métodos em relação ao algoritmo de Clique Compatível, e o *speedup* médio obtido por este algoritmo em relação aos outros. A técnica de Emparelhamento Máximo foi muito rápida (tempo de execução médio de 0,6 segundos em contraste com 4,8 segundos do Clique Compatível), porém ela produziu resultados muito ruins para a área do *datapath* (aumento de área médio de 20,4%). O método de Melhorias Sucessivas obteve um aumento de área médio de 8,4% e levou 21,8 vezes mais tempo que o Clique Compatível. Finalmente, as áreas produzidas pela Programação Inteira foram ligeiramente piores que aquelas obtidas com o algoritmo proposto, mas seu tempo de execução exponencial é proibitivo.

Em um outro experimento foi analisado o comportamento das técnicas de combinação quando o número de CDFGs de entrada cresce. Para cada aplicação, foram inicialmente

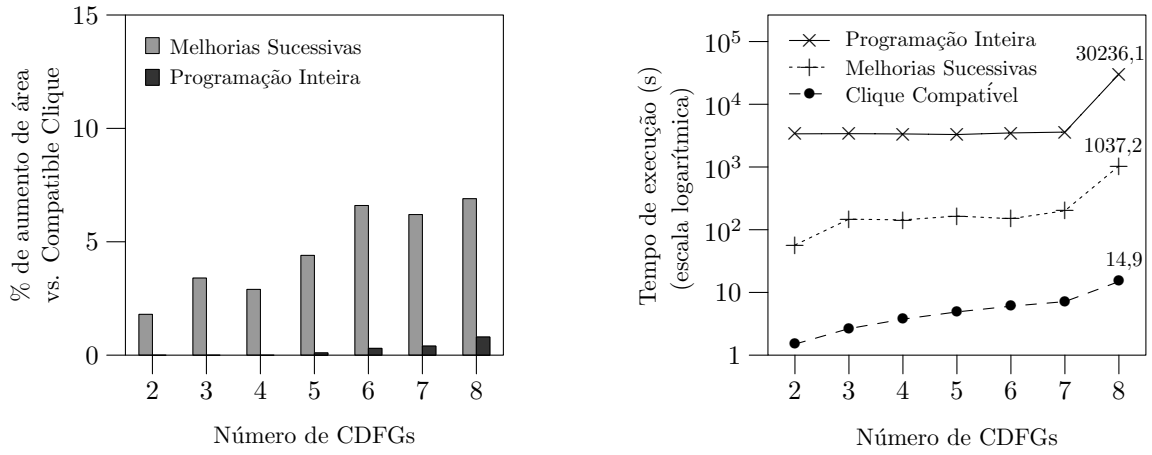


Figura 6.4: Aumento de área e tempo de execução, para "GSM encoder" (barras ausentes representam aumento de área de 0%)

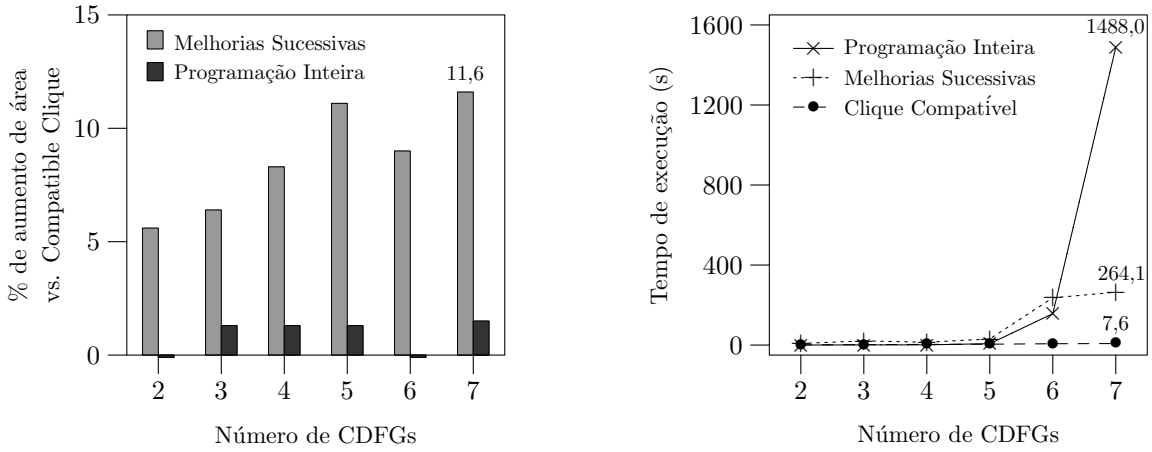


Figura 6.5: Aumento de área e tempo de execução, para "JPEG encoder" (barras invertidas representam aumento de área de -0,1%)

combinados os dois CDFGs mais relevantes (isto é, que correspondem aos dois laços que contribuem mais para o tempo de execução do programa). Em seguida, os três CDFGs mais relevantes foram combinados, e assim por diante. As Figuras 6.4 e 6.5 mostram estes resultados para as aplicações "GSM encoder" e "JPEG encoder", respectivamente. Para ambas as aplicações, o tempo de execução do método de Programação Inteira realmente cresceu exponencialmente e o de Melhorias Sucessivas também cresceu muito rápido, a medida que o número de CDFGs aumentou. O tempo de execução da heurística de Clique Compatível cresceu polinomialmente.

6.3.2 Avaliação da Abordagem de Combinação de CDFGs

Uma questão relevante é se a abordagem de combinação de CDFGs é realmente necessária. Os CDFGs podem ser simplesmente agrupados em um CDFG completo, apenas multiplexando as suas entradas e saídas. Então este CDFG completo é submetido a ferramentas de síntese comerciais para que elas explorem o compartilhamento de recursos. Portanto, é necessário saber se a abordagem de combinação de CDFGs pode prover algum ganho em área, quando comparada a esta abordagem baseada na ferramenta de síntese. Para avaliar esta questão, um novo experimento foi realizado.

Nesse experimento, os CDFGs de entrada foram combinados utilizando a nossa heurística proposta neste capítulo, e um *datapath* (referido como *datapath* combinado) foi gerado, a partir do grafo combinado. Também foi gerado um *datapath* (referido como *datapath* agrupado) a partir de um CDFG completo, agrupando todos os CDFGs de entrada, sem combiná-los. Estes dois *datapaths* foram então sintetizados usando as mesmas restrições de projeto e opções de otimização, e os resultados da síntese foram comparados.

Neste experimento foi utilizada a ferramenta de síntese *Synopsys Design Compiler* (DC) [78], a melhor ferramenta comercial de síntese de circuitos à época (2004). Foram utilizados três níveis de otimização: otimização arquitetural, otimização ao nível lógico e ao nível de portas. Durante a otimização arquitetural, o DC explora o compartilhamento de sub-expressões comuns e de recursos. As restrições de projeto utilizadas determinaram a minimização da área do circuito (atribuindo `area_only` para as diretivas `set_resource_allocation` e `set_resource_implementation` e `0.0` para `set_max_area`). O processo de otimização também foi ajustado de forma a aplicar o esforço máximo nas fases de mapeamento e redução de área (atribuindo o valor `high` para as opções `-map_effort` e `-area_effort` do comando de otimização `compile`).

Este experimento é ilustrado com um exemplo. Dados os CDFGs G_0 e G_1 da Figura 6.6, os *datapaths* combinado e agrupado foram gerados. As Figuras 6.7 e 6.8 mostram, respectivamente, os *datapaths* agrupado e combinado, e o código VHDL correspondente a eles. Quando estes dois *datapaths* foram submetidos ao DC, o *datapath* combinado possibilitou uma redução de área de 27% em relação ao *datapath* agrupado.

Os relatórios de alocação e compartilhamento de recursos gerados pelo DC durante a síntese são mostrados na Figura 6.9. No *datapath* agrupado, as subtrações a_0 e b_0 de G_0 e G_1 (linhas 38 e 48 na Figura 6.7, respectivamente) foram mapeadas no mesmo subtrador

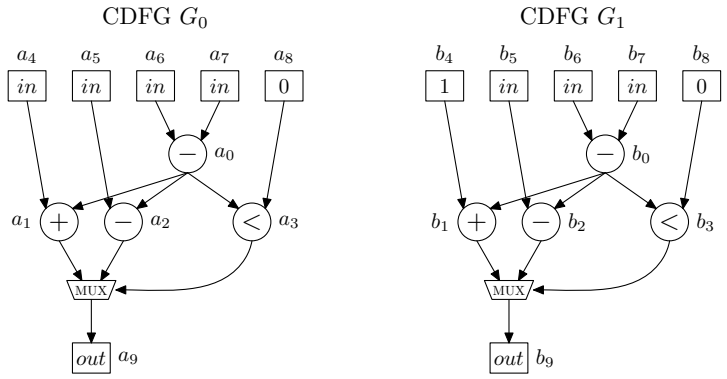


Figura 6.6: CDFGs G_0 e G_1

```
    ...
(37)  if id = '0' then -- CDFG 0
(38)     t0 := in2v - in3v ;
(39)     t1 := in0v + t0 ;
(40)     t2 := in1v - t0 ;
(41)     if t0 < zero32 then
(42)        t3 := t2 ;
(43)     else
(44)        t3 := t1 ;
(45)     end if ;
(46)     out0 <= t3 ;
(47)  else -- CDFG 1
(48)     t0 := in1v - in2v ;
(49)     t1 := one32 + t0 ;
(50)     t2 := in0v - t0 ;
(51)     if t0 < zero32 then
(52)        t3 := t2 ;
(53)     else
(54)        t3 := t1 ;
(55)     end if ;
(56)     out0 <= t3 ;
(57)  end if ;
    ...
```

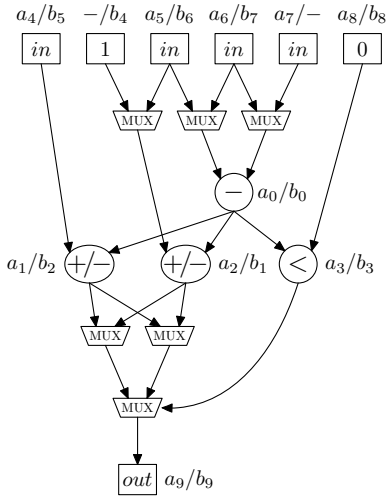


Figura 6.7: Código VHDL agrupando G_0 e G_1 e *datapath* agrupado gerado por DC

```
    ...
(37)  -- Merged CDFG
(38)  t0 := in2v - in3v ;
(39)  if id = '0' then
(40)     t1 := in0v + t0 ;
(41)  else
(42)     t1 := one32 + t0 ;
(43)  end if ;
(44)  t2 := in1v - t0 ;
(45)  if t0 < zero32 then
(46)     t3 := t2 ;
(47)  else
(48)     t3 := t1 ;
(49)  end if ;
(50)  out0 <= t3 ;
    ...
```

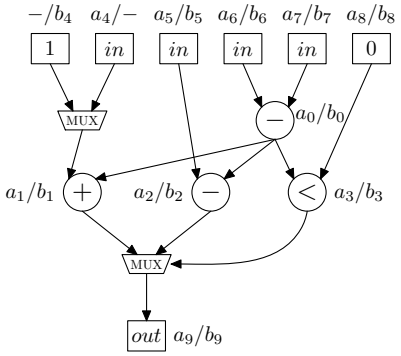


Figura 6.8: Código VHDL combinando G_0 e G_1 e *datapath* combinado gerado por DC

Resource	Module	Parameters	Contained Operations
r21	DW01_cmp2	width=32	lt_41/lt/lt
			lt_51/lt/lt
r22	DW01_addsub	width=32	add_39/plus/plus
			sub_50/minus/minus
r23	DW01_sub	width=32	sub_38/minus/minus
			sub_48/minus/minus
r79	DW01_addsub	width=32	add_49/plus/plus
			sub_40/minus/minus

(a) *Datapath* agrupado

Resource	Module	Parameters	Contained Operations
r20	DW01_cmp2	width=32	lt_45/lt/lt
r22	DW01_sub	width=32	sub_38/minus/minus
r70	DW01_add	width=32	add_40/plus/plus
			add_42/plus/plus
r73	DW01_sub	width=32	sub_44/minus/minus

(b) *Datapath* combinado

Figura 6.9: Relatórios de alocação e compartilhamento de recursos gerados por DC

(recurso r23 na Figura 6.9(a)). Todavia, a adição a_1 e subtração b_2 (linhas 39 e 50) foram mapeadas em um somador/subtrador (recurso r22), assim como a subtração a_2 e a soma b_1 (linhas 40 e 49), que foram mapeadas no recurso r79. No *datapath* combinado, também foi utilizado um subtrador para as operações mapeadas a_0/b_0 (linha 38 na Figura 6.8 e recurso r22 na Figura 6.9(b)). Porém, de forma diferente do *datapath* agrupado, a_1/b_1 (linhas 40 e 42) compartilharam o mesmo somador (recurso r70), e a_2/b_2 (linha 44) compartilharam um subtrador (recurso r73). Como resultado, o *datapath* agrupado usou dois blocos somador/subtrador para operações que no *datapath* combinado foram implementadas com apenas um somador e um subtrador. Além disso, menos MUXes foram necessários no *datapath* combinado, resultando na redução de área.

Pode ser observado que o DC é muito eficiente em mapear operações mutuamente exclusivas que recebem os mesmos operandos. Porém, ele pode introduzir MUXes adicionais caso estas operações possuam operandos diferentes. O algoritmo de combinação de CDFGs, por sua vez, mapeia operandos de entrada e saída, assim como operações. Assim, ele permite o mapeamento de operações exclusivas com operandos iguais ou diferentes, caso isto resulte em uma redução de área do *datapath* final.

A Tabela 6.3 mostra a área dos *datapaths* agrupado e combinado (obtido utilizando a nossa heurística), para as aplicações do MediaBench. Os *datapaths* foram submetidos à ferramenta de síntese Synopsys DC e as áreas foram medidas em unidades de área do DC. A redução de área obtida com a abordagem de combinação também é mostrada. Os resultados indicam que o algoritmo de combinação de CDFGs produziu *datapaths* com menor área, quando comparado com a abordagem de apenas agrupamento dos CDFGs.

Aplicação	Área do *datapath**		Redução
	agrupado	combinado	de área
EPIC decoder	15404	14634	5,0%
EPIC encoder	15760	14732	6,5%
G721	2755	1144	58,5%
GSM encoder	53702	49392	8,0%
JPEG decoder	6482	5417	16,4%
JPEG encoder	21985	19432	11,6%
MPEG2 decoder	5399	4304	20,3%
MPEG2 encoder	11291	8780	22,2%
PEGWIT	5280	3815	27,7%
Média			19,6%

*Em unidades de área do DC

Tabela 6.3: Comparação das abordagens de agrupamento e combinação de CDFGs

Capítulo 7

Conclusão

Este trabalho fez parte de um projeto maior que visa o estudo e desenvolvimento de sistemas dedicados e arquiteturas reconfiguráveis. Encerramos este texto apresentando os principais resultados e contribuições do trabalho e apontando possíveis trabalhos futuros que podem dar continuidade ao mesmo.

7.1 Resultados e Contribuições

Neste trabalho estruturamos as etapas do fluxo de projeto de um sistema dedicado, baseado em um modelo de arquitetura reconfigurável. Através da especialização do conjunto de instruções de um GPP, instruções especializadas foram criadas e implementadas em unidades funcionais específicas que, adicionadas ao hardware do GPP, formaram um processador especializado.

Formalizamos o problema de combinação de CDFGs para compartilhamento de recursos no projeto de um *datapath* reconfigurável. Este *datapath* implementa uma unidade funcional específica e é reconfigurado dinamicamente para realizar a computação correspondente a cada instrução especializada. Apresentamos uma demonstração de que o problema de combinação de CDFGs é NP-completo, realizando uma redução a partir do problema de isomorfismo de subgrafos. Tal demonstração não foi encontrada na literatura.

Propusemos, implementamos e avaliamos dois algoritmos heurísticos para o problema de combinação de CDFGs. Na avaliação dos algoritmos, implementamos outras soluções já propostas na literatura, para comparar seus resultados e desempenhos com as nossas soluções. Para isso, aplicamos nossas heurísticas e as demais soluções implementadas a

aplicações reais do *benchmark* MediaBench.

O primeiro algoritmo desenvolvido por nós é uma heurística incremental que busca minimizar a área das interconexões do *datapath*. O algoritmo realiza a tarefa de atribuição de unidades, supondo que a seleção das mesmas foi feita previamente, e realiza a atribuição de blocos lógicos e interconexões simultaneamente. Durante a atribuição de unidades, o algoritmo explora a comutatividade das operações para minimizar ainda mais a área de interconexões.

Conforme os resultados da avaliação experimental, o nosso algoritmo obteve uma redução média de 26,2% na área de interconexões do *datapath*, em relação ao método de Emparelhamento Máximo. O tempo de execução médio da heurística de Emparelhamento Máximo foi de 0,8 segundos, em contraste com o tempo médio de 4,1 segundos da nossa técnica. Apesar do método de Emparelhamento Máximo ter sido mais rápido, a nossa abordagem também mostrou-se eficiente. No entanto, dado que o método de Emparelhamento Máximo produziu resultados muito piores em relação à área de interconexões, ele não é uma boa escolha quando há restrições de área para o *datapath*.

Uma análise de erro máximo baseada em um modelo de PI mostrou que para algumas aplicações a nossa heurística produziu a solução ótima. Além disso, o erro máximo das soluções geradas foi bastante pequeno, em média 4,1%. Verificamos também que se os CDFGs combinados são muito diferentes estruturalmente, o grau de compartilhamento das interconexões do *datapath* produzido pode ser bastante baixo.

O segundo algoritmo proposto por nós visa minimizar a área total do *datapath*, que engloba a área de blocos lógicos e interconexões. O algoritmo realiza a seleção e atribuição de unidades, e a atribuição de blocos lógicos e interconexões é feita simultaneamente. Da mesma forma que a primeira heurística, este algoritmo é incremental e explora a comutatividade das operações.

Novamente, na avaliação experimental comparamos este segundo algoritmo com outros métodos. A técnica de Emparelhamento Máximo teve um tempo de execução médio de 0,6 segundos em contraste com 4,8 segundos da nossa heurística, porém ela produziu *datapaths* com área 20,4% maior em média. O método de Melhorias Sucessivas obteve um aumento de área médio de 8,4% e levou 21,8 vezes mais tempo que o nosso algoritmo. As áreas produzidas pelo método de Programação Inteira foram ligeiramente piores que aquelas obtidas com a nossa heurística, além de seu tempo de execução exponencial ser proibitivo.

Em outro experimento, comparamos a combinação de CDFGs com a abordagem em

que os CDFGs são agrupados em um CDFG completo que é submetido a uma ferramenta de síntese para que ela explore o compartilhamento de recursos. A nossa solução produziu *datapaths* com área menor. A ferramenta de síntese pode introduzir MUXes adicionais se as operações dos CDFGs possuem operandos diferentes. O algoritmo de combinação de CDFGs permitiu o mapeamento de operações mutuamente exclusivas com operandos possivelmente diferentes, quando isto resultava em uma redução na área do *datapath*.

7.2 Trabalhos Futuros

O problema de combinação de CDFGs é muito interessante e dá margem a diversas variações. Por ser NP-difícil, diferentes heurísticas podem ser aplicadas para resolvê-lo.

Uma primeira questão é que os dois algoritmos heurísticos propostos neste trabalho realizam a combinação incremental dos CDFGs. A busca de uma generalização destas heurísticas para que realizem a combinação global, porém mantendo um tempo de execução razoável, é uma possível extensão deste trabalho.

Outra idéia não explorada é o projeto de um algoritmo de combinação que, para obter soluções melhores, utilize características da estrutura dos CDFGs, como por exemplo o grau de entrada dos vértices ser, em geral, 1 ou 2. Além disso, uma questão interessante é a avaliação de quais características dos CDFGs fazem com que a combinação deles fique mais difícil, isto é, dificultam a obtenção de uma boa solução.

Dado que atualmente, o consumo de potência de um circuito é um critério de avaliação tão importante quanto área e desempenho, é possível projetar um algoritmo de combinação de CDFGs que tenha como objetivo a redução do consumo de potência do *datapath* produzido. Uma possível idéia é atribuir pesos aos arcos dos CDFGs representando uma estimativa do número de transições de bits nas interconexões correspondentes aos arcos. Os vértices do grafo de compatibilidade também podem possuir pesos representando a redução do consumo de potência obtida com aqueles mapeamentos.

Também é possível realizar a combinação de CDFGs utilizando informações de escalonamento dos mesmos ou a mobilidade[1] das operações dos CDFGs. O objetivo é não aumentar o caminho crítico do *datapath* ou o número total de ciclos consumido por ele.

Outro problema relacionado à alocação de recursos para um *datapath* é o que

[1]Intervalo entre os instantes de escalonamento ASAP (*As Soon As Possible*) e ALAP (*As Late As Possible*) de uma operação.

denominamos de compromisso área/desempenho. Uma das formas de obter um maior desempenho é explorar o paralelismo disponível na aplicação. Assim, mais operações são realizadas ao mesmo tempo, exigindo a existência de mais unidades funcionais no hardware e por conseqüência, aumentando sua área. Logo, os requisitos de minimizar área e maximizar desempenho são conflitantes e exigem, portanto, soluções de compromisso.

Dado um conjunto de aplicações, deseja-se construir um *datapath* capaz de executar cada aplicação, de forma multiplexada no tempo. Para cada aplicação existem várias implementações, que fornecem diferentes desempenhos e possuem diferentes demandas de recursos de hardware. O problema consiste em selecionar uma implementação para cada aplicação, de forma que requisitos de desempenho das aplicações e/ou área do *datapath* sejam satisfeitos.

Por fim, a interação entre os problemas de combinação de CDFGs e compromisso área/desempenho pode ser analisada.

Referências Bibliográficas

[1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley and Sons, 1989.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages (POPL)*, pages 177–189, January 1983.

[3] M. Arnold and H. Corporaal. Designing domain-specific processors. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES)*, pages 61–66, 2001.

[4] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference (DAC)*, pages 256–261, 2003.

[5] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4):610–637, Abril 2001.

[6] K. Bondalapati and V. Prasanna. Reconfigurable computing systems. *Proceedings of the IEEE*, 90(7):1201–1217, Julho 2002.

[7] E. Borin, F. Klein, N. Moreano, R. Azevedo, and G. Araujo. Fast instruction set customization. In *Proceedings of the Second Workshop on Embedded Systems for Real-Time Multimedia (ESTIMEDIA)*, pages 53–58, 2004.

[8] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 262–269, 2002.

[9] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of the Design Automation Conference (DAC)*, pages 395–400, 2004.

[10] M. Budiu and S. Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, pages 195–205, 1999.

[11] S. Cadambi, J. Weener, S. Goldstein, H. Schmit, and D. Thomas. Managing pipeline-reconfigurable FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, pages 55–64, 1998.

[12] T. Callahan. Kernel formation in garpcc. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–309, 2003.

[13] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, Abril 2000.

[14] T. Callahan and J. Wawrzynek. Instruction-level parallelism for reconfigurable computing. In *Lecture Notes in Computer Science – Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, volume 1482, pages 248–257. Springer-Verlag, 1998.

[15] J. Cardoso and M. Vestias. Architectures and compilers to support reconfigurable computing. *ACM Crossroads*, 5(3):15–22, Março 1999.

[16] P. Castro, E. Borin, R. Azevedo, and G. Araujo. Looking for instruction patterns in the design of extensible processors. In *Proceedings of the Workshop on Application Specific Processors (WASP)*, pages 69–75, September 2004.

[17] E. Chang and D. Gajski. A connection-oriented binding model for binding algorithms. Technical Report 96-49, ICS – University of California, Irvine, 1996.

[18] N. Cheung, S. Parameswaran, and J. Henkel. INSIDE: Instruction selection/identification & design exploration for extensible processors. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 291–297, 2003.

[19] H. Choi, J. Kim, C. Yoon, I. Park, S. Hwang, and C. Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, 48(6):603–614, 1999.

[20] N. Clark, H. Zhong, W. Tang, and S. Mahlke. Automatic design of application specific instruction set extensions through dataflow graph exploration. *International Journal of Parallel Programming*, 31(6):429–449, 2003.

[21] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, Junho 2002.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

[23] D. Cronquist, P. Franklin, S. Berg, and C. Ebeling. Specifying and compiling applications for RaPiD. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 116–125, 1998.

[24] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Proceedings of the Conference on Advanced Research in VLSI (ARVLSI)*, pages 23–41, 1999.

[25] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[26] A. DeHon. DPGA utilization and application. In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, pages 115–121, 1996.

[27] A. DeHon. The density advantage of configurable computing. *IEEE Computer*, pages 39–40, Abril 2000.

[28] A. DeHon and J. Wawrzynek. Reconfigurable computing: What, why, and implications for design automation. In *Proceedings of the Design Automation Conference (DAC)*, pages 610–615, 1999.

[29] S. Derrien and S. Rajopadhye. FCCMs and the memory wall. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 329–330, 2000.

[30] N. Dutt and K. Choi. Configurable processors for embedded computing. *IEEE Computer*, 36(1):120–123, Janeiro 2003.

[31] C. Ebeling, D. Cronquist, and P. Franklin. RaPiD – reconfigurable pipelined datapath. In *Lecture Notes in Computer Science – Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, volume 1142, pages 126–135. Springer-Verlag, 1996.

[32] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis – Introduction to Chip and System Design*. Kluwer Academic Pub., 1992.

[33] M.R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. Freeman and CO., 1979.

[34] W. Geurts, F. Catthoor, S. Vernalde, and H. De Man. *Accelerator Data-path Synthesis for High-Throughput Signal Processing Applications*. Kluwer Academic Publishers, 1997.

[35] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.Taylor. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):2–9, Abril 2000.

[36] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 28–39, 1999.

[37] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 137–147, 2003.

[38] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC)*, pages 3–14, December 2001. Disponível em http://www.eecs.umich.edu/mibench/.

[39] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE)*, pages 642–649, 2001.

[40] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, pages 65–74, 1998.

[41] S. Hauck. The future of reconfigurable systems. In *5th Canadian Conference on Field Programmable Devices (FPD)*, 1998. Keynote Address.

[42] S. Hauck and A. Agarwal. Software technologies for reconfigurable systems. Technical report, Dept. of ECE, Northwestern University, 1996.

[43] S. Hauck, T. Fry, M. Hosler, and J. Kao. The Chimaera reconfigurable functional unit. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 87–96, 1997.

[44] S. Hauck, Z. Li, and E. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 138–146, 1998.

[45] J. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 12–21, 1997.

[46] I. Huang and A. Despain. Synthesis of application specific instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 14(6):663–675, 1995.

[47] Z. Huang and S. Malik. Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE)*, pages 735–740, 2001.

[48] Z. Huang, S. Malik, N. Moreano, and G. Araujo. The design of dynamically reconfigurable datapath coprocessors. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(2):361–384, Maio 2004.

[49] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):605–627, 2002.

[50] S. Kumar, L. Pires, D. Pandalai, M. Vojta, J. Golusky, S. Wadi, and H. Spaanenburg. Benchmarking technology for configurable computing system. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 273–274, 1998.

[51] A. La Rosa, L. Lavagno, and C. Passerone. Hardware/software design space exploration for a reconfigurable processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 570–575, 2003.

[52] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 330–335, Dezembro 1997.

[53] M. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. Kurdahi, and E. Filho. Design and implementation of the MorphoSys reconfigurable computing processor. *Journal of VLSI Signal Processing – Systems for Signal, Image and Video Technology*, 24(2-3):147–164, Março 2000.

[54] Z. Li, K. Compton, and S. Hauck. Configuration caching management techniques for reconfigurable computing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 22–36, 2000.

[55] W. Luk, N. Shirazi, S. Guo, and P. Cheung. Pipeline morphing and virtual pipelines. In *Lecture Notes in Computer Science – Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, volume 1304, pages 111–120. Springer-Verlag, 1997.

[56] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the* 25^{th} *International Symposium on Microarchitecture (MICRO)*, pages 45–54, Dezembro 1992.

[57] W. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. Prasanna, and H. Spaanenburg. Seeking solutions in configurable computing. *IEEE Computer*, 30(12):38–43, Dezembro 1997.

[58] N. Moreano, G. Araujo, Z. Huang, and S. Malik. Datapath merging and interconnection sharing for reconfigurable architectures. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS)*, pages 38–43, 2002.

[59] N. Moreano, G. Araujo, and C. Souza. CDFG merging for reconfigurable architectures. Technical Report 18, Institute of Computing – UNICAMP, 2003.

[60] N. Moreano, E. Borin, C. Souza, and G. Araujo. Efficient datapath merging for partially reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 24(7):969–980, Julho 2005.

[61] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[62] S. Niskanen and P. Östergård. Cliquer user's guide, version 1.0. Technical Report T48, Communications Laboratory, Helsinki University of Technology, Finland, 2003.

[63] P. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, Agosto 2002.

[64] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization – Algorithms and Complexity*. Dover Publications, 1998.

[65] R. Razdan and M. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 172–180, 1994.

[66] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: A SystemC-based architecture description language. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Brasil, October 2004.

[67] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. A quick safari through the reconfiguration jungle. In *Proc. Design Automation Conf.*, pages 172–177, 2001.

[68] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Taylor. PipeRench: A virtualized programmable datapath in 0.18 micron technology. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, pages 63–66, 2002.

[69] SEMATECH. International technology roadmap for semiconductors – 1999 edition, 1999. Disponível em http://public.itrs.net.

[70] SEMATECH. International technology roadmap for semiconductors – 2002 update, 2002. Disponível em http://public.itrs.net.

[71] N. Shirazi, W. Luk, and P. Cheung. Automating production of run-time reconfigurable designs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 147–156, 1998.

[72] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Filho. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, Maio 2000.

[73] C. Souza, A. Lima, N. Moreano, and G. Araujo. The datapath merging problem in reconfigurable systems: Lower bounds and heuristic evaluation. In *Proceedings of the Third International Workshop on Experimental and Efficient Algorithms (WEA), Lecture Notes in Computer Science*, volume 3059, pages 545–558, 2004.

[74] C. Souza, A. Lima, N. Moreano, and G. Araujo. The datapath merging problem in reconfigurable systems: Lower bounds and heuristic evaluation. *Submetido para ACM Journal of Experimental Algorithmics (JEA)*, Janeiro 2005.

[75] SPARC International. *The SPARC Architecture Manual – version 8*, 1992. Disponível em http://www.sparc.org.

[76] R. Stallman. *GNU Compiler Collection Internals*, 2002. Disponível em http://gcc.gnu.org/onlinedocs/.

[77] Fei Sun, Srivaths Ravi, Anand Raghunathan, and Niraj Jha. Custom-instruction synthesis for extensible-processor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 23(2):216–228, 2004.

[78] Synopsys, Inc. *Design Compiler User Guide*, 2003.

[79] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 28(1-2):7–27, Maio 2001.

[80] F. Vahid. Making the best of those extra transistors. *IEEE Design and Test of Computers*, page 96, 2003.

[81] F. Vahid. The softening of hardware. *IEEE Computer*, pages 27–34, Abril 2003.

[82] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the 7th International Symposium on High-level Synthesis*, pages 11–16, 1994.

[83] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, and W. Bohm. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 116–124, 2001.

[84] A. van der Werf et al. Area optimization of multi-functional processing units. In *Proceedings of the 1992 International Conference on Computer-Aided Design*, pages 292–299, 1992.

[85] W. Wolf. *Computers as Components – Principles of Embedded Computing System Design*. Morgan Kaufmann Pub., 2001.

[86] Xilinx. Virtex^{TM} 2.5v field programmable gate arrays, Abril 2001. Disponível em http://www.xilinx.com.

[87] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2000.

[88] Z. Ye, N. Shenoy, S. Hauck, and P. Banerjee. A C compiler for a processor with a reconfigurable functional unit. Technical report, Dept. of ECE – Northwestern University, 1999.