2D-VLIW: Uma Arquitetura de Processador Baseada na Geometria da Computação

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Ricardo Ribeiro dos Santos e aprovada pela Banca Examinadora.

Esto exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: RICARDO RIBEIRO DOS SANTOS
e aprovada pela Banca Examinadora
Campinas, 03 de OUTUBRO de 07
Rodolfo Azevedo
COORDENADOR

Campinas, 03 de Agosto de 2007.

Rodolfo Azevedo
Prof. Dr. Rodolfo Jardim de Azevedo
(Orientador)

Guido Araújo
Prof. Dr. Guido Costa Souza de Araújo
(Co-orientador)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Santos, Ricardo Ribeiro dos

Sa59d 2D-VLIW: uma arquitetura de processador baseada na geometria da computação / Ricardo Ribeiro dos Santos -- Campinas, [S.P. :s.n.], 2007.

Orientadores : Rodolfo Jardim de Azevedo; Guido Costa Souza de Araújo

Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computadores. 2.Compiladores (Computadores). 3. Circuitos integrados digitais. 4. Alocação de recursos. I. Azevedo, Rodolfo Jardim de. II. Araújo, Guido Costa Souza de. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Título em inglês: 2D-VLIW: a processor architecture based on the geometry of the computation

Palavras-chave em inglês (Keywords): 1. Computer architecture. 2. Compilers (Computers). 3. Digital integrated circuits. 4. Resource allocation.

Área de concentração: Arquitetura de Computadores

Titulação: Doutor em Ciência da Computação

Banca examinadora: Prof. Dr. Rodolfo Jardim de Azevedo (IC-UNICAMP)
Prof. Dr. Alberto Ferreira de Souza (INF-UFES)
Profa. Dra. Liria Matsumoto Sato (POLI-USP)
Prof. Dr. Paulo César Centoducatte (IC-UNICAMP)
Prof. Dr. Mario Côrtes (IC-UNICAMP)

Data da defesa: 10-07-2007

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 10 de julho de 2007, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Alberto Ferreira de Souza
INF - UFES.

Profª. Drª. Liria Matsumoto Sato
POLI - USP.

Prof. Dr. Mario Lúcio Côrtes
IC - UNICAMP.

Prof. Dr. Paulo César Centoducatte
IC - UNICAMP.

Prof. Dr. Rodolfo Jardim de Azevedo
IC - UNICAMP.

Instituto de Computação
Universidade Estadual de Campinas

2D-VLIW: Uma Arquitetura de Processador Baseada na Geometria da Computação

Ricardo Ribeiro dos Santos[1]

Junho de 2007

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo (Orientador)
- Prof. Dr. Alberto Ferreira de Souza
  Departamento de Informática - UFES
- Profa. Dra. Liria Matsumoto Sato
  Departamento de Engenharia de Computação e Sistemas Digitais - USP
- Prof. Dr. Mario Lúcio Côrtes
  Instituto de Computação - UNICAMP
- Prof. Dr. Paulo César Centoducatte
  Instituto de Computação - UNICAMP
- Prof. Dr. Ivan Luiz Marques Ricarte
  Faculdade de Engenharia Elétrica e Computação - UNICAMP (Suplente)
- Prof. Dr. Sandro Rigo
  Instituto de Computação - UNICAMP (Suplente)

[1]Suporte financeiro do CNPq (processo 142011/2006-1) 2006–2007, Capes 2005-2006 e Universidade Católica Dom Bosco 2004–2005.

Dedicatória

À pessoa que mais me incentivou na longa caminhada dos estudos e que, de maneira tão abrupta e por razões que ultrapassam o nosso entendimento, não está mais aqui para nos abraçar...
Esta Tese é dedicada à memória de uma brasileira comum, mãe, esposa, professora do ensino fundamental da rede pública, enfim, uma guerreira que acreditava piamente na transformação que a educação pode exercer na vida das pessoas...
minha Tia, Nildete Ribeiro Porto Dourado.

Agradecimentos

Jamais imaginei que iniciar os agradecimentos de uma tese de doutorado fosse tão difícil...Foram tantas as pessoas que gostaria de mencionar neste espaço que, certamente, teria que dedicar o mesmo tempo de escrita da tese...Sabendo que isso não é possível, quero deixar um agradecimento a todas as pessoas que direta ou indiretamente fizeram e fazem parte da minha vida e que, por falta de espaço e/ou falhas de memória desse que vos escreve, não constarão na listagem a seguir.

Meus agradecimentos iniciais são dedicados a DEUS. Por tudo que passei nesse período do doutorado, posso dizer que sem Ele, chegar ao final desta etapa não seria possível.

Devo toda a minha gratidão à Rúbia Mara de Oliveira Santos, minha esposa, com quem tenho compartilhado os momentos mais felizes da minha vida. Como se isso já não fosse bastante, a Rúbia foi uma grande motivadora durante todo o período do doutorado.

Quero também agradecer ao Professor Rodolfo Azevedo. A sua disponibilidade e atenção constantes para discutir assuntos e propor alguma solução me impressionou muito. Além disso, sua cordialidade e paciência me passaram muita segurança para investir nas idéias e desenvolver o trabalho.

Anos após anos meus pais: Nelson e Nadir, continuam sendo uma fonte de bons exemplos e admiração para mim. Devo agradecê-los por terem procurado me ensinar que com o trabalho, a justiça e a bondade, conseguimos vencer muitas adversidades. Embora não entendendo muito o que eu estudei durante essa estada em Campinas, sei que eles estão muito orgulhosos de mim. Agradeço também à minha irmã Rejane. Quero agradece-la por ter cuidado dos meus pais nesse período em que, eu sei, estive tão ausente. Não posso deixar de agradecer a minha nova família, meus sogros Israel e Iraídes e cunhados: Misael e

Gisele, Elizeu e Érica e o pequeno Guilherme. Meus sogros e cunhados me acolheram com muita alegria e fizeram com que eu pudesse esquecer, por alguns momentos, os grandes problemas que tinha para resolver no doutorado.

Agradeço também ao Professor Guido Araújo, especialista e conhecedor exímio de arquiteturas de computadores e compiladores. Quero agradecer ao Professor Guido por ter me aceitado no Programa de Doutorado do IC-UNICAMP e pela paciência e boa vontade em sacrificar a sua agenda para discutir as minhas dúvidas e ler meus artigos.

Aqui no IC quero dedicar um agradecimento especial a dois colegas cuja troca de experiências foi muito frutífera para mim. A amizade que fiz com o Evandro Bracht e com o Carlos Froldi é dessas coisas que levarei para o resto da vida. No caso do Carlos, a paixão comum pelo "Todo Poderoso Timão" também serviu para muitas situações de risadas, alegrias e sofrimentos.

Aqui em Campinas encontrei pessoas maravilhosas que fizeram a minha adaptação à cidade muito mais fácil e prazeirosa. Quero registrar meus sinceros agradecimentos ao grande amigo Júnior (e sua esposa Luciane) cuja amizade iniciou em São Carlos, nos tempos do mestrado e, com a ajuda de DEUS, nunca irá acabar; Ao André Drummond (e sua esposa Elaine) que com seu bom humor possibilitava momentos muito descontraídos na sala 86; A amiga Paolla e sua família que graças ao seu carinho e ajuda em vários momentos, fizeram com que a estadia aqui em Campinas ficasse marcada para sempre.

Agradeço também aos colegas que encontrei no LSC, docentes e funcionários do IC que sempre, com muita paciência, tem me auxiliado nas situações mais diversas.

Quero agradecer ao apoio financeiro da Capes e do CNPq que, em diferentes momentos, foram imprescindíveis para manter meu foco apenas na pesquisa. Quero também agradecer à UCDB pelo suporte financeiro na fase inicial do doutorado.

Por fim, registro meus sinceros agradecimentos à UNICAMP que, com sua filosofia de ensino e pesquisa, fornece muitos exemplos para serem seguidos no restante desse país. Para mim, que vem de uma região onde estudar em escolas públicas e de qualidade é contrariar a estatística, ter estudado aqui foi o cumprimento de um sonho maravilhoso.

Resumo

Anúncios recentes sobre os limites do desempenho dos processadores devido ao alcance da barreira térmica têm motivado a pesquisa sobre novas organizações arquiteturais e modelos de execução que visam continuar o aumento de desempenho dos processadores. Este trabalho propõe uma nova arquitetura de processador denominada 2D-VLIW. A arquitetura possui uma organização arquitetural baseada em uma matriz bidimensional de unidades funcionais e de registradores distribuídos ao longo dessa matriz. O modelo de execução 2D-VLIW possibilita que instruções longas, formadas por operações simples, sejam buscadas na memória e executadas sobre a matriz de unidades funcionais. Além disso, são propostos algoritmos para geração de código para extrair o paralelismo e preparar o código para ser executado sobre a arquitetura. Algumas contribuições deste trabalho são a concepção de uma nova arquitetura de processador que explora paralelismo em nível de instruções através de um novo arranjo dos elementos arquiteturais, a adoção de um modelo de execução que captura a geometria dos DAGs e associa os vértices e arestas desses DAGs aos recursos do hardware, um conjunto de algoritmos para escalonamento de instruções, a alocação de registradores e a codificação de instruções na arquitetura 2D-VLIW. Os resultados experimentais comparam o desempenho do modelo de execução dessa arquitetura com o modelo EPIC adotado pelo processador HPL-PD. O *speedup* obtido por 2D-VLIW foi de 5% até 63%. A estratégia de escalonamento adotada por 2D-VLIW foi também avaliada e os ganhos obtidos através do OPC e OPI foram até 4 vezes melhores que aqueles obtidos por um algoritmo de escalonamento baseado em *list scheduling*.

Abstract

Recent announcements on processor performance limits due to the thermal barrier have motivated research into innovative architectural organizations and execution models to sustain the increase of performance. This work proposes a new architecture named 2D-VLIW. The architecture provides a new architectural organization of the processing elements by using a two-dimensional functional units matrix and registers spread out along this matrix. The 2D-VLIW execution model fetches long instructions comprised of simple operations in the memory and dispatches these operations to the matrix. Moreover, the work presents new algorithms for code generation which are the responsible for extracting the parallelism of the applications and preparing the code for the 2D-VLIW architecture. Some contributions of this work are a new high performance architecture that exploits instruction level parallelism by a new arrangement of the architectural elements, the adoption of an execution model that captures the geometry of the DAGs and matches them to the hardware resources, a set of algorithms for code generation that make them possible to schedule instructions, allocate registers and encode long instructions of the 2D-VLIW architecture. Experimentos were used for comparing the performance of the 2D-VLIW execution model to the EPIC execution model of the HPL-PD architecture. The speedup obtained by 2D-VLIW ranges from $5\% - 63\%$ for all the evaluated programs. The scheduling strategy based on subgraph isomorphism was also evaluated and the OPC and OPI gains were up to $4\times$ better than that of the list scheduling algorithm.

Notação

- Conjuntos são definidos em letras maiúsculas e em formato matemático. Assim, V, I, P são conjuntos. A operação $|V|$ indica a cardinalidade do conjunto, isto é, a quantidade de elementos do conjunto.
- Elementos de um conjunto são apresentados em letras minúsculas e formato matemático. Elementos específicos de um conjunto possuem um índice subscrito junto à letra que define o elemento, como é o caso de $x_i \in S$.
- Grafos recebem a notação $G = (V, E)$. V é o conjunto de vértices e E o conjunto de arestas do grafo G. Para diferenciar dois ou mais grafos, adota-se um índice subscrito junto a cada uma das letras. Assim, $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ são grafos distintos.
- Funções e variáveis de um algoritmo são apresentados em modo matemático. Dessa forma, $evaluate()$ representa uma função e tag representa uma variável.
- Ao longo do texto, procura-se traduzir termos e definições em inglês para a língua portuguesa. No entanto, palavras e termos que não possuem uma tradução bem conhecida em português são mantidas sem tradução (em inglês), no formato itálico. Esse é o caso de *backtracking* e *list scheduling*, por exemplo. Termos que já foram adicionados aos dicionários da língua portuguesa, como chip e cache, são escritos sem formatação.
- Mnemônicos de operações como addi, st e sub são formatados com fonte sans serif.
- A relação de dependência entre duas operações x e y, indicando que y depende do resultado de x, é dada por $x \rightarrow y$.

Lista de Acrônimos

ADRES: *Architecture for Dynamicaly Reconfigurable Embedded Systems*. Arquitetura composta por um processador VLIW e uma matriz de unidades funcionais reconfiguráveis.

CMOS: *Complementary Metal Oxide Semiconductor*. Técnica amplamente utilizada na fabricação de transistores.

DAG: *Directed Acyclic Graph*. Grafo que evidencia as dependências entre operações. Uma aresta e_{ij} de um DAG, indica que o vértice i produz um valor que será usado pelo vértice j. Em outras palavras, a aresta representa a dependência da operação j para a operação i. Uma aresta e_{ij} também carrega consigo um peso $k > 0$, indicando a latência da operação i. Os vértices de um DAG representam as operações de um programa.

DFG: *Data Flow Graph*. Grafo que indica o fluxo de dependências de dados entre blocos de um programa.

DRESC: *Dynamically Reconfigurable Embedded System Compiler*. Compilador utilizado na geração de código para a arquitetura ADRES.

EDGE: *Explicit Data Graph Execution*. Um modelo de execução e conjunto de instruções adotado pelo processador TRIPS.

ELI-512: *Enormously Longword Instructions-512 bits*. Um processador VLIW proposto por Joseph A. Fisher na Universidade de Yale no início dos anos 80.

EPIC: *Explicitly Parallel Instruction Computing*. Um modelo de arquitetura que explora paralelismo em nível de instrução através de diversas otimizações de compilação.

GPA: *Grid Processors Architecture*. Definição inicialmente atribuída a um processador que segue o modelo EDGE.

HMDES: *Hardware Machine Description language*. Linguagem para descrição de arquiteturas de processadores acoplada ao compilador Trimaran.

HPL-PD: *HP Labs PlayDoh*. Processador parametrizável que implementa o modelo arquitetural EPIC para execução de operações de um programa.

ILP: *Instruction Level Parallelism*. Medida utilizada para indicar o paralelismo alcançado por um processador.

LSC: Laboratório de Sistemas de Computação.

MIMD: *Multiple Instruction Multiple Data*. Um modelo de execução para máquinas paralelas em que várias instruções diferentes podem executar simultaneamente, sendo que cada uma utiliza dados (operandos) diferentes.

MIPS: *Microprocessor without Interlocked Piped Stages*. Designa uma família de microprocessadores que implementam o padrão RISC.

OPC: *Operações por Ciclo*. Medida que indica a quantidade de operações por ciclo alcançada por uma técnica, sistema ou algoritmo.

OPI: *Operações por Instrução*. Medida que informa a quantidade de operações por instrução alcançada por um algoritmo de escalonamento.

PPE: *Power Processor Element*. Processador que compõe a arquitetura Cell e é responsável pela atribuição de tarefas aos SPEs. É também o responsável pela execução do sistema operacional.

PRISC: *Programmable Instruction Set Computers*. Arquitetura reconfigurável com unidades funcionais reconfiguráveis adicionadas à via de dados do processador.

RAW: *The Raw Architecture Workstation*. É uma arquitetura com múltiplos processadores (arquitetura com múltiplos núcleos) que possibilita a execução a execução de diferentes granularidades de dados.

RAWCC: *The Raw Architecture Workstation C Compiler*. É o compilador baseado em linguagem C que gera código para processadores da arquitetura RAW.

RG: *Registrador Global*. Registrador que armazena os valores de parâmetros, operandos de raízes de um DAG e resultados de folhas de um DAG na arquitetura 2D-VLIW. Essa arquitetura possui um banco de RGs composto por 32 registradores.

RISC: *Reduced Instruction Set Computer*. Um modelo de arquitetura de processadores baseado em um conjunto de instruções reduzido.

RT: *Registrador Temporário*. É o registrador que armazena resultados temporários das computações na arquitetura 2D-VLIW. Existem vários bancos de RTs no interior da matriz de UFs 2D-VLIW. Cada UF 2D-VLIW pode ler valores de dois bancos de RTs e escrever resultados em registradores de um banco de RTs.

RUF: *Registrador da Unidade Funcional* . Registrador presente em cada unidade funcional da arquitetura 2D-VLIW.

SIMD: *Single Instruction Multiple Data*. Um modelo de execução para arquiteturas paralelas em que uma instrução opera sobre dados (operandos) diferentes.

SPE: *Synergistic Processor Element*. A arquitetura Cell possui 8 SPEs. Esses processadores possuem unidades vetoriais e são os responsáveis pela realização da maior parte das tarefas em um processador Cell.

SPEC: *Standard Performance Evaluation Corporation*. Uma empresa que comercializa uma família de programas benchmarks muito usada na avaliação do desempenho de processadores. Além disso, SPEC também define um conjunto de métricas que podem ser usadas na avaliação de desempenho de arquiteturas de processadores.

SSA: *Single Static Assignment*. Otimização realizada sobre o código do programa que transforma fluxos de controle em fluxos de dados.

SUIF: *Stanford University Intermediate Format*. Uma infraestrutura de compilação que possibilita o suporte e colaboração envolvendo pesquisas sobre otimizações de código e paralelismo em nível de instrução.

TRIPS: *Tera-op, Reliable, Intelligently adaptive Processing System*. Um processador composto por uma matriz de unidades funcionais, memórias caches e registradores organizados ao longo dessa matriz. Esse processador implementa o modelo de execução EDGE.

UF: *Unidade Funcional* . É a unidade que executa o processamento das operações em várias arquiteturas como, por exemplo, 2D-VLIW.

UFP: *Unidades Funcionais Programáveis*. São unidades funcionais da arquitetura PRISC que permitem a reconfiguração de suas funções de acordo com as características da aplicação.

ULA: *Unidade de Lógica e Aritmética.* Elemento de hardware responsável pela execução de operações aritméticas e lógicas.

VF: *Vento e Fogia.* Iniciais dos nomes dos autores da biblioteca VF que agrupa um conjunto de algoritmos para isomorfismo de grafos e subgrafos.

VLIW: *Very Long Instruction Word.* Um estilo de arquitetura de processadores que buscam instruções longas na memória e executam as operações que compõem essas instruções de maneira paralela.

VLSI: *Very Large Scale Integration.* Indica uma escala de integração (densidade) adotada na fabricação de transistores.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Anúncios recentes sobre os limites do desempenho dos processadores devido ao alcance da barreira de temperatura têm levado a indústria e a comunidade científica a buscar soluções inovadoras na tentativa de manter o constante aumento de desempenho dos processadores. Em 2004, uma das companhias líderes do mercado de microprocessadores, a Intel® Inc., revelou uma mudança em sua estratégia de pesquisa e evolução de microprocessadores devido ao alcance da barreira termal [54]. Essa mudança está voltada para a obtenção de mais desempenho do processador através de múltiplos elementos de processamento em um mesmo chip.

Por muitos anos, o aumento constante no desempenho do processador foi devido, principalmente, ao avanço na tecnologia VLSI. Esse avanço tem possibilitado encapsular milhões e, atualmente, bilhões de transistores dentro de um único chip e, como conseqüência, construir processadores com mais poder de processamento e melhor desempenho. É importante esclarecer que o aumento de desempenho dos processadores também advém de técnicas embutidas no compilador [2, 62] e da adição de mais elementos de processamento junto ao hardware [19, 31, 55].

A discussão em torno dos limites tecnológicos para a construção de dispositivos de processamento cada vez menores e com maior capacidade de processamento não é algo recente. Igualmente, a busca por alternativas alheias a esses limites já dura algumas décadas. A partir de meados dos anos 80, diversos trabalhos de pesquisa têm abordado técnicas de compilação que consideram regiões de código maiores, transformação de fluxos de controle em fluxos de dados, a colocação de dados em memórias rápidas e mais

próximas do processadores buscando melhorar o paralelismo entre as operações de um programa e, também, aumentar o desempenho [44, 50, 76, 85]. Com esse mesmo intuito, a comunidade da área de arquiteturas de computadores tem presenciado o surgimento de diversas propostas de arquiteturas de processadores que adotam múltiplas unidades de processamento, memórias distribuídas ao longo do chip, unidades para manter a coerência da execução do programa, entre outras inovações [12, 13, 56, 69, 84].

O interessante nessa linha de trabalhos de pesquisa é que, além de utilizar vários elementos de hardware, há uma crescente preocupação em como organizar esses elementos de modo a fazer o melhor uso dos recursos existentes. Em outras palavras, como definir modelos ou paradigmas de execução que maximizam o uso de todos esses elementos enquanto procuram minimizar o tempo de processamento? Nesse sentido, uma das questões-chave é o uso balanceado dos diversos recursos. De forma geral, os projetistas procuram balancear as demandas pelos elementos do processador, pela memória, pelos dispositivos de entrada e saída e pelas estruturas de interconexão. Um agravante para essa questão é que o desempenho do processador cresce por um fator muito maior que o desempenho da memória. A conseqüência disso é que o projeto de uma arquitetura de processador deve considerar essas diferenças e lançar mão de técnicas para minimizá-las o máximo possível. Segundo [25], um desafio fundamental para projetistas de processadores está em como organizar os módulos de memória cache de modo a reduzir a latência do acesso por parte das operações de memória. A utilização da tecnologia VLSI encoraja o projeto de processadores que são mais isolados do subsistema de memória ao passo em que também são mais flexíveis para adaptar às nuâncias do comportamento desse subsistema. Kozyrakis e Patterson [49] acrescentam que outro grande desafio para os projetistas é utilizar eficientemente o grande número de transistores de modo a encontrar os requisitos de aplicações futuras. Esses e outros trabalhos [1] informam ainda sobre a necessidade de soluções que consideram o modelo que rege a execução desses elementos, a granularidade dos dados que serão executados, a redução da área de memória ocupada pelo programa e a redução da potência consumida pelo processador.

Diante desse contexto, o problema central tratado nesta Tese é a ausência de um modelo de execução e de uma organização arquitetural que consideram, de maneira

global, os padrões de processamento e organização de um programa. Por padrões de processamento, deve-se entender o grau de paralelismo que pode ser obtido por um programa, os tipos de operações que são executadas e os requisitos de hardware exigidos por essas operações. Os padrões de organização referem-se à geometria dos grafos de dependências das operações do programa, entre outras características desses grafos. A Seção 1.1 apresenta a proposta e algumas contribuições desta Tese. Na seqüência, a Seção 1.2 detalha o conteúdo dos próximos capítulos.

1.1 Proposta e Contribuições da Tese

Em face dos limites tecnológicos que a indústria de processadores está sendo obrigada a considerar e, diferente de outras propostas que buscam a melhoria de desempenho por meio do acréscimo de mais elementos de processamento, esta Tese propõe uma discussão diferenciada visando manter a evolução de desempenho que sempre caracterizou os processadores. Na realidade, a Tese apresenta uma nova arquitetura de processador denominada 2D-VLIW[1] e, atrelado a essa nova arquitetura, um modelo de execução cuja característica marcante é o mapeamento de regiões de código do programa sobre as unidades funcionais de uma matriz de processamento. O trabalho aqui apresentado defende o pressuposto de que o código existente em um programa obedece a um padrão de execução e as dependências entre as operações formam uma geometria bastante peculiar que pode ir ao encontro da organização dos recursos existentes na arquitetura. Adicionalmente, acredita-se que ao manipular regiões de código, ao invés de operações simples, pode-se tratar o uso dos recursos da arquitetura de maneira mais eficiente uma vez que está sendo considerada uma abordagem global para utilização desses recursos.

A arquitetura proposta, 2D-VLIW, possui um arranjo dos elementos de processamento diferente do que se tem observado na literatura da área. De maneira geral, a arquitetura adota otimizações de código que minimizam as dependências de fluxo de controle e maximizam a construção de agrupamentos de operações que se adequam à organização arquitetural dos elementos de hardware. Além disso, há uma preocupação

[1] Assim chamada por buscar na memória instruções compostas por vários operações e executá-las em uma matriz bidimensional de unidades funcionais.

sobre a área ocupada de um processador baseado em 2D-VLIW assim como de programas que executam sobre esse processador. A solução encontrada para a arquitetura é distribuir registradores ao longo de uma matriz de unidades funcionais de maneira a reduzir o número de acessos ao banco de registradores globais, além de restringir a interligação entre essas unidades. No caso do tamanho do programa, propõe-se uma técnica de codificação de instruções que extrai padrões das instruções e mantém esses padrões em uma memória cache especial, denominada cache de padrões.

Quando comparada com os processadores disponíveis atualmente que adotam múltiplos núcleos de processamento, a arquitetura apresentada nesta Tese pode ser vista como uma unidade básica desses processadores. Dessa forma, uma visão de alto nível revela um processador com múltiplos núcleos ou elementos de processamento enquanto que uma visão mais detalhada revela que cada elemento de processamento é composto por uma matriz de unidades funcionais. Essa é uma abordagem direta para exploração do paralelismo com diferentes granularidades. No nível da matriz de unidades funcionais pode-se explorar o paralelismo em nível de instruções. No nível dos elementos de processamento existe a possibilidade de explorar um paralelismo de granularidade mais grossa, por exemplo, em nível de processos.

Em suma, esta Tese procura apresentar soluções para algumas das questões apresentadas e discutidas anteriormente. Essas soluções estão reunidas em uma arquitetura de processador denominada 2D-VLIW.

Dentre as contribuições desta Tese, pode-se citar as seguintes:

- A concepção de um novo modelo de execução que considera a geometria do programa e, através de um grafo de dependências de operações, combina esse grafo sobre a organização dos elementos da arquitetura;

- A proposição de uma organização do hardware que explora o paralelismo existente nos programas e, ao mesmo tempo, propicia a escalabilidade da arquitetura;

- O projeto e implementação de algoritmos para geração de código que manipulam as operações de um programa de maneira global e preparam essas operações para serem executadas sobre o hardware 2D-VLIW;

1.2 Organização da Tese

O texto desta Tese está organizado em cinco capítulos que fornecem os conceitos e pressupostos metodológicos para o entendimento completo do trabalho. Os parágrafos a seguir detalham cada um desses capítulos.

O Capítulo 2 apresenta os conceitos que serão considerados ao longo da Tese. Ademais, relata as pesquisas relacionadas e procura estabelecer uma comparação entre essas pesquisas com vista a tornar evidente a existência de lacunas na área e a possibilidade de desenvolvimento de novos trabalhos.

O Capítulo 3 descreve os elementos que compõem a arquitetura 2D-VLIW. O capítulo apresenta a arquitetura 2D-VLIW a partir de uma visão de alto nível para um detalhamento dos elementos da arquitetura. Além disso, há uma preocupação em relatar como operações de um programa são executadas sobre a arquitetura através de seu modelo de execução.

O Capítulo 4 apresenta as questões envolvendo a geração de código na arquitetura 2D-VLIW. Há ênfase na descrição das atividades de escalonamento de instruções, alocação de registradores e codificação de instruções, além da justificativa sobre a adoção do compilador Trimaran como infraestrutura básica para geração de código na arquitetura 2D-VLIW. A escolha dessas atividades se dá em virtude do impacto que exercem sobre o desempenho dos programas. Nesse sentido, dadas as particularidades inerentes à arquitetura 2D-VLIW, essas atividades fazem uso de técnicas inovadoras e proporcionam resultados bastante interessantes que podem servir de motivação para adoção em processadores com características similares a 2D-VLIW.

Experimentos envolvendo características da arquitetura 2D-VLIW são descritos no Capítulo 5. Esse capítulo detalha os experimentos realizados e, principalmente, justifica e discute os resultados obtidos. Seguindo o que foi apresentado nos capítulos anteriores, os experimentos são direcionados à avaliação do desempenho da arquitetura através de seu modelo de execução e na comparação desse desempenho com uma arquitetura de processador com o mesmo número de unidades funcionais. Outro grupo de experimentos foca a eficiência das técnicas implementadas para geração de código. Em especial, procura-se mostrar os ganhos obtidos com um algoritmo de escalonamento proposto exclusivamente

para a arquitetura 2D-VLIW em contraposição a um algoritmo utilizado por arquiteturas de processadores em geral. Há também experimentos que avaliam, estaticamente, a técnica de codificação de instruções mostrando os ganhos em termos da redução no tamanho dos programas.

Ao longo de seu desenvolvimento, este trabalho tem se caracterizado como uma proposta que abre novas oportunidades de pesquisa para diversos problemas relacionados ao projeto de processadores. O Capítulo 6 conclui sobre o que foi realizado durante o desenvolvimento deste trabalho e relata as contribuições que foram alcançadas. As publicações conseguidas com esta Tese são apresentadas e, ao final, sugere-se a extensão desta proposta através de vários projetos que podem ser realizados como trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

O aumento da disponibilidade de transistores a um custo cada vez mais reduzido tem motivado o acréscimo de elementos de hardware nos processadores atuais. Com o advento da computação reconfigurável e, mais recentemente, das arquiteturas com múltiplos núcleos, essa adição de recursos através de circuitos somadores, subtratores, barramentos e memórias tem sido ainda mais evidente. Diante desse contexto, este capítulo revisa a literatura da área apresentando um conjunto de trabalhos que definem novos conceitos, algoritmos e técnicas que propiciam o uso dessa multiplicidade de recursos na tentativa de maximizar o desempenho na execução de programas. As propostas aqui apresentadas foram escolhidas tendo como base a proximidade de características com a arquitetura 2D-VLIW.

2.1 Definição de Parâmetros para Análise

Um dos problemas comumente enfrentados na comparação entre diferentes arquiteturas de processadores reside na grande quantidade de variáveis (características) existentes e que podem ser levadas em consideração. Alguns exemplos dessas variáveis são: conjunto de instruções, modelos de execução, granularidade, características de acesso à memória, número de unidades funcionais (UFs), número e tamanho dos registradores, rede de interconexão, entre outros.

Diante de tantas variáveis, é imprescindível determinar, claramente, quais devem ser usadas visando relacionar diferentes trabalhos. Considerando que há similaridade

entre características de outros trabalhos e a proposta presente nesta Tese, a descrição dos trabalhos da área apresentados a seguir foi embasada em cinco características principais:

- **Organização Arquitetural.** Esta característica tem sido um fator preponderante para o avanço e mudanças em muitos projetos de arquitetura de processadores apresentados nos últimos anos. O intuito aqui é apresentar informações quantitativas considerando: número de UFs, número e tamanho dos registradores, quantidade e tamanho de memórias caches, *buffers*, entre outros.

- **Topologia de Interconexão.** Esta característica diz respeito à topologia e desempenho da rede de interconexão adotada para interligar os elementos de processamento.

- **Modelo de Execução.** O foco aqui é mostrar como uma operação é executada no processador e quais recursos são utilizados em tempo de execução.

- **Geração de Código.** Todo processador deve oferecer uma interface, através da qual o código de um programa pode utilizar os recursos de hardware em tempo de execução. No entanto, para que um programa possa utilizar tais recursos, ferramentas de compilação são necessárias para traduzir programas em seqüências de código que sejam inteligíveis pelo hardware. Essas ferramentas podem ser desde infraestruturas de compilação que geram código binário a partir de um programa em linguagem de alto nível, até algoritmos para alocação de recursos, otimizações de código, maximização de operações paralelas.

- **Desempenho e Implementação.** O intuito aqui é apresentar informações sobre o desempenho da arquitetura, experimentos e implementações realizadas em hardware.

- **Granularidade do paralelismo.** A granularidade do paralelismo indica se uma arquitetura procura melhorar o desempenho através do paralelismo de instruções (granularidade fina), linhas de execução e/ou processos (granularidade grossa).

Além da problemática envolvendo a definição de critérios para avaliação de arquiteturas de processadores diferentes, uma outra dificuldade reside na discrepância de

termos e, até mesmo, alguns conceitos usados pelas arquiteturas. Nesse sentido, deve-se considerar as definições a seguir para os trabalhos discutidos na Seção 2.2.

Definição 1 *Uma **operação** representa uma função que deve ser executada pelo hardware. Uma operação engloba também os operandos (parâmetros) da função assim como o local onde os resultados devem ser armazenados. Exemplos de operações são* add, sub, mul *e* and.

Definição 2 *Uma **instrução** é armazenada na memória como um conjunto composto por uma ou mais operações.*

Definição 3 *A **unidade funcional** é a unidade responsável pela execução de uma operação. Possui multiplexadores para selecionar as entradas e saídas, unidade de lógica e aritmética e unidade de controle. Pode ser especializada para uma classe de operações.*

Definição 4 *Um **elemento de processamento** é formado por um processador com capacidade para executar todo o conjunto de instruções de uma arquitetura. De maneira geral, um elemento de processamento possui estágios de pipeline, interface de acesso à memória, unidade de lógica e aritmética e registradores.*

2.2 Propostas de Arquiteturas

Esta seção apresenta as características de algumas arquiteturas de processadores que utilizam múltiplas unidades funcionais e/ou elementos de processamento na tentativa de explorar o paralelismo de instruções existente em programas. Existe uma grande quantidade de propostas de arquiteturas com múltiplas unidades funcionais disponíveis na literatura. Nesse sentido, as propostas aqui apresentadas foram escolhidas tendo como base as similaridades de características com o objeto de estudo desta Tese e a disponibilidade de informações.

2.2.1 Arquitetura RAW

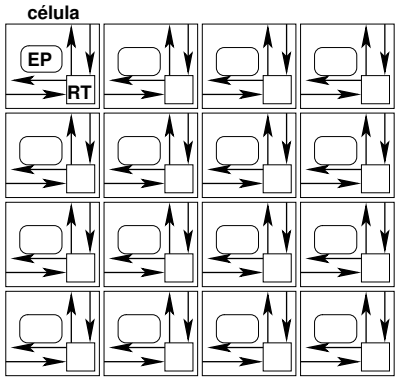
A arquitetura RAW [84] é um projeto de pesquisa iniciado no *Massachussets Institute of Technology - MIT* em meados da década de 90, cuja característica principal

reside na organização de células de computação visando à execução de operações. Cada célula possui memória para instruções e dados, um elemento de processamento (EP), registradores, dispositivos de lógica configurável e um roteador que suporta roteamento dinâmico e estático. Através da lógica configurável, operações específicas e de vários tipos de granularidade podem ser executadas. O modelo de execução RAW segue o modelo básico de execução de um processador RISC com *pipelines* (por exemplo, o processador MIPS). A diferença básica é que RAW possui uma matriz de elementos de processamento e, com isso, existem mais possibilidades de exploração do paralelismo.

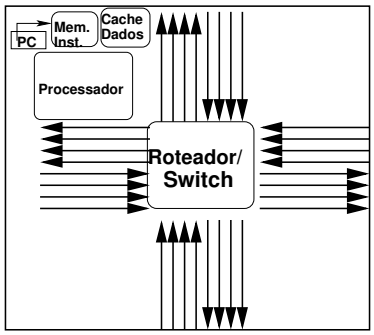
A Figura 2.1(a) apresenta a organização da arquitetura disposta em uma matriz M com 4×4 células de computação. Cada célula m_i atua como um processador independente e pode comunicar com qualquer outra célula m_j, $i \neq j$, da matriz através do roteamento dinâmico de pacotes de dados. Na Figura 2.1(b) pode-se observar os componentes de uma célula de processamento assim como o sistema de interconexão. A Figura 2.1(c) detalha os estágios do *pipeline* no processador de uma célula.

A geração de código para execução na arquitetura RAW fica a cargo do compilador RAWCC [52]. Esse compilador é baseado na infraestrutura de compilação SUIF e possui três fases principais: otimizações independentes de máquina, escalonamento e alocação de registradores, geração de código e roteamento. Dentre essas três fases, a etapa de alocação de recursos (escalonamento) merece atenção especial em virtude de seu impacto sobre a utilização dos recursos disponíveis e a eficiência no desempenho do programa. De maneira geral, o escalonamento de instruções na arquitetura RAW consiste em atribuir operações, dentro de blocos básicos, para células de processamento considerando a conexão dessa célula para outras células que a operação mantém dependência. O compilador RAWCC utiliza um algoritmo de *list scheduling* [34, 40] estendido com heurísticas para calcular a latência de comunicação para escalonar as instruções.

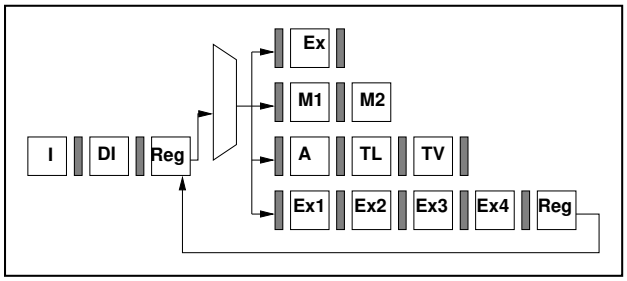
Os autores [81] relatam a implementação de um protótipo RAW com 16 células de processamento utilizando, cada uma, tecnologia de $180nm$ e com uma freqüência de 450MHz. O desempenho dessa implementação foi comparado com um processador Pentium® III com 600MHz e também fabricado com tecnologia de $180nm$. As capacidades de memória e registradores de ambas arquiteturas sob comparação são equivalentes. Nos



(a) Matriz de células. EP=Elemento de Processamento; RT=Roteador.



(b) Detalhamento de uma célula.



(c) Estágios de *pipeline* de um EP. Os três estágios iniciais (busca da instrução, decodificação e leitura dos registradores) são comuns para todas as classes de operações.

Figura 2.1: Arquitetura RAW.

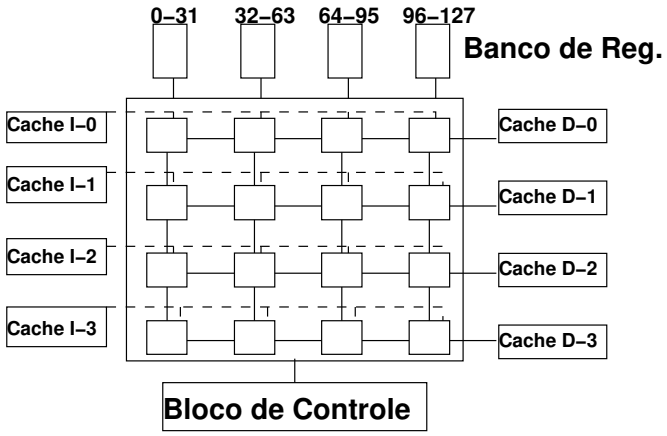
experimentos realizados com aplicações científicas voltadas para cálculos com matrizes densas, RAW alcançou um *speedup* variando de $1.3 - 6.4$ sobre o processador Pentium III.

2.2.2 Arquitetura TRIPS

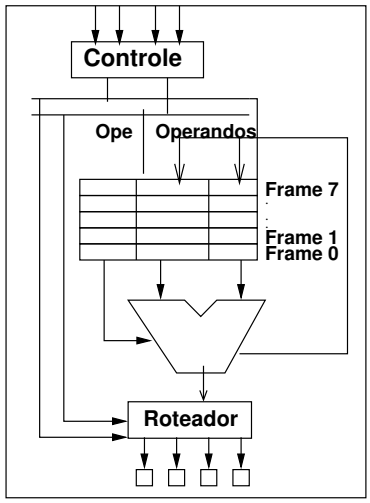
A Arquitetura TRIPS (denominada previamente GPA) é uma instância do conjunto de instruções EDGE [13, 64]. Essa arquitetura foi projetada com o pressuposto de alcançar maiores níveis de ILP que arquiteturas convencionais através de um novo modelo de execução e arranjo dos elementos de hardware. A base da arquitetura TRIPS consiste em uma matriz M, com 4×4 elementos de processamento, em que cada elemento $m_i,\ 1 \leq i \leq 16$ possui uma memória de instrução e uma unidade de execução. Esses elementos são conectados através de uma rede dedicada a fim de transmitir operandos e dados. O controle dos elementos de processamento é feito por uma linha de execução[1] que controla o mapeamento de blocos de operações para esses elementos. De maneira similiar às arquiteturas VLIW, um compilador é utilizado para detectar paralelismo e escalonar as instruções para a matriz M, de forma que o grafo de fluxo de dados pode ser mapeado inteiramente na arquitetura. A Figura 2.2 apresenta uma visão geral da arquitetura TRIPS.

Na Figura 2.2(a), a memória cache de instruções, $CacheI$, é responsável por armazenar as operações que serão enviadas para os elementos da matriz M enquanto que a cache de dados, $CacheD$, mantém os operandos dessas operações. Há uma unidade funcional denominada bloco de controle que determina qual grupo de operações deve ser mapeado para a matriz e quando tal grupo termina a execução. Na Figura 2.2(b), cada elemento da matriz possui uma ULA, portas de entrada para os operandos, *buffers* de operandos e operações e um roteador que envia os valores de saída para portas específicas. Cada ULA possui 8 *frames* que armazenam operações de um bloco de operações mapeado sobre a matriz. A arquitetura possui suporte para especulação de blocos e permite que até 8 blocos (1024 operações= 8 blocos $\times$ 8 *frames* $\times$ 16 EPs) sejam trazidos para a matriz.

[1] Do inglês: *thread*.



(a) Visão geral da matriz TRIPS.



(b) Detalhamento de um elemento de processamento.

Figura 2.2: Arquitetura TRIPS.

No modelo de execução TRIPS, o compilador é o responsável por organizar as operações em blocos. Com isso, um bloco de operações pode ser um bloco básico[2] [3], um hiperbloco[3] [53] ou um traço de execução [3, 62]. Os dados presentes em um bloco são de três tipos: (1) dados de saída, que são valores criados pelo bloco e usados por grupos subseqüentes; (2) temporários, que são valores produzidos e consumidos dentro do bloco e (3) dados de entrada, que são valores produzidos pelos blocos anteriores na seqüência de execução.

A execução de um bloco de operações acontece após a busca do bloco e o mapeamento dessas operações sobre a matriz. Cada operação é armazenada em uma entrada do EP. Após isso, uma operação **move** lê as entradas do bloco e repassa os operandos para os EPs que estão armazenando as operações. A medida que uma operação termina sua operação, os resultados são enviados para outros EPs que mantêm operações consumidoras desses resultados ou para registradores, caso os resultados sejam dados de saída do grupo. O destino do resultado é codificado explicitamente na operação. Cada destino é referido pelo nome do EP de maneira que o resultado possa ser enviado diretamente para a operação consumidora. Uma operação pode armazenar informações de localização de até 3 outras operações consumidoras. Caso o número de consumidores seja maior que 3, uma operação especial, denominada **split**, é inserida no escalonamento a fim de permitir o envio dos resultados para todos os destinos.

A arquitetura TRIPS utiliza o compilador Scale [13] para a geração de código. Esse compilador executa quatro tipos de otimizações:

- **Formação de hiperblocos:** hiperblocos são utilizados pelo fato de agruparem mais operações do que blocos básicos comuns e transformarem algumas dependências de fluxo de controle em fluxo de dados através da inserção de predicados.

- **Execução predicada:** predicados possibilitam que ambos os lados de um bloco de decisão (*if-else*) sejam avaliados simultaneamente, embora, apenas o lado que foi

[2]Blocos de operações cuja principal característica é que a execução inicia pela primeira operação do bloco e o saída ocorre apenas pela última operação do bloco.

[3]Um hiperbloco é um conjunto de blocos básicos com predicados. Nesse conjunto, o controle pode iniciar apenas na primeira operação mas, ao contrário do bloco básico, pode sair por mais de um local. Hiperblocos são formados a partir de técnicas de *if-conversion*.

tomado pelo bloco de decisão seja efetivamente executado. Essa otimização aumenta consideravelmente a quantidade de operações por ciclo.

- **Adequação de hiperblocos:** essa otimização averigua o uso de recursos por parte das operações de um hiperbloco e, se necessário, divide o hiperbloco até que o número de operações restantes não ultrapasse 128 (8 *frames* $\times$ 16 EPs) sendo, no máximo, 32 operações de carregamento ou armazenamento na memória, leitura máxima de 32 registradores e escrita em 32 registradores.

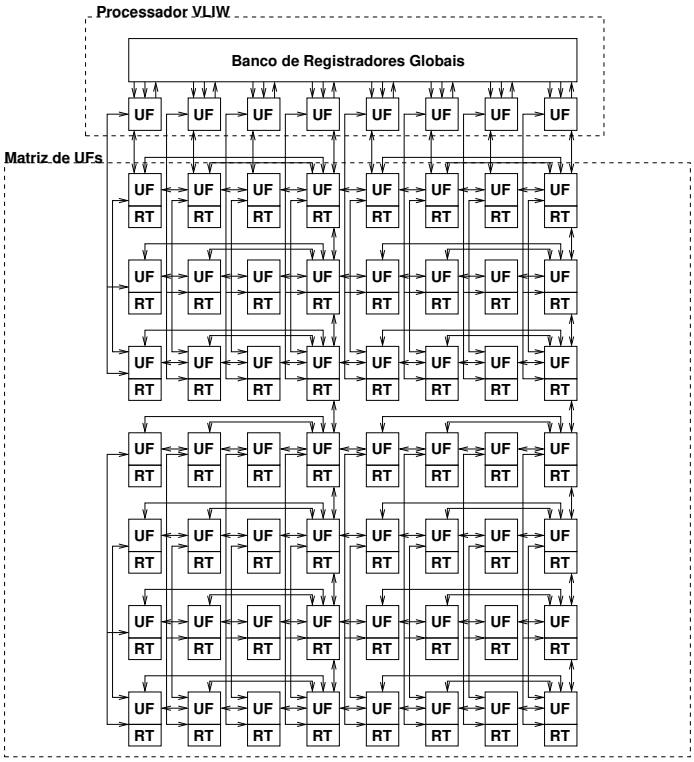
- **Alocação de recursos físicos:** essa otimização mapeia operações para EPs e insere operações de cópia quando uma operação produtora deve rotear seu resultado para operação(ões) consumidora(s). Através de heurísticas implementadas junto ao escalonador [21, 63], operações independentes são alocadas em diferentes EPs para aumentar a concorrência. Operações dependentes são alocadas em elementos próximos a fim de minimizar o roteamento e atrasos de comunicação.

Os autores relatam experimentos [69] onde foram consideradas variações de 2×2, 4×4, 8×4, e 8×8 no tamanho da matriz da arquitetura TRIPS. O intuito dos experimentos foi avaliar a escalabilidade e os ganhos de desempenho sob diferentes configurações da arquitetura. O melhor resultado foi de 15 operações por ciclo alcançado apenas pelo programa mgrid com uma matriz de 8×8 elementos de processamento e uma linha de execução.

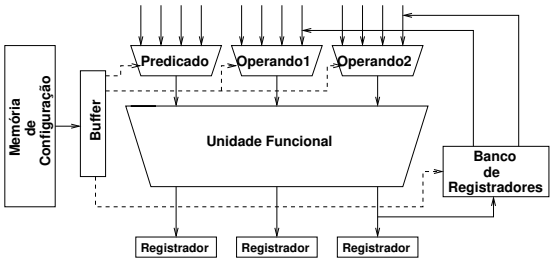
2.2.3 Arquitetura ADRES

ADRES [56,58] é uma arquitetura reconfigurável de granularidade grossa cujo foco principal se concentra na melhoria de desempenho de aplicações para sistemas embarcados. A arquitetura é composta por um processador VLIW, com L unidades funcionais, acoplado com uma matriz M de unidades funcionais reconfiguráveis. As unidades funcionais possuem bancos de registradores dedicados que são usados para armazenar valores intermediários. Assim como arquiteturas VLIW tradicionais, o processador VLIW usado por ADRES possui um conjunto de unidades funcionais e um banco de registradores que fornece operandos e recebe os resultados dessas unidades funcionais. Pelo fato de ser uma

arquitetura reconfigurável, o número de unidades funcionais é um parâmetro que depende da aplicação que será executada. A Figura 2.3 mostra duas visões da arquitetura ADRES: uma visão a partir do processador VLIW e do sistema de interconexão entre as unidades funcionais, Figura 2.3(a), e uma visão detalhada de uma unidade funcional, Figura 2.3(b).



(a) Visão geral da arquitetura ADRES. UF=Unidade Funcional. RT=Registrador Temporário.



(b) Detalhamento de uma unidade funcional.

Figura 2.3: Arquitetura ADRES.

Na Figura 2.3(b) é possível observar que uma unidade funcional provê suporte a operações predicadas, a fim de eliminar possíveis dependências de fluxo de controle. Os

resultados de uma UF podem ir diretamente para outras unidades funcionais através dos registradores temporários. A memória de configuração funciona como uma memória cache para configurações que foram e/ou podem ser utilizadas pela UF.

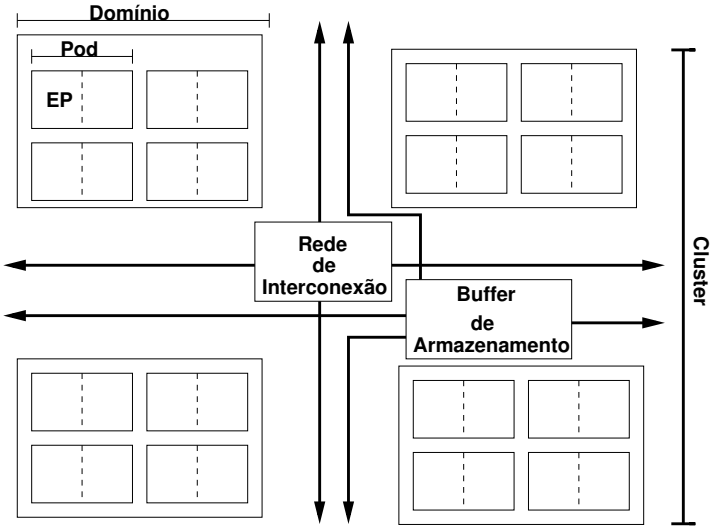
A geração de código para a arquitetura está a cargo do compilador DRESC [57, 58]. Além de reconhecer os recursos disponíveis no hardware, uma das principais funções desse compilador consiste em identificar núcleos de laços que consomem a maior parte do tempo de processamento do programa e mapear tais núcleos sobre a matriz de UFs reconfiguráveis. Para tanto, o compilador utiliza um algoritmo conhecido como *modulo scheduling* [67] a fim de minimizar o intervalo entre iterações de um laço. O compilador representa os recursos existentes na arquitetura através de um grafo, onde os vértices são os recursos e as arestas são as interconexões entre esses recursos.

Um fato interessante a observar nesta arquitetura é que o processador VLIW e a matriz de unidades funcionais trabalham independentemente de forma que o controle do programa pode ser trocado entre esses dois conjuntos de elementos. Os experimentos que avaliaram o desempenho da arquitetura ADRES utilizaram programas multímidias e de telecomunicações compilados utilizando o compilador DRESC. O número de operações por ciclo foi de 19 até 42 utilizando uma matriz 8×8 de UFs. A freqüência do processador assim como o número de registradores e a configuração de memória não foram mencionados pelos autores.

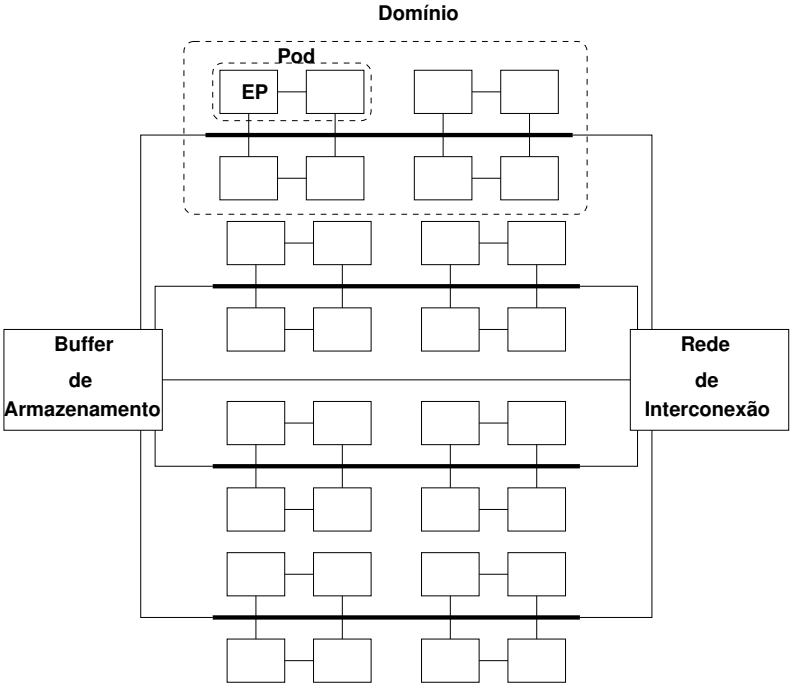
2.2.4 Arquitetura WaveScalar

WaveScalar [78, 79] é um modelo de execução e um conjunto de instruções de fluxo de dados. A diferença básica entre essa abordagem e propostas de arquiteturas de fluxo de dados anteriores é que WaveScalar consegue manipular semânticas de memória baseadas no modelo de execução von-Neumman em uma máquina de fluxo de dados. Isso é possível pois WaveScalar possui uma implementação descentralizada da técnica *token-store* que garante a coerência da execução das operações de memória.

Um processador baseado no modelo WaveScalar é chamado WaveCache [80]. A Figura 2.4 apresenta as principais características existentes em um WaveCache.



(a) Cluster de elementos de processamento.



(b) Sistema de interconexão dentro de um cluster.

Figura 2.4: Arquitetura WaveScalar.

O WaveCache possui uma matriz M de elementos de processamento que são utilizados pelas operações de um programa. O WaveCache é, na realidade, um processador que possui múltiplos elementos de processamento EPs) organizados hierarquicamente em *clusters* para redução de custos de comunicação dos operandos, Figura 2.4(a). O nível mais alto na hierarquia WaveCache é conhecido como cluster de processamento. Cada cluster possui quatro domínios, cada domínio possui oito EPs. Dois EPs formam um *Pod*. Os EPs em um *Pod* compartilham lógica de *bypassing* com o intuito de facilitar a execução de operações com dependência de dados. A Figura 2.4(b) apresenta os detalhes da interligação entre os elementos desses diferentes níveis da hierarquia. Adicionalmente, cada cluster possui uma memória cache de dados, interfaces para operações de memória e roteadores para comunicação com outros clusters. Cada EP possui uma unidade funcional, *buffers* para armazenar operandos e circuitos para controlar e executar operações. Todos esses recursos estão distribuídos ao longo de um *pipeline* de cinco estágios.

A arquitetura WaveScalar considera um programa como um conjunto de grafos de fluxo de dados que devem ser executados obedecendo as relações de dependência entre as operações. Diferente de arquiteturas orientadas a um contador de programa, uma operação WaveScalar envia seus resultados para operações consumidoras e essas executam assim que todos os seus operandos estejam disponíveis nas entradas de um EP. As operações WaveScalar são agrupadas em blocos chamados *Waves* que, por sua vez, são mapeados para os EPs do WaveCache. Assim que todas as operações de uma *Wave* terminam sua execução, uma nova *Wave* é carregada sobre os EPs. Para garantir a semântica correta de execução das operações, principalmente no caso de iterações de laços e operações de desvio, WaveScalar usa operações especiais, funções ϕ e ρ que mantêm a coerência dos dados e preservam a ordem de execução correta em caso de desvios no programa. Não há informações sobre a adoção ou implementação de um compilador específico para WaveScalar. Em [59,60], os autores mencionam um conjunto de algoritmos para escalonamento de instruções.

Experimentos com um simulador da arquitetura WaveScalar [80] consideraram aplicações com um única linha de execução e também com várias linhas de execução. A comparação foi realizada com um processador DEC Alpha EV7. Em [80], os autores

não mencionaram informações sobre a configuração de hardware das arquiteturas sob comparação e nem o compilador utilizado. Para aplicações com múltiplas linhas de execução, o desempenho de WaveScalar foi até 9 vezes melhor comparado à arquitetura Alpha.

2.2.5 Arquitetura Matrix

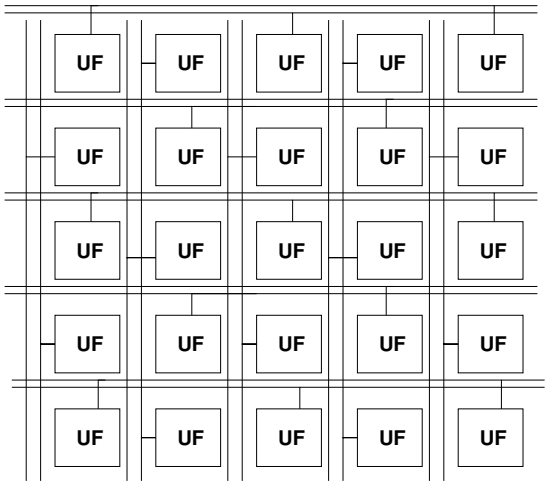
A arquitetura Matrix [61] é uma arquitetura reconfigurável que suporta configuração e distribuição dos recursos de hardware através das operações de um programa. A grande motivação para arquiteturas como Matrix é possibilitar a unificação de recursos para armazenamento e controle de dados e operações. A arquitetura é formada por uma matriz de unidades funcionais de 8 bits interconectadas através de um sistema de interconexão reconfigurável. Cada unidade funcional possui uma memória de 256 bytes, ULA, unidade de multiplicação e uma rede de interconexão hierárquica. A Figura 2.5 apresenta a visão geral da arquitetura Matrix e os principais recursos da uma unidade funcional.

A arquitetura utiliza *pipelines*, no nível das unidades funcionais, para a execução de operações. Os autores argumentam que um modelo de execução simples, baseado em *pipelines*, junto com uma rede reconfigurável que oferece muitas possibilidades de interconexão permite construir modelos de execução ainda mais complexos como *arrays* sistólicos, arquiteturas VLIW e modelos SIMD.

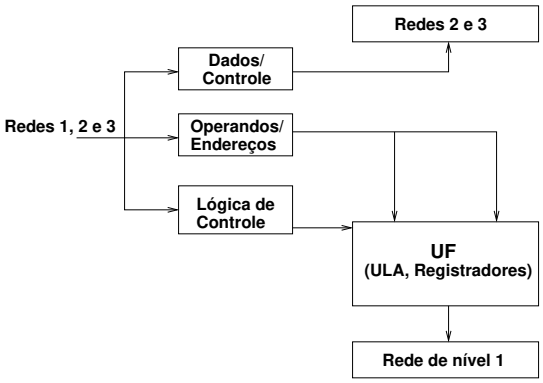
Uma implementação dessa arquitetura foi realizada em um processo de 0.5μ CMOS. Nessa implementação, os autores relatam que uma configuração com 100 unidades funcionais operando a uma freqüência de 100MHz, alcançou um desempenho de pico de 10^{10} operações (8 bits) por segundo. Os autores não informaram detalhes sobre ferramentas para geração de código na arquitetura Matrix.

2.2.6 Modelo Arquitetural EPIC

Diante das limitações de escalabilidade e complexidade presentes em processadores modernos que utilizam modelos de execução superescalar ou VLIW, a abordagem EPIC [70,71] propõe um novo paradigma para a construção de processadores que exploram



(a) Matriz de UFs e suas interconexões.



(b) Detalhe da UF e a hierarquia de interconexões.

Figura 2.5: Arquitetura Matrix.

paralelismo em nível das operações de um programa. EPIC é comumente definida como uma classe de arquiteturas de processadores que propõe o uso de técnicas avançadas de otimização de código junto com elementos de hardware que suportam essas otimizações de código. Dois exemplos de arquiteturas baseadas em EPIC são: a arquitetura Intel® IA-64 [27] e HPL-PD [47]. Para os propósitos deste capítulo, é interessante destacar as características do processador HPL-PD em virtude de ser um projeto com uma gama de informações disponíveis e implementa grande parte dos conceitos EPIC.

HPL-PD [46] é um processador parametrizável que implementa diversos conceitos do modelo EPIC. Assim como qualquer processador que segue a "filosofia" EPIC, HPL-

PD adota técnicas de compilação para extrair melhor desempenho das aplicações. A arquitetura possui um conjunto de unidades funcionais especializadas em: operações de memória, operações envolvendo inteiros e, por fim, operações de ponto flutuante. Bancos de registradores são os responsáveis pela interconexão dessas unidades funcionais. HPL-PD possui dois modos de execução: superescalar e EPIC. No primeiro modo, operações são buscadas na memória e são escalonadas dinamicamente para as unidades funcionais. No segundo modo, várias operações são organizadas em instruções. Esse modo possibilita que operações com dependências de dados (escrita-leitura) sejam codificadas na mesma instrução. Ainda sobre esse modo de execução, o processador HPL-PD busca uma instrução na memória e executa suas operações respeitando as dependências de dados existentes.

Independente do modo de execução, o processamento de uma operação HPL-PD consiste basicamente: 1) na leitura dos operandos da operação; 2) na execução da operação propriamente dita e, finalmente, 3) na escrita dos resultados na memória ou no registrador de destino. A Figura 2.6 apresenta os principais componentes da arquitetura HPL-PD.

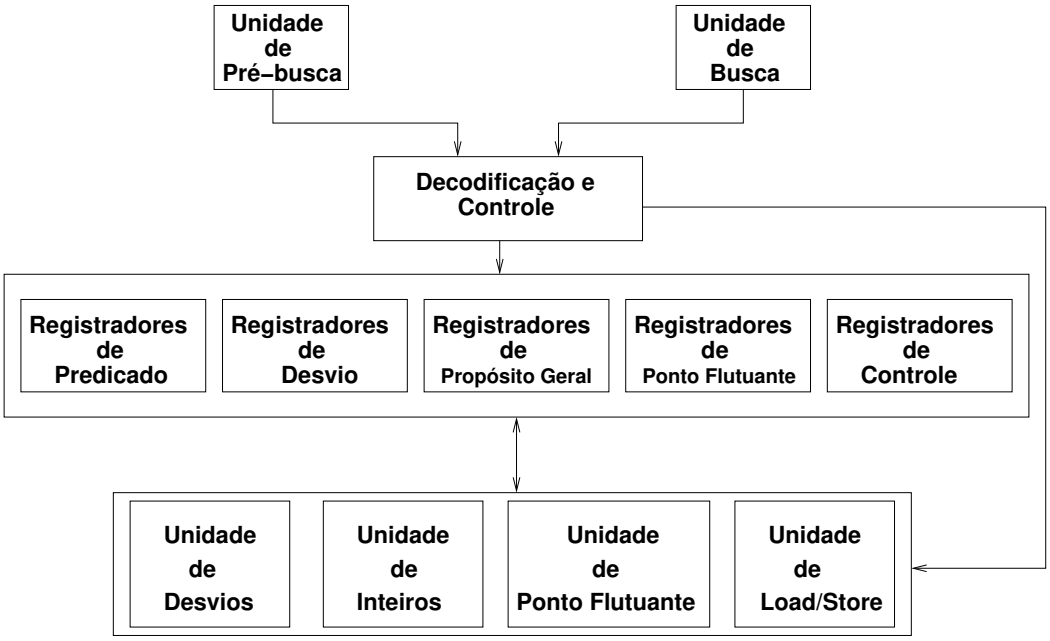


Figura 2.6: Visão geral da arquitetura HPL-PD.

A geração de código para o processor HPL-PD está a cargo do compilador Trimaran [16]. Dessa forma, o compilador Trimaran recebe uma descrição, especificada através da linguagem HMDES [37], da arquitetura HPL-PD a ser adotada e simula o código do programa considerando as características presentes nessa especificação.

2.2.7 Arquitetura PipeRench

A arquitetura PipeRench [17,35,72,73] procura resolver limitações relacionadas ao reuso de *pipelines* de hardware através da reconfiguração dinâmica e de uma abstração denominada "hardware virtual". O modelo de hardware virtual adotado por PipeRench é formado por um *pipeline* com uma determinada quantidade de estágios (*stripes*). Cada estágio consiste de 16 elementos de processamento e uma rede de interconexão para possibilitar a comunicação entre EPs. Cada EP possui uma unidade de lógica e aritmética, 8 registradores e utiliza 42 bits para sua configuração. Logo, com 16 EPs por estágio são necessários 672 bits para reconfigurar cada estágio da arquitetura. Uma visão mais detalhada da organização dos estágios e a interconexão entre EPs pode ser observada na Figura 2.7.

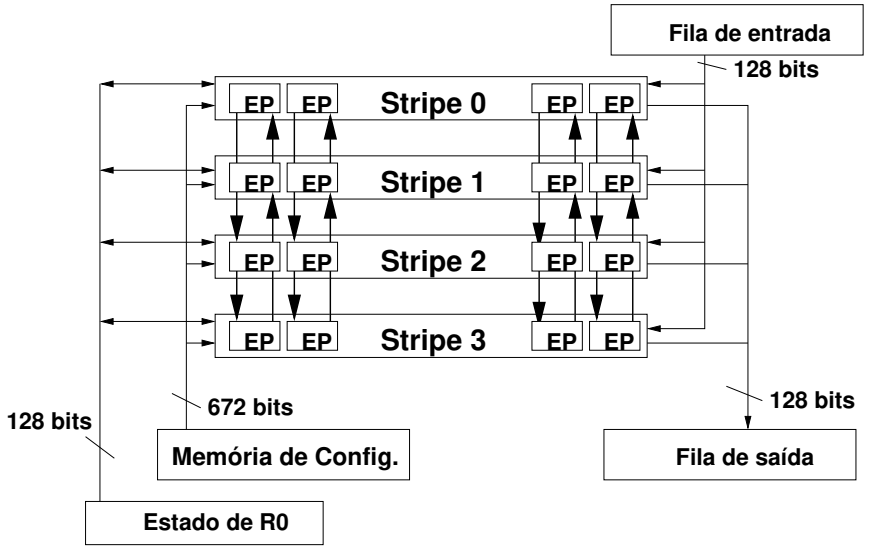


Figura 2.7: Diagrama de blocos da arquitetura PipeRench.

O processo de compilação de um programa para o hardware PipeRench ocorre da seguinte forma: 1) o compilador PipeRench [35] toma uma descrição da arquitetura virtual que inclui o número de EPs por estágio, número de registradores por EP, entre outros; 2) o compilador realiza o *parsing* do programa escrito em uma linguagem intermediária de fluxo de dados; 3) depois do *parsing*, o compilador desenrola laços e determina a largura de fios e a quantidade de lógica requerida por cada operação; 4) após isso, decompõe operadores de alto nível e aqueles que excedem o tempo de ciclo; 5) um passo de recomposição pode ser necessário se as decomposições introduzem ineficiências. A recomposição usa padrões para encontrar subgrafos que podem ser mapeados nos módulos já parametrizados.

A fim de demonstrar a factibilidade dessa arquitetura em uma implementação real, Schmit e outros [73] descrevem todos os passos realizados na implementação da arquitetura PipeRench utilizando tecnologia CMOS [66] em um processo de 0.18μ, área total do *die* de $7,3 \times 7,6\ mm^2$, $3,65$ milhões de transistores e uma freqüência de 120MHz. Além disso, Goldstein e outros [35] apresentam experimentos de desempenho com 9 núcleos de programas em que um protótipo PipeRench com freqüência de 100MHz, EPs de 8 bits e 8 registradores por EP alcançou *speedup* de até $189,7$ sobre uma arquitetura UltraSparc II de 300MHz.

2.2.8 Outras Propostas

Dado o grande avanço da área de arquitetura de computadores, é fácil imaginar que existe uma variedade de trabalhos que propõem novas arquiteturas de processadores. Esta seção apresenta algumas propostas de arquiteturas que, embora não estejam tão relacionadas diretamente com a arquitetura 2D-VLIW como aquelas apresentadas nas Subseções 2.2.1- 2.2.7, possuem algumas particularidades que serviram de base para o advento de arquiteturas mais inovadoras.

Como já é bem conhecido, o acréscimo de múltiplos elementos de processamento junto ao hardware a fim de aproveitar o paralelismo existente entre as operações de um programa não é algo recente. O surgimento de técnicas superscalares, máquinas vetoriais e arquiteturas de fluxo de dados datam dos idos dos anos 60 e 70. No início dos anos 80, Fisher [31] introduziu o conceito de máquina VLIW e apresentou a arquitetura ELI-52.

Essa arquitetura possui 16 UFs, cada uma contendo uma ULA e armazenamento local. Para fornecer operações para essas UFs, a máquina ELI-52 busca na memória instruções longas com mais de 500 bits. Nessa arquitetura, a detecção do paralelismo e a garantia da coerência da execução do código fica a cargo do compilador Bulldog.

No final dos anos 80, pesquisas a respeito de arquiteturas com capacidade de reconfiguração começaram a receber maior atenção pela comunidade científica da área. Nessas arquiteturas, o objetivo é repassar determinadas operações para serem processadas por um hardware reconfigurável. Por reconfigurável, deve-se entender a capacidade de reprogramar o hardware, via software, de maneira a especializá-lo para um processamento específico. A arquitetura PRISC [68] é um exemplo bem conhecido de arquitetura reconfigurável. PRISC estende o conjunto de instruções RISC através de operações específicas que são implementadas em Unidades Funcionais Programáveis em hardware. Essas UFPs são adicionadas à microarquitetura de maneira a manter os benefícios de alto desempenho das arquiteturas RISC e, ainda, gerar impacto mínimo no tempo de ciclo do processador. PRISC adota rotinas de compilação a fim de gerar automaticamente operações para UFPs. Essas rotinas analisam a complexidade do hardware para determinadas operações e interagem com ferramentas de síntese em hardware para selecionar operações que executarão com melhor desempenho se direcionadas para uma UFP. Pelo fato de serem adicionadas em paralelo com as unidades funcionais já existentes no processador, essas UFPs estendem a funcionalidade existente na via de dados. PRISC foi inicialmente vislumbrada para acelerar o desempenho de aplicações de propósito geral que executam sobre um hardware de baixo custo. Razdan e Smith [68] relatam *speedups* da ordem de 22% ao executar programas do benchmark SPECint92 em uma arquitetura MIPS $R2000$. Um ponto interessante a ser notado na apresentação dessa arquitetura, refere-se à mudança semântica nos campos da instrução MIPS a fim de suportar o hardware reconfigurável e a ausência de técnicas que exploram o paralelismo da aplicação considerando as novas unidades funcionais.

A arquitetura WormHole [8] é uma arquitetura dinamicamente reconfigurável. Mais especificamente, particiona os estágios de reconfiguração em sub-módulos de granulação fina que são colocados na arquitetura a medida que são necessários. Um dos

atrativos desta arquitetura está na possibilidade de minimizar o problema de reconfiguração dinâmica através de um esquema de controle distribuído. Para tanto, WormHole adota o conceito de *streams* de informações. Esses *streams* são compostos por um segmento de cabeçalho que transporta informações de configuração de cada unidade funcional, e um segmento de dados, que transporta os operandos. Basicamente, a arquitetura é organizada por uma matriz $M = 4 \times 4$ unidades funcionais. A medida que um *stream* propaga pela matriz, as informações de configuração de uma unidade funcional são retiradas do cabeçalho.

Em 2000, um consórcio de companhias ligadas à area de tecnologia: Sony®, Toshiba® e IBM®, anunciaram que modelos de execução tradicionais não são suficientes para manter a melhoria de desempenho desejada em face do aumento da complexidade das aplicações atuais. Esse consórcio de empresas propôs então uma nova arquitetura denominada Cell [45]. Tal arquitetura foi direcionada, inicialmente, para prover o melhor desempenho possível para aplicações multimídia e jogos. A arquitetura Cell é formada por um processador de propósito geral (PPE) e oito elementos de processamento (SPE). O PPE é a implementação de 64 bits da arquitetura PowerPC®, possui memórias cache de nível 1 e nível 2 e um *pipeline* de 23 estágios. Cada SPE possui uma memória interna (256KB) que armazena instruções e dados, um banco de registradores com 128 registradores, unidades funcionais para processamento de inteiros, ponto-flutuante, operações de memória e operações de desvio. Os SPEs podem executar até duas operações simultâneas enquanto que o PPE pode manipular até duas linhas de execução simultaneamente. No entanto, operações identificadas como vetoriais, são repassadas às unidades do mesmo nome e seguem um modelo de execução SIMD.

2.3 Relacionamento entre os Trabalhos da Área

Após apresentar as propostas de arquiteturas de processadores, pode-se agora analisá-las conjuntamente sob a ótica das características definidas na Seção 2.1.

Primeiramente, deve-se destacar a tendência das propostas com relação à organização dos recursos de processamento na forma de matrizes bidimensionais. Essa característica só não é observada nas arquiteturas HPL-PD e PipeRench. HPL-PD

aborda a execução de operações através de unidades funcionais especializadas que se comunicam através de bancos de registradores globais. PipeRench utiliza elementos de processamento interligados na forma de um *pipeline* de execução. As demais arquiteturas, de outra forma, possuem algum mecanismo para comunicação entre unidades funcionais sem a necessidade de enviar os dados para registradores globais. Como exemplo, as arquiteturas RAW, TRIPS, WaveScalar e Matrix utilizam roteadores e barramentos para estabelecer uma conexão dinâmica entre as unidades que executam operações, enquanto que a arquitetura ADRES faz uso de registradores temporários, distribuídos na matriz de unidades funcionais, para possibilitar a comunicação entre duas unidades.

Com relação ao modelo de execução, apenas a arquitetura WaveScalar utiliza um modelo orientado a fluxo de dados. Nessa arquitetura, os resultados das operações são enviados diretamente às operações consumidoras e essas, por sua vez, executam assim que seus operandos estão disponíveis. RAW e TRIPS, de outra forma, buscam operações na memória a cada ciclo de relógio e distribuem essas operações sobre os elementos de processamento de acordo com o escalonamento definido pelo compilador. O modelo de execução da arquitetura Matrix pode ser configurado como: SIMD, MIMD ou VLIW, de acordo com as características de cada aplicação. Independente do modelo de execução, cada unidade funcional dessa arquitetura possui um *pipeline* de 5 estágios. Na arquitetura ADRES, a execução das operações pode acontecer através do processador VLIW ou pelas unidades funcionais da matriz reconfigurável. O que acontece em geral é que operações que fazem parte de laços são mapeadas para a matriz reconfigurável enquanto que operações fora de laços são executadas pelo processador VLIW. Em se tratando da arquitetura HPL-PD, uma nova instrução é buscada na memória a cada ciclo e suas operações são escalonadas para as unidades funcionais. O escalonamento é determinado em tempo de compilação. Todos os operandos e resultados são obtidos a partir de bancos de registradores globais, de maneira que não há uma interconexão direta entre as unidades funcionais HPL-PD. Na arquitetura PipeRench, as operações são executadas por meio do *pipeline* de EPs.

No tocante à geração de código, apenas a arquitetura Matrix não apresenta informaçõs sobre algoritmos e/ou ferramentas de compilação para geração de código. As arqui-

teturas RAW, ADRES, TRIPS e HPL-PD já possuem compiladores específicos (RAWCC, DRESC, Scale e Trimaran, respectivamente) para a geração de código a partir de uma linguagem de alto nível. As arquiteturas WaveScalar e PipeRench, por outro lado, possuem algoritmos de escalonamento de instruções que, baseados em técnicas clássicas de escalonamento *list scheduling*, permitem simular a execução de um programa considerando os recursos existentes na arquitetura alvo.

A Tabela 2.1 resume as características presentes nas arquiteturas da Seção 2.2.

É interessante observar que a maioria das arquiteturas apresentadas recorrem a elementos de processamento formados por processadores completos (com *pipelines*, banco de registradores, memórias cache) para melhorar os níveis de desempenho dos programas. Além disso, a maior parte dessas arquiteturas utiliza sistemas de interconexão dinâmicos através de dispositivos roteadores e comutadores a fim de possibilitar a comunicação entre quaisquer unidades de processamento. Por fim, deve ser notado que a maioria das propostas adota modelos de execução onde o software de compilação exerce papel fundamental. Esse software é, muitas vezes, o único responsável pela utilização eficiente dos recursos do hardware e, por conseqüência, tem grande influência sobre o desempenho final de uma aplicação.

2.4 Considerações Finais

Um dos grandes desafios para arquiteturas que exploram paralelismo em nível de instrução é relacionar o acréscimo de mais elementos de hardware com os ganhos de desempenho obtidos pelas aplicações. Idealmente, esses ganhos devem ser diretamente proporcionais ao aumento dos elementos de hardware na arquitetura. O modelo de execução e as otimizações feitas pelo compilador são cada vez mais importantes para conseguir tais ganhos de desempenho.

Este capítulo apresentou as principais características de projetos de arquiteturas de processadores que utilizam múltiplas unidades para execução de operações. A escolha dos trabalhos apresentados na Seção 2.2 foi motivada pela proximidade de características com a arquitetura proposta nesta Tese. Um estudo mais detalhado envolvendo a comparação entre arquiteturas que exploram paralelismo em nível de instrução pode ser encontrado

em [24, 39]. Avaliações e comparações envolvendo características gerais e o desempenho de uma gama de arquiteturas de processadores reconfiguráveis podem ser encontrados em [7, 20]. Apesar das diversas características, existem ainda algumas lacunas a serem preenchidas. Exemplos dessas lacunas são: a ausência de topologias de interconexão que capturam a topologia de DAGs e técnicas de codificação de instruções que adotam novas caches para minimizar a latência e reduzir a área ocupada pela região de código do programa. O próximo capítulo apresenta as características da arquitetura 2D-VLIW bem como o paradigma de execução adotado por essa abordagem.

Arquiteturas	Organização dos Recursos	Interconexão	Modelo de Execução	Geração de Código
RAW	Matriz de EPs	Roteamento dinâmico	Cada EP executa sua operação segundo o escalonamento definido em tempo de compilação.	Compilador RAWCC
TRIPS	Matriz de EPs	Roteamento dinâmico	Cada hiperbloco de operações é mapeado sobre a matriz.	Compilador Scale
ADRES	Matriz de UFs e Processador VLIW	Registradores intermediários	Instruções carregam operações que são executadas ou pelo processador VLIW ou pela matriz de UFs.	Compilador DRESC
WaveScalar	Matriz de EPs	Roteamento hierárquico	Operações são executadas assim que os operandos estão disponíveis.	Algoritmo de escalonamento
Matrix	Matriz de UFs	Barramento entre UFs	Depende da aplicação, podendo ser SIMD, MIMD ou VLIW.	Não Informado
HPL-PD	Conjunto de UFs	Banco de Registradores Globais	Instruções longas carregam operações, com e sem dependências de dados, que são atribuídas às UFs.	Compilador Trimaran
PipeRench	Conjunto de EPs	Roteamento dinâmico e banco de registradores em cada EP	Operações executam sobre os estágios de execução previamente configurados.	Algoritmo de escalonamento

Tabela 2.1: Características das arquiteturas.

Capítulo 3

A Arquitetura 2D-VLIW

Atualmente, diversas propostas de arquiteturas de processadores têm explorado a obtenção de maiores níveis de paralelismo e, por conseqüência, melhor desempenho das aplicações através do acréscimo de mais recursos de processamento, do uso de canais de comunicação mais largos e de maiores quantidades de memória. Neste contexto, a arquitetura 2D-VLIW está focada no uso eficiente desses recursos por meio de novos arranjos dos elementos do hardware e de um modelo que privilegia a execução de várias operações a cada ciclo de relógio. Essa nova organização dos elementos de hardware vai ao encontro da geometria dos grafos de dependências de operações procurando maximizar o desempenho na execução do programa e melhorar a escalabilidade da arquitetura.

Este capítulo cobre as principais características da organização do hardware que compõe a arquitetura 2D-VLIW. O texto aborda a organização da via de dados 2D-VLIW, a caracterização das unidades funcionais, a estrutura dos registradores e memória e a rede de interligação entre as unidades funcionais. Ao final, há ainda seções apresentando o conjunto de instruções e o modelo de execução dessa arquitetura. Esse último é exemplificado através de operações extraídas de um programa real.

3.1 A Arquitetura 2D-VLIW

A arquitetura 2D-VLIW está baseada no pressuposto de que a organização dos elementos que compõem o processador associado a um modelo de execução que explora esses recursos eficientemente podem exercer um papel preponderante no aumento de

desempenho na execução das aplicações. Dois pontos inovadores na abordagem adotada por essa arquitetura residem na topologia e na organização desses elementos.

Na perspectiva da arquitetura 2D-VLIW, a computação dos dados de um programa possui duas características intrínsicas:

1. possibilidade de determinar um arranjo bidimensional das operações.

2. existência de computação local.

A primeira característica diz respeito à possibilidade de organizar grupos de operações de maneira bidimensional. Nesse espaço de duas dimensões, o eixo x é formado por operações independentes e que podem ser executadas em paralelo; o eixo y compreende a cadeia de dependência entre operações que deve ser obedecida a fim de garantir a semântica correta do programa. A segunda característica, computação local, está baseada no fato que uma região de código recebe alguns (poucos) parâmetros de entrada e realiza toda a sua computação com base nesses parâmetros. Tais parâmetros atuam geralmente como ponto de partida para o início da computação. Após isso, a região que recebeu os parâmetros segue seu processamento utilizando apenas os valores que foram e estão sendo definidos dentro da própria região. Aliás, essa segunda característica pode ser observada, com alguma facilidade, em funções e procedimentos de um programa. Uma função recebe alguns parâmetros de entrada, define variáveis locais a partir desses parâmetros de entrada e executa todas as suas operações internas tendo como base os valores definidos internamente.

A forma que a arquitetura 2D-VLIW adota para ir ao encontro das características mencionadas é através de uma organização distribuída dos elementos de hardware responsáveis pelo processamento das operações e armazenamento dos resultados e operandos. No contexto da arquitetura 2D-VLIW, esses elementos de processamento distribuídos são definidos como unidades funcionais e os elementos de armazenamento são chamados de registradores temporários. A Figura 3.1 apresenta uma visão geral da arquitetura 2D-VLIW. Nessa figura é possível observar a via de dados usada por uma instrução e a organização das UFs através de uma matriz 4×4. A matriz M de UFs 2D-VLIW é composta por todas as UFs da arquitetura, de maneira que $M = \bigcup_{i=1}^{16} UF_i$. Por simplicidade,

a hierarquia de memória e os registradores temporários não foram exibidos. No entanto, esses e outros elementos relevantes da arquitetura são apresentados com mais detalhes nas Subseções 3.1.1 à 3.1.5.

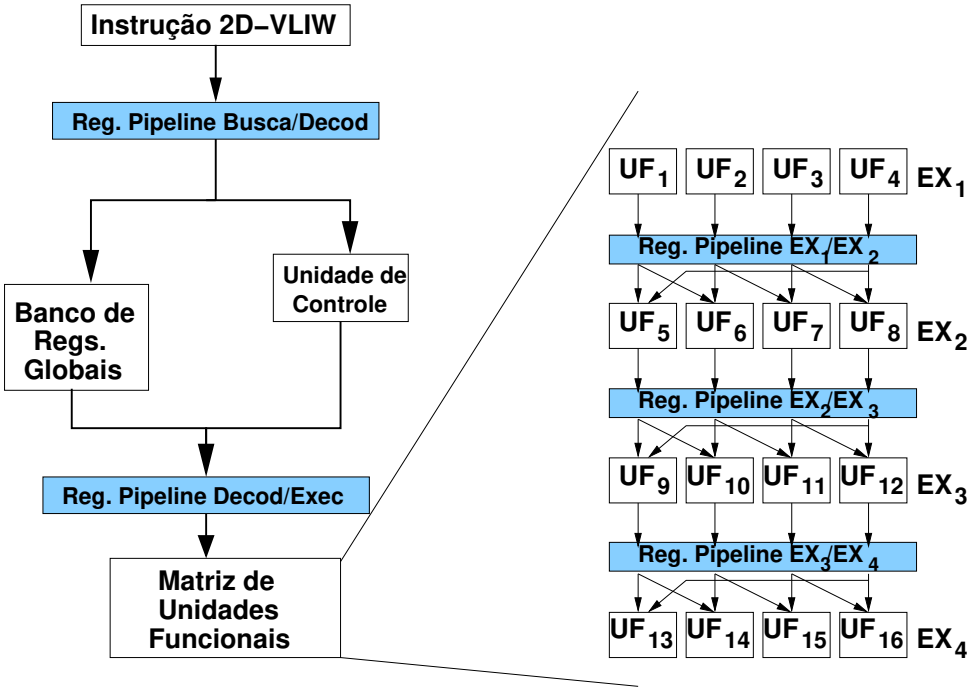


Figura 3.1: Via de dados da arquitetura 2D-VLIW.

Antes de apresentar os elementos que compõem a via de dados 2D-VLIW, deve-se ter em mente a definição de uma instrução nessa arquitetura. As instruções 2D-VLIW são formadas por operações simples (add, sub, ld, st, entre outros) que executam sobre a matriz de unidades funcionais. O número de operações em uma instrução é equivalente à quantidade de unidades funcionais na matriz. Assim, para a matriz 4×4 da Figura 3.1, uma instrução 2D-VLIW possui 16 operações. Caso não seja possível encontrar 16 operações no programa para preencher toda a matriz, operações especiais do tipo **nop** (operações nulas que não alteram os estados dos registradores) são inseridas.

A cada ciclo de relógio, uma instrução 2D-VLIW é buscada da memória e colocada no registrador de *pipeline* Busca/Decod. No estágio de decodificação, a instrução é decodificada e os operandos que usam registradores globais são repassados ao banco de

registradores globais. Depois disso, a instrução está pronta para ser repassada à matriz de unidades funcionais. A cada ciclo de execução, EX_1, EX_2, EX_3, EX_4, quatro operações de uma instrução são repassadas para as UFs. A instrução é levada de um estágio EX_i para um estágio EX_j, $j = i + 1,\ 1 \leq i < 4$, através dos registradores de *pipeline*. Deve ser destacado que esse modelo permite, de maneira geral, que uma operação atribuída à uma UF no estágio EX_j use os resultados das operações que executaram no estágio EX_i. Obviamente, essa característica vale apenas para os casos onde o tempo de processamento da operação no estágio EX_i é igual ou menor que o tempo que a instrução leva para ser enviada do estágio EX_i para EX_j. Note ainda que o sistema de conexão entre as UFs possibilita que uma UF do estágio EX_i forneça operandos para duas outras UFs no estágio EX_j. Essa conexão é feita por meio dos registradores temporários que estão distribuídos ao longo da matriz de UFs e são os responsáveis principais pela comunicação direta entre UFs.

Uma das razões para escolha desse modelo de execução, em contraposição a outros modelos que mapeiam todas as operações da instrução simultaneamente, reside na flexibilidade que o modelo adotado traz para explorar o paralelismo em regiões que vão além dos limites de blocos de controle (blocos básicos, hiperblocos ou superblocos[1]). Na abordagem onde todas as operações são enviadas simultaneamente, como é o caso da arquitetura TRIPS, há de se esperar o término de todas as operações para que um novo conjunto seja submetido para o hardware. De forma diferente, a abordagem adotada por 2D-VLIW permite que até quatro conjuntos de operações estejam executando simultaneamente sobre a matriz de UFs. Observe que, em muitos casos, esses conjuntos podem representar blocos de controle distintos.

É importante salientar que, embora a Figura 3.1 apresenta uma matriz com 4×4 UFs, o projeto da arquitetura 2D-VLIW não restringe o tamanho dessa matriz. Alguns experimentos apresentados no Capítulo 5 mostrarão, inclusive, o desempenho de programas executando sobre a arquitetura com diversas configurações de tamanho da matriz de unidades funcionais. Sem perda de generalidade, as referências ao tamanho da matriz

[1]Um superbloco é um conjunto de operações tal que o controle pode entrar apenas pela primeira operação enquanto que a saída pode acontecer por um ou mais locais dentro do conjunto. Diferente de hiperblocos, as operações dentro de um superbloco não são predicadas de forma que tal superbloco possui apenas operações de um caminho (traço) de controle.

ao longo deste capítulo consideram uma configuração de 16 UFs organizadas como uma matriz quadrada 4×4.

Uma discussão importante sobre a execução de operações na arquitetura 2D-VLIW concerne à geração de código. Ao observar a via de dados 2D-VLIW, pode-se inferir que, diferentemente de arquiteturas superescalares, não há unidades dedicadas ao escalonamento dinâmico de operações e à entrega de operações fora de ordem. A arquitetura aqui apresentada provê elementos de hardware simples e atribui à infraestrutura de compilação a tarefa de usar os recursos de maneira eficiente.

A arquitetura 2D-VLIW tem utilizado a infraestrutura de compilação Trimaran [16] como plataforma básica para geração de código. Um dos motivos para a escolha de Trimaran está na capacidade desse compilador de construir hiperblocos [53] e escalonar operações dentro desses hiperblocos para as UFs. Hiperblocos são agrupamentos de vários blocos básicos, acrescentando predicados em operações dentro de desvios condicionais na tentativa de maximizar o paralelismo em nível de operações. Essa característica é bastante adequada para arquiteturas como 2D-VLIW pois disponibiliza mais operações e, por conseqüência, mais possibilidades de paralelismo, do que uma organização baseada em blocos básicos. Além disso, Trimaran possui recursos para representar e simular arquiteturas que possuem múltiplas unidades funcionais. Esse recurso permite que programas sejam compilados e simulados considerando as características de arquiteturas descritas através da linguagem HMDES [37]. O Capítulo 4 apresenta as questões principais envolvidas na geração de código para 2D-VLIW.

As subseções a seguir discorrem, com mais detalhes, sobre as unidades funcionais da arquitetura 2D-VLIW, as características dos registradores, a rede de interligação entre UFs, a hierarquia de memória e o conjunto de instruções dessa arquitetura.

3.1.1 A Unidade Funcional 2D-VLIW

As unidades funcionais da arquitetura 2D-VLIW são as responsáveis pela execução das operações. As UFs são formadas por uma ULA, um banco de registradores temporários composto por 2 registradores, um registrador interno e interface com a memória. Os operandos de uma operação têm suas origens em três fontes possíves: o registrador global,

o registrador temporário ou o registrador interno da UF. Assim, uma UF_i é um conjunto composto por $\{rt0_i, rt1_i, RUF_i, ULA_i, IM_i\}$, em que IM_i é a interface de memória da UF_i. Além disso, algumas operações predicadas podem fazer uso dos valores em registradores de predicados. A Figura 3.2 mostra todos os blocos lógicos e sinais de controle dentro de uma UF 2D-VLIW.

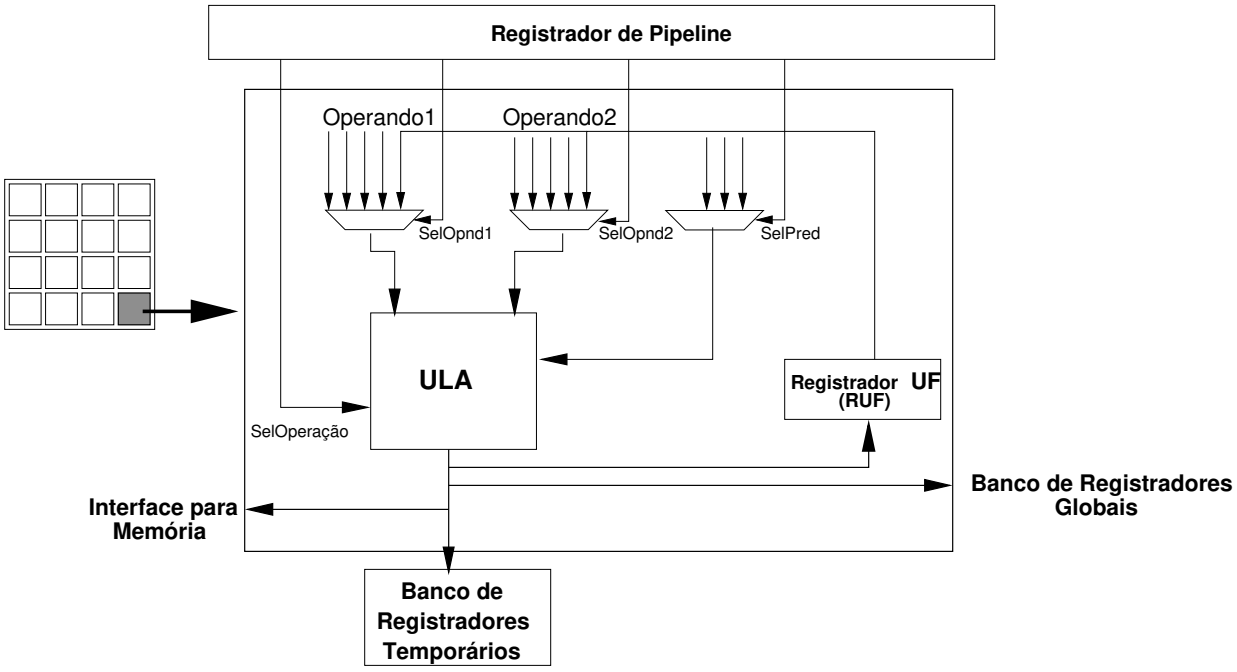


Figura 3.2: A unidade funcional 2D-VLIW.

Os resultados computados por uma UF podem ser escritos em um registrador temporário ou em um registrador global. Adicionalmente, o resultado sempre é escrito no RUF. A atividade de escrita no registrador global, entretanto, não ocorre da mesma forma que em um RT ou no RUF. Nesses dois últimos, há uma ligação direta entre a UF e o banco de RTs e RUF. De forma diferente, uma UF não possui uma porta de escrita específica para o banco de registradores globais. Na realidade, há uma porta de escrita compartilhada pelas UFs de cada linha da matriz. Assim, em um dado ciclo de execução, apenas uma operação por linha da matriz pode armazenar seus resultados no banco de registradores globais. Exceção à essa regra ocorre apenas para as UFs da última linha da

matriz. Nesse caso, há uma porta de escrita no banco de registradores globais dedicada para cada UF. O motivo para essa diferença entre portas de escrita para UFs em linhas diferentes da matriz se dá pela intuição de que UFs da última linha são locais prováveis para operações de retorno de uma função, da mesma forma que UFs internas são locais prováveis para operações intermediárias da função.

O resultado de uma operação escalonada na linha j e que executa em um ciclo i de relógio, estará disponível, através dos RTs e do RUF, para três outras operações no ciclo seguinte $i + 1$ uma vez que: duas operações, na linha $j + 1$, podem usar esse resultado através do registrador temporário enquanto outra operação, na mesma linha j, pode usar o resultado do RUF. Os sinais de controle SelOpnd1, SelOpnd2, SelPred e SelOperação são fornecidos a partir dos registradores de *pipeline*. Os dois primeiros sinais possibilitam a seleção da origem dos operandos de uma operação. O sinal SelPred seleciona valores em registradores de predicado. Por fim, o sinal de controle SelOperação informa qual operação deverá ser executada pela UF.

O uso de RTs possibilita um modelo bem simples de conexão entre UFs, além de minimizar o uso de registradores globais. Observe que essa redução no uso de registradores globais vai ao encontro da característica de computação local que foi discutida na Seção 3.1. Os registradores globais na arquitetura 2D-VLIW podem ser vistos como parâmetros de entrada para uma função e os registradores temporários, específicos para cada UF, funcionam como os valores intermediários produzidos e usados por essa função. Nessa analogia, a instrução 2D-VLIW pode ser entendida como uma função.

3.1.2 Registradores Temporários

Para dar suporte ao modelo de execução 2D-VLIW, que é o responsável pela execução de operações nas UFs da matriz, existem três classes de registradores dentro da arquitetura: os registradores globais, os registradores temporários e os registradores de predicado. A arquitetura prevê o uso de 32 registradores globais organizados em um banco de registradores no estágio de decodificação, 32 registradores temporários organizados em 16 bancos com 2 registradores cada e, por fim, 32 registradores de predicado.

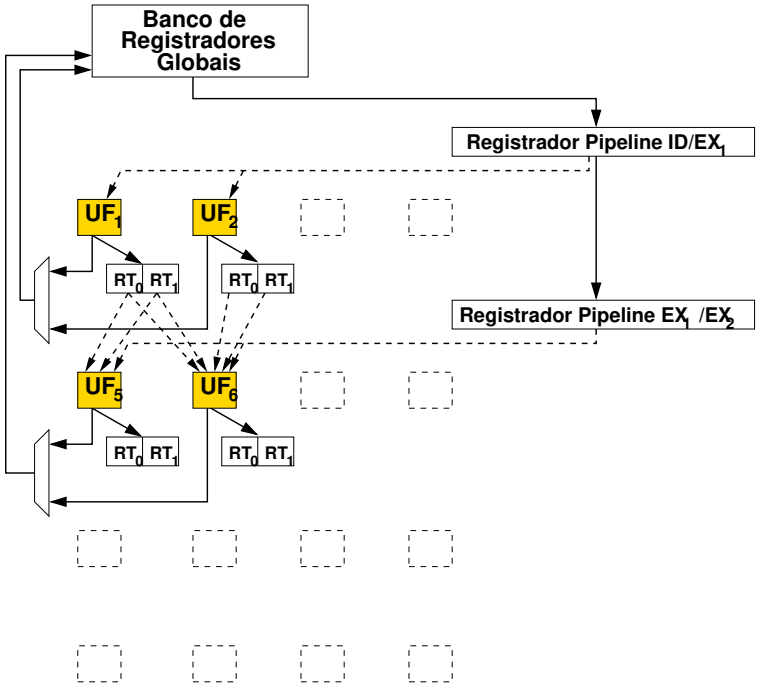


Figura 3.3: Interligação da UF com seu banco de registradores temporários.

A Figura 3.3 apresenta os detalhes da interconexão entre uma UF e um banco de registradores temporários. É interessante observar também a restrição com relação à escrita no banco de registradores globais. Observe que apenas uma UF pode armazenar resultados nesse banco de registradores em um determinado instante.

Na Subseção 3.1.1 foi apresentada uma analogia do uso dos registradores globais e temporários com os parâmetros de entrada e as operações executadas dentro de uma função. No contexto da arquitetura 2D-VLIW, essa divisão propicia duas vantagens importantes: associação dos operandos e resultados com os registradores de acordo com a semântica de uso das operações dentro de grafos acíclicos dirigidos (DAGs); a redução no número de portas de escrita e leitura no banco de registradores globais.

A primeira vantagem está relacionada com a capacidade da arquitetura de mapear DAGs inteiros sobre a matriz de unidades funcionais. Esse mapeamento só é possível porque os RTs podem armazenar os resultados de operações e fornecer esses resultados no ciclo seguinte ao armazenamento. Nesse caso, as operações produtoras e consumidoras dos

resultados podem estar agrupadas dentro da mesma instrução 2D-VLIW. O agrupamento de operações produtoras e consumidoras dentro de uma mesma instrução é possível pois os RTs são usados durante os estágios de execução da via de dados. Os registradores globais, por outro lado, podem armazenar apenas valores durante os ciclos de execução. A leitura dos valores é feita apenas no estágio de decodificação. Observe que os RTs funcionam muito bem como elementos de armazenamento dos valores internos de um DAG, assim como os registradores globais armazenam os operandos das operações raízes de um DAG ou, até mesmo, os resultados produzidos pelas operações que são folhas de um DAG.

O nome registrador temporário vem do fato que os valores armazenados geralmente são válidos apenas dentro do DAG. Considere duas operações op_m e op_n que mantêm uma relação de dependência $op_m \rightarrow op_n$, significando que op_n depende do resultado de op_m. Se a operação op_m é escalonada em uma UF da linha i, $1 \leq i < 4$ e, no ciclo j, armazena seu resultado em um registrador temporário, op_n pode ser escalonada para uma UF da linha $i + 1$ e ler o resultado de op_m no ciclo $j + 1$. Observe que op_m e op_n podem ser colocadas em uma mesma instrução e, mais importante, podem ser trazidas da memória em uma mesma transação de busca de instrução. Se, de outra forma, as operações op_m e op_n utilizam registradores globais, cada operação deve estar em uma instrução diferente, uma vez que a leitura dos registradores globais é realizada no estágio de decodificação, e a escrita nesses registradores é feita no estágio de execução.

A segunda vantagem é uma conseqüência direta da primeira. Já foi comentado que os operandos dos vértices raízes de um DAG são mapeados para os registradores globais e os valores intermediários de um DAG são armazenados em RTs. Como, geralmente, há mais vértices internos do que vértices raízes em um DAG, essa associação possibilita reduzir o número de portas de leitura e escrita no banco de registradores globais já que a maior parte das operações utilizam os RTs. Essa redução tem um impacto direto na complexidade de área e latência do banco de registradores globais.

Em geral, a complexidade de área de um banco de registradores é limitada, no pior caso, por $\mathcal{O}(n^2)$ [14] e a latência de pior caso é limitada por $\mathcal{O}(log\ pl)$. Considere pe =número de portas de escrita, pl =número de portas de leitura e $n = pe + pl$. Observe que o número de portas de leitura tem um impacto direto na área e latência do banco

de registradores. Diversas arquiteturas baseadas em múltiplas unidades de execução, em especial arquiteturas VLIW, possuem um número de portas de leitura limitado por $\mathcal{O}(k)$ (em termos exatos, $2 \times k$), onde k é a quantidade de unidades funcionais da arquitetura. Ao adotar registradores temporários e considerando que apenas raízes dos DAGs utilizam efetivamente o banco de registradores globais, tem-se a possibilidade de reduzir a complexidade de pior caso da latência desse banco de registradores. Esse pior caso passa a ser agora $\mathcal{O}(\sqrt{k})$ (em termos exatos, $2 \times \sqrt{k}$). A redução de k para $\sqrt{k}$ possibilita que até $\sqrt{k}$ operações que representam raízes de DAGs possam ser colocadas em uma mesma instrução 2D-VLIW. Como exemplo do impacto que essa redução pode trazer, considere uma arquitetura VLIW com 16 UFs e com 16×2 portas de leitura no banco de registradores globais. Note que esse mesmo banco de registradores para uma arquitetura 2D-VLIW com 4×4 unidades funcionais possui apenas 8 portas de leitura. Essa redução de portas de leitura permite também uma redução da área ocupada pelo banco global da ordem de 4 vezes e uma redução de latência de 40%, se for considerado que ambas as arquiteturas possuem o mesmo número de portas de escrita.

Uma primeira impressão sobre essa restrição no número de portas pode sugerir que tal redução acarreta um crescimento no código do programa e, por conseqüência, redução do desempenho final da aplicação. Esse crescimento do código seria devido à necessidade de dividir operações em duas ou mais instruções pelo fato de que tais operações ultrapassam o limite de portas disponíveis. No entanto, experimentos realizados com programas que realizam computação com números inteiros e de ponto-flutuante mostram que tal restrição não degrada o desempenho da aplicação já que a maior parte dos DAGs não possuem 4 raízes e, mesmo quando possuem, alguns operandos dessas raízes são valores imediatos e até valores que já estão armazenados dentro dos RTs. Em outros casos, os DAGs que têm mais de 4 raízes já possuem mais vértices do que o limite suportado por uma instrução (16 operações). Assim, esses DAGs já seriam obrigados a usar mais que uma instrução 2D-VLIW.

3.1.3 Interligação entre Unidades Funcionais

A interligação entre as UFs da arquitetura 2D-VLIW pode ser caracterizada como estática, de um único estágio e com conexões limitadas, similar às redes de malha usadas em multiprocessadores como *Goodyear Aerospace*, Intel® Paragon e o *J-Machine* do MIT [28]. Como já pôde ser observado pela Figura 3.1, as UFs são interconectadas através de registradores temporários de forma que uma UF na linha i, $1 \leq i < 4$, pode enviar seus resultados para duas outras UFs na linha $i+1$. A motivação principal para a adoção desse modelo de interligação é a similaridade com os padrões geométricos de um DAG. Considerando que os vértices de um DAG possuem, no máximo, duas arestas de entrada e que os vértices de um nível j estão interligados com outros vértices de níveis $> j$, o modelo de interligação da matriz vai ao encontro do padrão de interconexão do DAG. Além disso, um modelo de interligação como esse facilita a construção de algoritmos que tratam o mapeamento de um DAG inteiro sobre a matriz de UFs. Nesse caso, a matriz é representada como um grafo dirigido $G = (V, E)$, onde $V =$ conjunto das UFs da matriz e $E =$ conjunto das arestas de comunicação da matriz que interligam as UFs. A Figura 3.4 apresenta uma visão mais detalhada do sistema de interconexão de UFs na arquitetura 2D-VLIW.

A interconexão entre duas UFs pode ser representada como uma aresta de comunicação e_{ij}, sendo que $e_{ij} = (UF_i, UF_j)$ indica que as unidades funcionais UF_i e UF_j são interligadas no sentido $UF_i \longrightarrow UF_j$ através de um registrador temporário. Logo, $e_{ij} \in E$. A interconexão estática possibilita que o transporte de um dado da UF_i para a UF_j, através de uma aresta e_{ij}, seja realizado em tempo constante $\mathcal{O}(1)$. Se o mesmo resultado da UF_i deve ser usado por uma UF_l através de uma aresta $e_{il} = (UF_i, UF_l)$ mas, $e_{il} \notin E$, mesmo assim a comunicação entre essas duas UFs ainda é possível. Nesse caso, operações de cópia devem ser inseridas na matriz de UFs a fim de possibilitar o envio do valor de UF_i para UF_l. Nessa situação, a latência para comunicação de UF_i e UF_l passa a ser $\mathcal{O}(\sqrt{|V|})$.

Outro ponto que motiva a utilização desse sistema de interconexão é a facilidade para escalar de acordo com o número de UFs. Observando atentamente a Figura 3.4, pode-se construir matrizes de UFs a partir de matrizes menores, seguindo o mesmo padrão

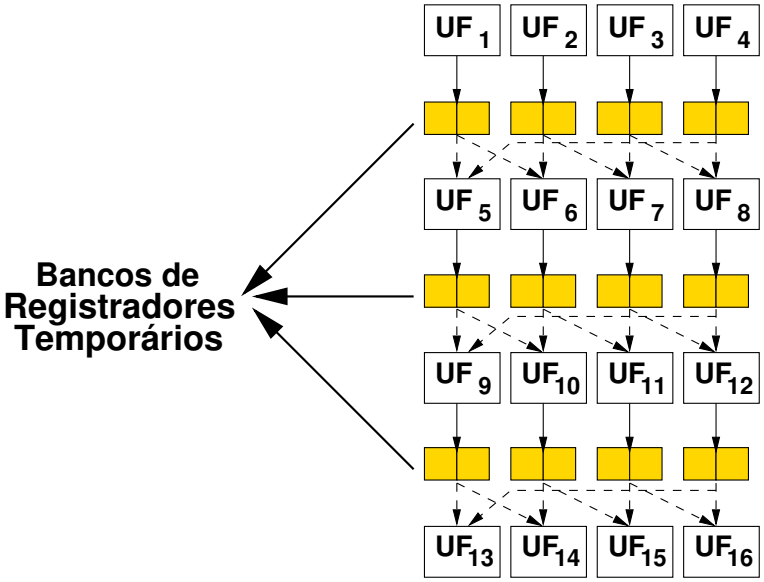


Figura 3.4: Interligação entre os elementos da matriz.

de interligação dos nós da matriz menor. Mais importante, o número de arestas $|E|$ da matriz cresce linearmente com o número de UFs. Dessa forma, o número de arestas de comunicação na arquitetura é limitado por $\Theta(|V|)$.

3.1.4 Hierarquia de Memória

As atividades (busca de instruções, armazenamento de dados e carregamento de dados em registradores) que envolvem o sistema de memória da arquitetura 2D-VLIW utilizam uma hierarquia de memórias cache similar ao que já ocorre com a maior parte dos microprocessadores usados atualmente. Há um primeiro nível de memória cache dividida em dados e instruções. Um segundo nível de memória cache que agrupa os dados e instruções em um local único. Por fim, o terceiro nível consiste na memória de trabalho onde o programa é carregado quando inicia a execução no processador.

Uma diferença para esse modelo tradicional de hierarquia de memória é que 2D-VLIW usa uma memória cache adicional no nível 1. Essa cache é denominada cache de padrões e funciona como um local de armazenamento rápido para a busca de padrões de instruções 2D-VLIW. Esses padrões são extraídos, em tempo de compilação, de instruções

não codificadas e funcionam como um mapa para decodificar instruções trazidas da memória. Assim, a hierarquia de memória presente na arquitetura 2D-VLIW pode ser representada conforme a Figura 3.5.

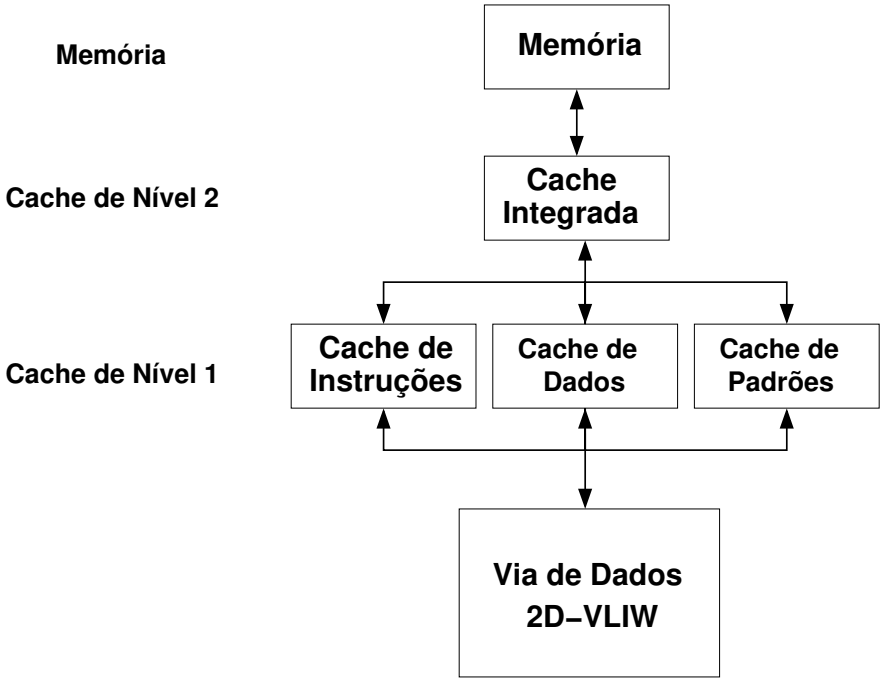


Figura 3.5: Hierarquia de memória 2D-VLIW.

Diferentemente das caches de dados e instruções, a cache de padrões não é utilizada durante o estágio de busca de instrução nem durante o estágio de execução. O padrão responsável pela representação completa das operações de uma instrução é buscado durante o estágio de decodificação pois, dessa forma, pode ser executado concomitante com a leitura dos registradores globais. Essa característica é bastante útil pois reduz o efeito dessa cache sobre o caminho crítico de execução de uma instrução, uma vez que não adiciona um novo estágio na via de dados. Na realidade, o que está ocorrendo é que a busca de um padrão e a decodificação da instrução estão sendo executadas em paralelo com a leitura de operandos no banco de registradores globais. As latências de acesso às memórias caches utilizadas nos experimentos envolvendo a arquitetura 2D-VLIW, assim como as latências de leitura e escrita nos bancos de registradores são as mesmas adotadas pelo compilador Trimaran

para acesso aos recursos da arquitetura HPL-PD [47]. A Figura 3.6 apresenta novamente a via de dados 2D-VLIW mas agora caracterizada com os elementos de memória.

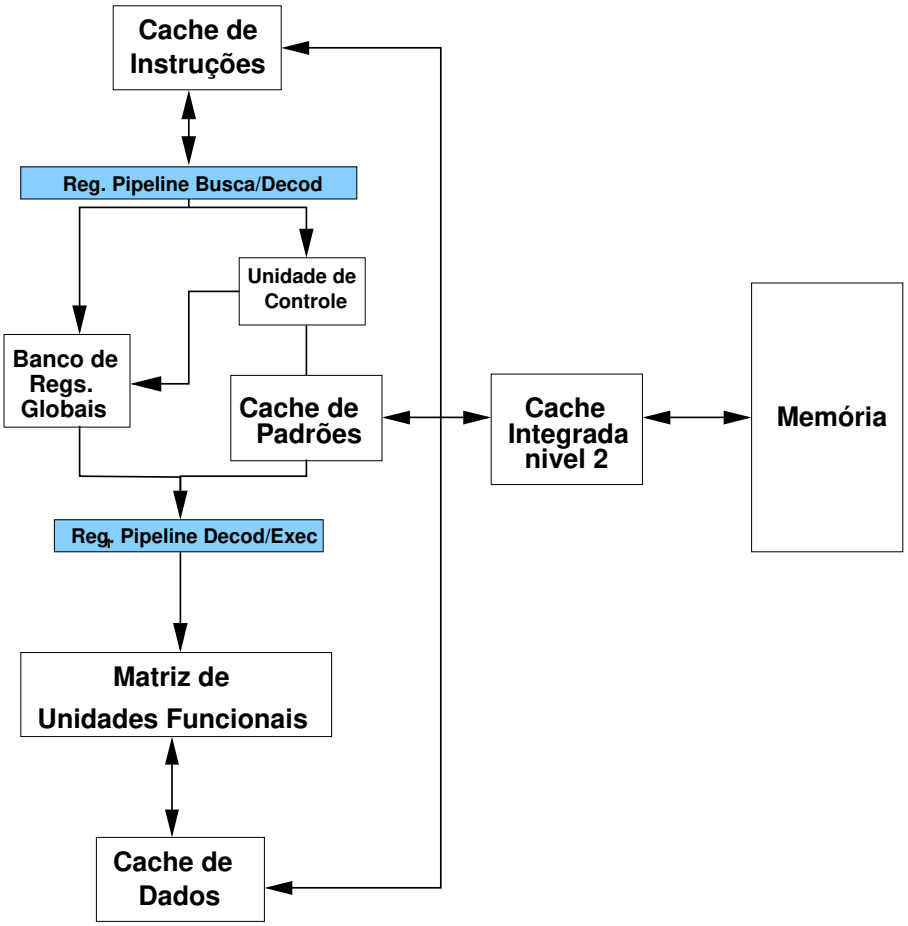


Figura 3.6: Via de dados 2D-VLIW e os elementos de memória cache.

A utilização de instruções codificadas, ao invés de instruções longas e simples, está pautada na necessidade de reduzir a latência na busca de operações de um programa e, com isso, minimizar os impactos dessa latência no desempenho do programa. A motivação para usar instruções codificadas e, por conseqüência, extrair padrões e armazená-los em uma memória cache junto à via de dados do processador tem origem em estudos iniciais no escopo do projeto 2D-VLIW apresentado nesta Tese. No entanto, há um trabalho de mestrado em desenvolvimento no LSC/IC-Unicamp pelo estudante Rafael Fernandes Batistella cujos objetivos são investigar algoritmos de codificação para instruções 2D-

VLIW e analisar o desempenho da cache de padrões. Sendo assim, esse tópico é tratado nesta Tese em caráter informativo e com o intuito de mostrar a hierarquia de memória vislumbrada para a arquitetura 2D-VLIW. Um detalhamento melhor sobre a técnica de codificação e o emprego da cache de padrões pode ser visto na Seção 4.5.

3.1.5 Conjunto de Instruções 2D-VLIW

Algumas regras e operações devem estar presentes em um conjunto de instruções adotado pela arquitetura 2D-VLIW a fim de adequar a execução do programa aos recursos da arquitetura e, além disso, possibilitar melhorias no desempenho da aplicação:

- **Operações com predicados**. Algumas operações que estão nos dois lados de um desvio condicional devem possuir predicados a fim de que o processador 2D-VLIW saiba, ao trazer da memória tais operações, distinguir efetivamente qual deverá ser executada. A determinação de quais operações devem possuir esses predicados depende do número máximo de instruções que utilizam a via de dados de um processador 2D-VLIW simultaneamente.

- **Operações ϕ**. Operações ϕ funcionam como um dispositivo multiplexador selecionando uma de suas entradas. Tais operações são usadas quando uma operação op_i possui um operando p que é criado por operações op_j e op_k no *delay slot* dos dois lados de uma operação de desvio. Sob essa situação, a operação ϕ deve ser executada antes de op_i, de acordo com a seguinte cadeia de dependência: $op_j,\ op_k \rightarrow \phi \rightarrow op_i$. Deve-se destacar que o significado adotado para a operação ϕ no contexto da arquitetura 2D-VLIW é o mesmo usado pelas funções ϕ no contexto de transformações SSA [26].

- **Operações com até dois operandos**. Essa característica deve estar presente em todas as operações, excetuando àquelas com predicados, a fim de que os DAGs possam ser mapeados diretamente sobre a matriz de UFs. Um conjunto de instruções com operações que aceitam mais que 2 operandos poderia formar, por exemplo, DAGs isomorfos para o grafo $K_{3,3}$ [9] que não são mapeados para a matriz de UFs da arquitetura 2D-VLIW.

- **Operação com mais de um valor imediato**. O sistema de codificação de palavras 2D-VLIW permite que apenas um valor imediato seja usado em cada operação. Dessa forma, operações que utilizam mais de um operando imediato devem ser divididas.

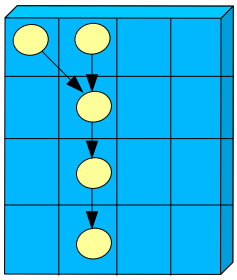
Em virtude da adoção do compilador Trimaran como plataforma básica para compilação e simulação de programas na arquitetura 2D-VLIW, os experimentos apresentados no Capítulo 5, envolvendo o desempenho de aplicações sobre o processador 2D-VLIW, utilizam o conjunto de instruções da arquitetura HPL-PD.

3.2 Modelo de Execução

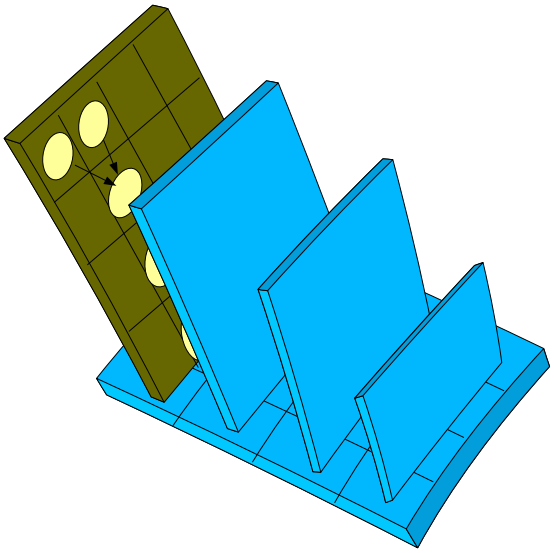
Como já mencionado na Seção 3.1, a execução de instruções de um programa sobre a arquitetura 2D-VLIW passa por registradores de *pipeline* ao longo da via de dados. Em cada ciclo de relógio, uma instrução 2D-VLIW é buscada na memória e colocada no registrador Busca/Decod. Quando alcança os estágios de execução, as operações que formam a instrução são executadas, em paralelo, de acordo com o número de UFs em cada linha da matriz. A maneira como as instruções de um programa percorrem a via de dados 2D-VLIW e são executadas é o que caracteriza o modelo de exercução 2D-VLIW.

A Figura 3.7 exemplifica a execução de uma instrução na via de dados 2D-VLIW. A Figura 3.7(a) exibe como as operações de um DAG podem ser organizados em uma instrução 2D-VLIW. A Figura 3.7(b) apresenta uma visão em perspectiva da execução dessa instrução após cada ciclo de relógio. Essa visão se dá pela presença dos registradores temporários entre unidades funcionais de linhas diferentes. Assim, pode-se visualizar três dimensões na execução de instruções 2D-VLIW: o eixo y é representado pelos sinais de controle enviados pelos registradores de *pipeline*; o eixo x indica o caminho dos dados que passam pelas UFs; o eixo z é formado pelos registradores temporários que armazenam os resultados para serem usados pelas operações dos próximos ciclos. Observe que a instrução tem o seu tamanho reduzido ciclo-após-ciclo. Isso indica que operações que já foram executadas não são repassadas para os próximos estágios de execução.

A Figura 3.8 apresenta uma outra visão da execução 2D-VLIW nas UFs de uma matriz 4×4. Nessa figura, pode-se observar o fluxo de dados e controle entre as UFs assim



(a) Representação de uma instrução 2D-VLIW como uma matriz de operações.



(b) Execução da instrução apresentada em 3.7(a) na via de dados 2D-VLIW

Figura 3.7: Exemplo de mapeamento e execução sobre a matriz de UFs.

como o paralelismo através de diferentes instruções 2D-VLIW (indicadas com diferentes cores nas entradas de cada UF). As UFs de uma mesma linha recebem os sinais de controle e dados simultaneamente. No entanto, há o envio dos dados e controle das unidades funcionais de uma linha i para uma linha $i+1$. É importante observar a execução paralela entre linhas diferentes da matriz. Enquanto as UFs da primeira linha executam operações de uma mesma instrução e com um conjunto de sinais de controle, as UFs das demais linhas estão também executando operações de diferentes instruções a partir de sinais de controle previamente armazenados nos registradores de *pipeline*. Note que os resultados enviados de uma UF da linha i para uma UF da linha $i+1$ só podem ser utilizados no

próximo ciclo de relógio desde que é apenas nesse próximo ciclo que os sinais de controle estarão disponíveis para as UFs da linha $i + 1$. Por simplicidade, a interconexão entre todas as unidades funcionais da matriz não é apresentada. Mesmo assim, pode-se entender como é essa interconexão ao observar as duas UFs mais à esquerda da primeira e segunda linhas da matriz (linhas contínuas). A interconexão entre essas UFs reflete o padrão de interligação adotado para as demais UFs da matriz (linhas tracejadas).

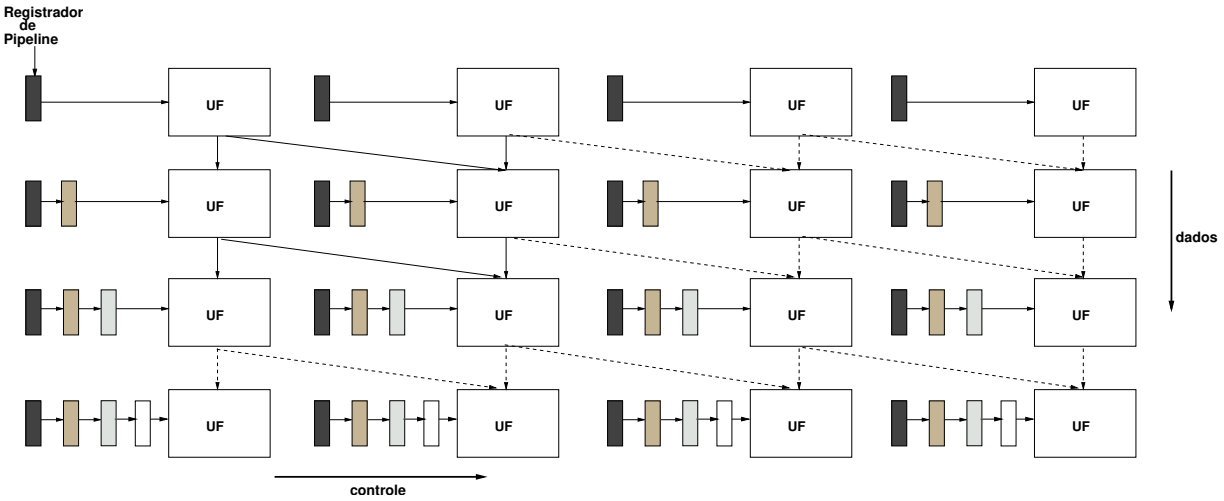
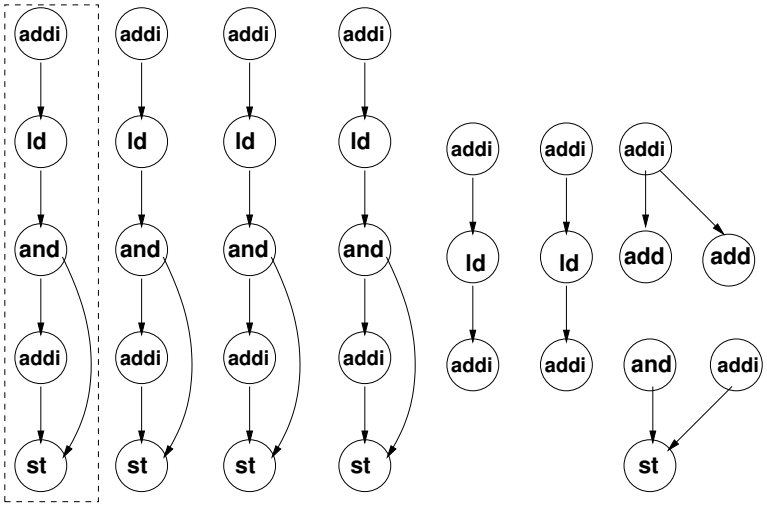


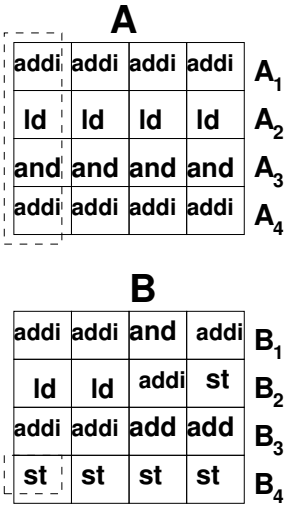
Figura 3.8: Estilo de execução 2D-VLIW: dados e controle são enviados para as UFs através do *pipeline*.

A partir dos exemplos mencionados nas Figuras 3.7 e 3.8, pode-se agora desvendar a execução de DAGs a partir de programas reais. A Figura 3.9 apresenta DAGs do programa 181.mcf do benchmark SPECint00 e as instruções 2D-VLIW formadas a partir do agrupamento das operações desses DAGs. As operações desses DAGs fazem parte do conjunto de instruções HPL-PD.

A Figura 3.9(a) exibe oito DAGs do programa 181.mcf e a Figura 3.9(b) apresenta como as operações desses DAGs são organizadas nas instruções 2D-VLIW A e B. Por agruparem o mesmo número de operações que as UFs da matriz, uma instrução 2D-VLIW pode ser também representada como uma matriz de operações. Operações que não possuem dependências são colocadas na mesma linha da instrução indicando que podem ser executadas em paralelo. A região tracejada na Figura 3.9(b) representa todo o DAG,



(a) DAGs do programa 181.mcf.



(b) Instruções 2D-VLIW obtidas após o escalonamento dos DAGs do programa 181.mcf.

Figura 3.9: DAGs do programa 181.mcf e suas instruções 2D-VLIW equivalentes.

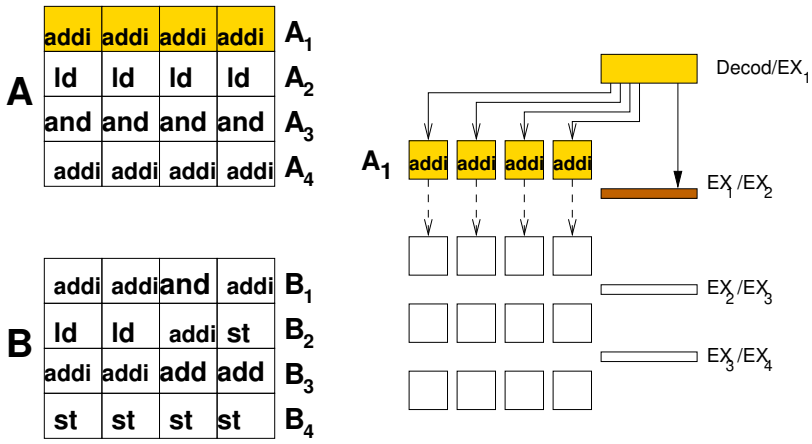
também tracejado, na Figura 3.9(a). Considerando a região tracejada na Figura 3.9(b), nota-se que os operandos da operação st são fornecidos pelas operações and e addi através do registrador temporário e do RUF, respectivamente. Isso pode ser confirmado ao observar que a operação st está localizada, na instrução B, uma linha abaixo da posição da operação and na instrução A. A operação st está também na mesma posição da operação addi. Neste ponto, deve estar claro que a posição de uma operação na instrução indica qual UF será usada. Operações que estão na mesma posição mas em instruções diferentes utilizarão a mesma UF mas em ciclos de relógio diferentes.

As Figuras 3.10 e 3.11 exibem o status do processamento das instruções A e B na matriz 2D-VLIW. Sem perda de generalidade, deve-se considerar que todas as operações são executadas em um ciclo de relógio. A fim de retratar o modelo de execução da arquitetura, o processamento da instrução será mostrado já no estágio EX_1. O estágio de busca de instrução é semelhante ao de arquiteturas que utilizam instruções longas na memória. O estágio de decodificação, em acréscimo à leitura dos registradores globais, realiza a busca do padrão da instrução junto a cache de padrões e, posteriormente, a decodificação da instrução. Essa atividade de decodificação necessita de um conhecimento prévio sobre o formato da instrução codificada e, por isso, será detalhada na Seção 4.5.

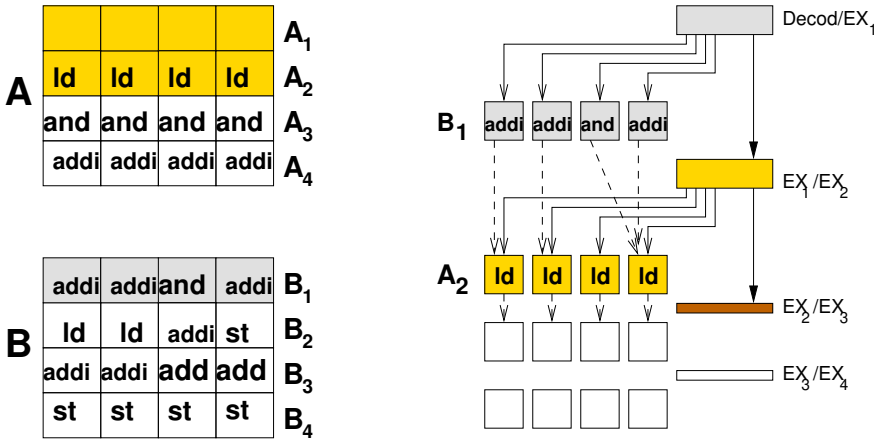
A Figura 3.10(a) representa o primeiro ciclo de execução na matriz de UFs. A primeira linha recebe as operações, operandos e sinais de controle do registrador de *pipeline* Decod/EX_1. As quatro unidades funcionais da linha A_1 executam as operações addi, addi, addi, addi da instrução A. As setas tracejadas indicam quais UFs receberão os resultados dessas operações. Note que os resultados são enviados para as UFs consumidoras obedecendo a rede de interligação entre UFs.

No segundo estágio de execução, Figura 3.10(b), as operações ld, ld, ld, ld da linha A_2 são executadas na segunda linha de unidades funcionais. Nesse mesmo ciclo de relógio, as operações da linha B_1 estão sendo executadas na primeira linha de UFs da matriz.

No terceiro estágio de execução, Figura 3.11(a), o registrador de *pipeline* EX_2/EX_3 armazena informações para execução das operações da linha A_3. Observe que, no mesmo ciclo de relógio, a segunda linha da matriz está executando as operações ld, ld, addi, st da



(a) Ciclo 1.

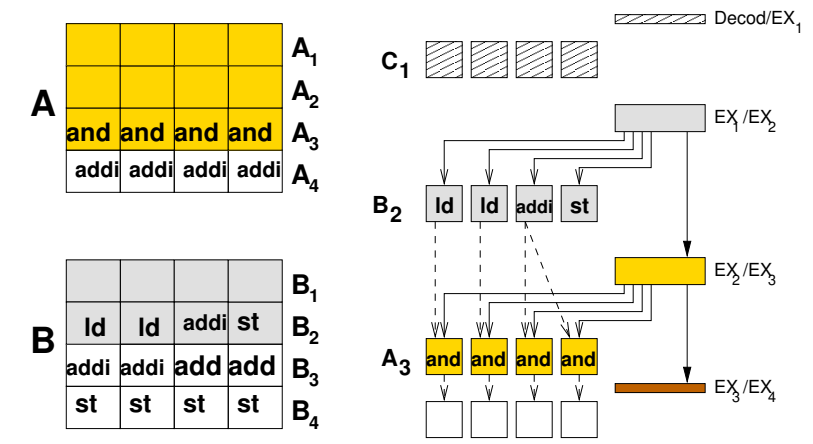


(b) Ciclo 2.

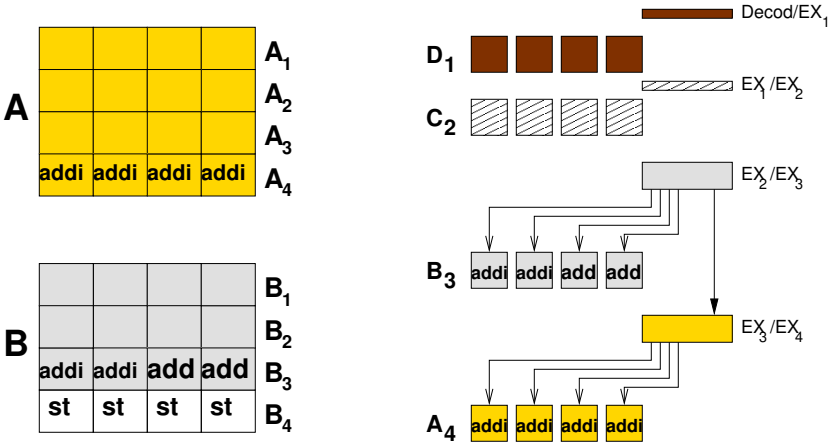
Figura 3.10: Estágios iniciais da execução das instruções A e B.

instrução B_2 e, agora, a primeira linha de uma nova instrução C inicia sua execução na matriz.

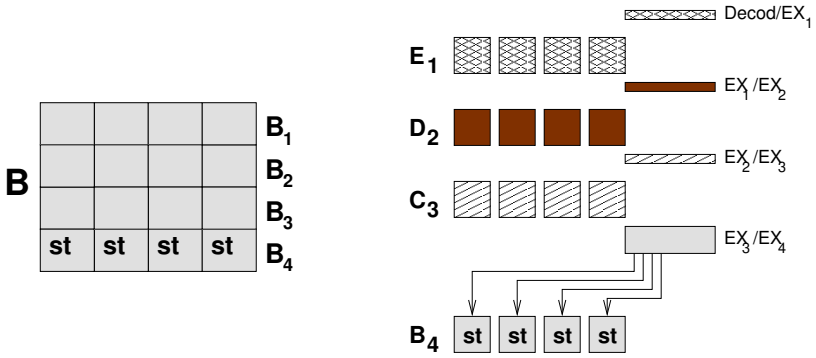
No quarto estágio de execução, Figura 3.11(b), as operações de A_4 são executadas na quarta linha, as operações de B_3 são executadas na terceira linha, as operações da linha C_2 da instrução C estão executando na segunda linha e uma nova instrução, D, inicia sua execução na matriz. O último estágio de execução é representado na Figura 3.11(c) onde quatro operações `st` da linha B_4 estão sendo executadas na última linha da matriz e todas as operações da instrução A já foram executadas.



(a) Ciclo 3.



(b) Ciclo 4.



(c) Ciclo 5.

Figura 3.11: Estágios restantes da execução das instruções A e B.

Analisando o quarto estágio de execução, é possível notar que a matriz de UFs está totalmente preenchida com operações de quatro diferentes instruções 2D-VLIW: A, B, C e D. Após o quarto estágio, uma instrução 2D-VLIW termina de executar sobre a matriz a cada ciclo de relógio, enquanto uma nova instrução inicia seu processamento. Por fim, é interessante observar que todas as operações executadas através dos cinco estágios de execução, Figuras 3.10(a), 3.10(b), 3.11(a), 3.11(b) e 3.11(c), representam exatamente todas as operações das instruções A e B definidas na Figura 3.9(b).

No exemplo do modelo de execução 2D-VLIW apresentado nas Figuras 3.10 e 3.11, os sinais de controle referentes às leituras/escritas em registradores, à execução das unidades funcionais e à ativação dos multiplexadores foram omitidos na tentativa de facilitar o entendimento do modelo. Pelo mesmo motivo, as latências de todas as operações do exemplo foram consideradas unitárias. Como usual, uma execução baseada em um conjunto de instruções mais realista considera operações com latências diferentes e até latências variáveis, como é o caso das operações de memória que buscam dados para serem armazenados em registradores. É importante destacar que esse exemplo, mesmo sendo simplificado, não prejudica o entendimento básico do modelo de execução 2D-VLIW. No caso de operações com latências não-unitárias, tais operações teriam o uso exclusivo da UF de acordo com a sua latência. Para as operações com latências variáveis, o compilador é o responsável por incluir operações no DAG que testam se o dado requisitado na memória já está disponível para ser utilizado. Se a latência da operação é variável a ponto de poder afetar a semântica de execução do programa, o processador 2D-VLIW deve sofrer um *stall* até que a operação seja finalizada.

3.3 Considerações Finais

Os elementos que constituem a arquitetura e o modelo de execução 2D-VLIW foram apresentados neste capítulo. É importante deixar claro que, embora não possuindo uma implementação em hardware, diversas características definidas nessa arquitetura estão baseadas em análises experimentais e na simulação de programas considerando o modelo de execução proposto. Mesmo assim, alguns recursos presentes na arquitetura, em

particular, a manipulação das operações de memória, ainda carecem de mais investigações e, portanto, podem sofrer mudanças quanto à abordagem aqui apresentada.

A presença de algumas características já têm se tornado usual nos processadores disponíveis atualmente. Tais características, mesmo não sendo abordadas diretamente neste capítulo, são factíveis para utilização na arquitetura 2D-VLIW. Algumas dessas são: predição de saltos, acesso especulativo à memória e pré-busca de dados.

De posse das informações a respeito da organização dos elementos de hardware da arquitetura pode-se, a partir de agora, discorrer sobre a geração de código para 2D-VLIW. As etapas que compõem a geração de código exercem importância fundamental para a arquitetura pois são as responsáveis pela distribuição das operações de um programa sobre os recursos da arquitetura e, como conseqüência, pelo uso eficiente desses recursos e obtenção de desempenho. As características envolvendo a geração de código para a arquitetura 2D-VLIW são apresentadas no Capítulo 4.

Capítulo 4

Geração de Código na Arquitetura 2D-VLIW

Um dos maiores desafios para processadores que exploram paralelismo em nível de instrução reside na dificuldade para gerar código que utiliza eficientemente todos os recursos presentes na arquitetura. Para suprir essa dificuldade, uma grande quantidade de técnicas de otimização de código, assim como novas abordagens para lidar com as diferentes fases que compõem a geração de código, estão presentes na literatura da área. Ao referir-se às arquiteturas com múltiplos recursos para processamento de operações é interessante observar que a dificuldade para geração de código está centrada, em muitos casos, nas otimizações realizadas sobre o código e nas etapas de escalonamento de instruções e alocação de registradores. Essas etapas são também responsáveis pelo uso eficiente dos recursos de um processador e, por conseqüência, pelo desempenho na execução da aplicação.

Diante desse contexto, o presente capítulo discorre sobre as técnicas e algoritmos adotados para gerar código tendo como base a arquitetura 2D-VLIW. Em particular, o capítulo está focado nas técnicas utilizadas nas atividades de escalonamento de instruções, alocação de registradores e codificação de instruções. As duas primeiras são bem conhecidas e estão presentes em qualquer projeto envolvendo a geração de código para processadores. A última atividade é mais específica e é tratada na arquitetura 2D-VLIW visando minimizar a sobrecarga com a busca de instruções longas na memória e a redução da área ocupada pelo programa. Este capítulo não cobre a definição de uma linguagem

de programação para 2D-VLIW e, tampouco, as etapas de compilação envolvendo análise léxica, sintática e semântica. Embora sejam técnicas muito importantes na definição de um compilador, a arquitetura 2D-VLIW não está voltada para uma linguagem de programação específica nem para um conjunto de instruções único. Mesmo assim, o texto do capítulo apresenta informações sobre a ferramenta de compilação adotada para geração de código considerando o modelo de execução 2D-VLIW.

4.1 Infraestrutura para Geração de Código

Em arquiteturas de processadores que exploram o paralelismo existente em programas, uma parte considerável do trabalho de detecção e extração do paralelismo reside no software que gera o código para essa arquitetura. Além da premissa básica de traduzir as especificações de um programa em uma descrição que seja inteligível para o processador, várias otimizações são executadas sobre o programa durante o processo de geração de código. Através dessas otimizações, os requisitos do programa são detectados e mapeados sobre os recursos oferecidos pelo hardware do processador. Para os propósitos deste capítulo, duas atividades realizadas durante a geração de código serão tratadas com maior destaque: o escalonamento de instruções e a alocação dos registradores.

A arquitetura 2D-VLIW transfere as atividades de escalonar as instruções do programa e alocar os registradores da arquitetura para serem tratadas exclusivamente durante o processo de geração de código. Essas tarefas têm um impacto direto no desempenho da aplicação em razão de serem as responsáveis por determinar quais operações do programa utilizarão quais recursos de hardware. Além disso, ambas as tarefas requerem algoritmos extremamente eficientes, uma vez que tanto o escalonamento de instruções quanto a alocação de registradores, em suas formas gerais, foram provados como pertencentes à classe de problemas $\mathcal{NP}$-completos[1] [33, 74]. A prova de que pertencem à essa classe indica que não há algoritmos de complexidade polinomial para resolver a situação de pior

[1]De maneira geral, problemas NP-completos são aqueles cujas soluções ótimas não podem ser determinadas em tempo polinomial. Formalmente, um problema NP-completo pertence ao conjunto de problemas que não podem ser resolvidos em tempo polinomial e, mais ainda, um problema em NP pode ser reducido a esse problema NP-completo.

caso desses problemas. Como conseqüência, há necessidade de lançar mão de técnicas heurísticas a fim de resolvê-los em tempo computacional viável.

Para o problema de escalonamento de instruções, uma heurística comumente empregada é aquela baseada no algoritmo de *list scheduling* [34, 40], em que o programa é dividido em regiões de código chamadas de blocos básicos e as operações desses blocos são organizadas em DAGs. Seja $G = (V, E)$ um DAG, o algoritmo de *list scheduling* computa o atraso máximo de cada vértice $v_i \in V$ para um vértice folha de G; seleciona os vértices ainda não escalonados, cujos ancestrais já foram escalonados, para participarem da lista de candidatos ($Cands$); O algoritmo escolhe o vértice com maior atraso, dentre os vértices da lista $Cands$, para ser escalonado em cada momento; O algoritmo termina quando a lista $Cands$ está vazia. Na versão proposta por [34], os autores argumentam que a complexidade do tempo de execução de pior caso é $\mathcal{O}(|V|^2)$ embora, na prática, o tempo de execução observado seja linear.

No caso da alocação de registradores, heurísticas baseadas no problema de coloração de vértices de um grafo de interferência têm se constituído na abordagem mais utilizada [10, 15]. O problema de coloração de vértices é, por si só, um problema $\mathcal{NP}$-completo.

Definição 5 *Um grafo de interferência $G = (V, E)$ é um grafo formado pelas variáveis de um programa que intereferem entre si. O conjunto V é formado pelas variáveis do programa. Existe uma aresta $e_{ij} = (v_i, v_j) \in E$ entre as variáveis v_i e v_j se elas interferem entre si. Isto é, se ambas estão vivas*[2] *em um determinado ponto do programa.*

Basicamente, heurísticas de coloração de vértices criam um grafo de interferência $G = (V, E)$ e tentam colorir os vértices de G com R cores, $R =$ conjunto de registradores do processador. Nessa coloração, cada vértice recebe uma cor e vértices adjacentes devem receber cores distintas. Pesquisas recentes em torno do problema de alocação de registradores têm mostrado que existem programas cujos grafos de interferência pertencem à classe dos grafos cordais [9]. Sob essa condição, a coloração pode ser realizada de maneira ótima em tempo polinomial [38, 65].

[2]Indica que o valor atual dessas variáveis ainda serão usados.

Apesar de amplamente discutidas na literatura da área, abordagens tradicionais para os problemas de escalonamento de instruções e alocação de registradores não se mostram atrativas para arquiteturas de processadores com diversas restrições arquiteturais, como é o caso de 2D-VLIW. Sobre tais arquiteturas, abordagens que utilizam regiões de código maiores que um bloco básico e que consideram a multiplicidade de recursos podem oferecer ganhos de desempenho para a aplicação. Diante de tal cenário, a adoção de algoritmos eficientes é ainda mais necessária.

No caso específico de 2D-VLIW, o problema de escalonamento de instruções foi tratado através da proposição de duas heurísticas distintas: um algoritmo baseado em isomorfismo de subgrafos e um algoritmo guloso baseado em *list scheduling*. O primeiro algoritmo representa tanto as operações de um programa quanto os recursos da arquitetura através de dois grafos distintos: o grafo do programa e o grafo da arquitetura. O objetivo do algoritmo de isomorfismo é obter um subgrafo do grafo da arquitetura que seja isomorfo ao grafo do programa. Esse subgrafo indica exatamente quais recursos do processador devem ser utilizados por cada operação. O algoritmo guloso baseado em *list scheduling* estende o algoritmo tradicional de *list scheduling* adaptando-o para representar a topologia da arquitetura e obedecer às restrições de interconexão.

O problema da alocação de registradores na arquitetura 2D-VLIW é abordado através de uma extensão do algoritmo original de escalonamento de instruções baseado em isomorfismo de subgrafos. Nesse novo algoritmo, o grafo da arquitetura é dotado com vértices representando os registradores temporários e globais e, com isso, possibilitar que a alocação de registradores seja realizada simultaneamente com a atividade de escalonamento.

Além de algoritmos específicos para escalonamento de instruções e alocação de registradores, a geração de código na arquitetura 2D-VLIW também trata do problema de codificar instruções longas em representações mais simples visando reduzir a latência de busca de instruções. A solução encontrada é baseada em técnicas de fatoração de operandos [5] e é executada logo após a atividade de escalonamento de instruções.

Como já apresentado no Capítulo 3, a geração de código para a arquitetura 2D-VLIW está a cargo do compilador Trimaran [16]. Mesmo não tendo sido desenvolvido

para gerar código considerando arquiteturas como 2D-VLIW, esse compilador oferece diversos recursos atrativos para arquiteturas com múltiplas unidades funcionais e que exploram o paralelismo em nível de instrução. Como exemplo, há a possibilidade de organizar o código em regiões que propiciam um número maior de operações que os blocos básicos tradicionais. Essa característica já é, por si só, uma fonte de motivação para o uso do Trimaran pois permite alcançar maiores níveis de paralelismo e, por conseqüência, melhor desempenho do que se blocos básicos fossem utilizados. Apesar de todos os recursos, a infraestrutura Trimaran não possui muitas opções de algoritmos para estudo dos problemas de escalonamento de instruções e alocação de registradores. Em se tratando de escalonamento de instruções, os algoritmos implementados originalmente e que podem ser utilizados por esse compilador são variações do algoritmo de *list scheduling* e do algoritmo de *modulo scheduling* [67]. Para o caso da alocação de registradores, o algoritmo utilizado é uma variação do algoritmo de coloração de vértices. Embora bastante úteis para arquiteturas baseadas no modelo EPIC, esses algoritmos carecem de várias adaptações para serem utilizados numa arquitetura como 2D-VLIW. Além dessas questões, uma das características fundamentais da arquitetura 2D-VLIW é a extração de padrões das instruções obtidas pelo escalonador de maneira a criar instruções codificadas menores que as originais. Essa funcionalidade não existe na infraestrutura Trimaran já que seu foco não abrange a codificação de instruções longas.

Diante do exposto, nota-se a necessidade de prover novos algoritmos para os problemas de escalonamento de instruções, alocação de registradores e codificação de instruções para gerar código para a arquitetura 2D-VLIW a partir do compilador Trimaran. As seções a seguir detalham e exemplificam esses algoritmos mostrando como as soluções são mapeadas para a arquitetura 2D-VLIW.

4.2 Algoritmo de Escalonamento Baseado em Isomorfismo de Subgrafos

O isomorfismo de subgrafos é uma forma geral do problema de casamento de padrões e uma generalização comum de diversos problemas relacionados à teoria dos grafos. Alguns exemplos desses problemas são: determinação de caminhos hamiltonianos,

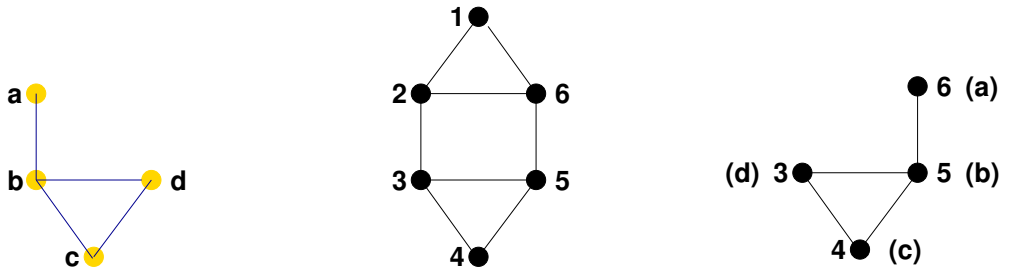
cliques, determinação de menor caminho, entre outros [29]. Variações do problema de isomorfismo de subgrafos têm sido usadas em problemas práticos como otimizações em compiladores, teste de circuitos integrados e processamento de imagens.

Definição 6 *No problema de isomorfismo de grafos, dois grafos* $G_1 = (V_1, E_1)$ *e* $G_2 = (V_2, E_2)$ *são ditos isomorfos, denotado por* $G_1 \cong G_2$ *se existe uma bijeção* $\varphi : V_1 \rightarrow V_2$ *tal que, para todo par de vértices* $v_i, v_j \in V_1$ *vale que* $(v_i, v_j) \in E_1$ *se e somente se* $(\varphi(v_i), \varphi(v_j)) \in E_2$.

Definição 7 *No problema de isomorfismo de subgrafos, o grafo* $G_1 = (V_1, E_1)$ *é isomorfo ao grafo* $G_2 = (V_2, E_2)$ *se existe um subgrafo de* G_2*, por exemplo* G'_2*, tal que* $G_1 \cong G'_2$.

Para algumas escolhas de G_1 e G_2 existe uma ordem exponencial de ocorrências de isomorfismo de forma que a tentativa de listá-las já foi demonstrado como sendo um problema $\mathcal{NP}$-completo [33]. Sem perda de generalidade, o grafo G_1 será doravante chamado de grafo de entrada e o grafo G_2 será denominado grafo base.

A Figura 4.1 mostra um exemplo de isomorfismo de subgrafos. As Figuras 4.1(a) e 4.1(b) mostram o grafo de entrada e o grafo base, respectivamente. O resultado do mapeamento das arestas e vértices do grafo G_1 para o grafo G_2 é o grafo resultante na Figura 4.1(c). Observe que $\varphi(a) = 6, \quad \varphi(b) = 5, \quad \varphi(c) = 4, \quad \varphi(d) = 3$.



(a) Grafo de entrada G_1. (b) Grafo base G_2. (c) Grafo resultante G'_2.

Figura 4.1: Exemplo do problema de isomorfismo de subgrafos.

O escalonamento de instruções tem se mostrado como uma das tarefas mais complexas na geração de código para a arquitetura 2D-VLIW. A razão para isso é que o

compilador é obrigado a capturar diversas características do hardware como tamanho da matriz e topologia de interligação, número de registradores globais e temporários e considerar tudo isso durante a fase de escalonamento. Nesse sentido, técnicas que consideram a geometria dos grafos de entrada assim como a topologia da matriz de unidades funcionais, como é o caso de algoritmos para o problema de isomorfismo de subgrafos, podem ser boas alternativas para resolução do escalonamento de instruções. Um esquema baseado em isomorfismo de subgrafos pode minimizar parte desta complexidade ao lidar com a matriz de UFs na forma de um grafo base e tentar associar os vértices e arestas do grafo de entrada com os vértices e arestas desse grafo base.

Dessa forma, dado um grafo de entrada G_1 e um grafo base G_2, o objetivo do escalonador é encontrar um subgrafo G'_2, $G'_2 \subseteq G_2$, tal que G'_2 é isomorfo a G_1. O subgrafo G'_2 representa os recursos de hardware (UFs e canais de comunicação) utilizados pelos vértices (operações do programa) de G_1. Esse mapeamento de operações em unidades funcionais gera uma instrução 2D-VLIW completa. A Figura 4.2 exemplifica o escalonamento de instruções na arquitetura 2D-VLIW como um problema de isomorfismo de subgrafos. Observe que as unidades funcionais, os registradores de *pipeline* e suas interligações, apresentados inicialmente na Figura 3.1, são agora representados como vértices (retângulos) e arestas no grafo base. O resultado do escalonamento é uma instrução 2D-VLIW, em que cada posição da instrução possui uma operação que será executada à medida que a instrução percorre a matriz de UFs.

Problemas de isormofismo de subgrafos pertencem à classe de problemas $\mathcal{NP}$-completos [82]. Com isso, adotar algoritmos eficientes é um requisito primordial para utilizar soluções que resolvam esses problemas. No escalonamento de instruções para a arquitetura 2D-VLIW, utiliza-se a biblioteca VF para isomorfismo de grafos [22, 23]. Essa biblioteca possui a implementação de alguns algoritmos clássicos para isomorfismo de grafos e subgrafos, incluindo um algoritmo, também denominado VF, definido pelos próprios autores da biblioteca. O motivo para a adoção dessa biblioteca é utilizar uma de suas implementações para prover o escalonamento de instruções 2D-VLIW. Algumas otimizações foram adicionadas ao código da biblioteca VF a fim de tornar seus algoritmos aptos para utilização na atividade de escalonamento de instruções. Além de um conjunto

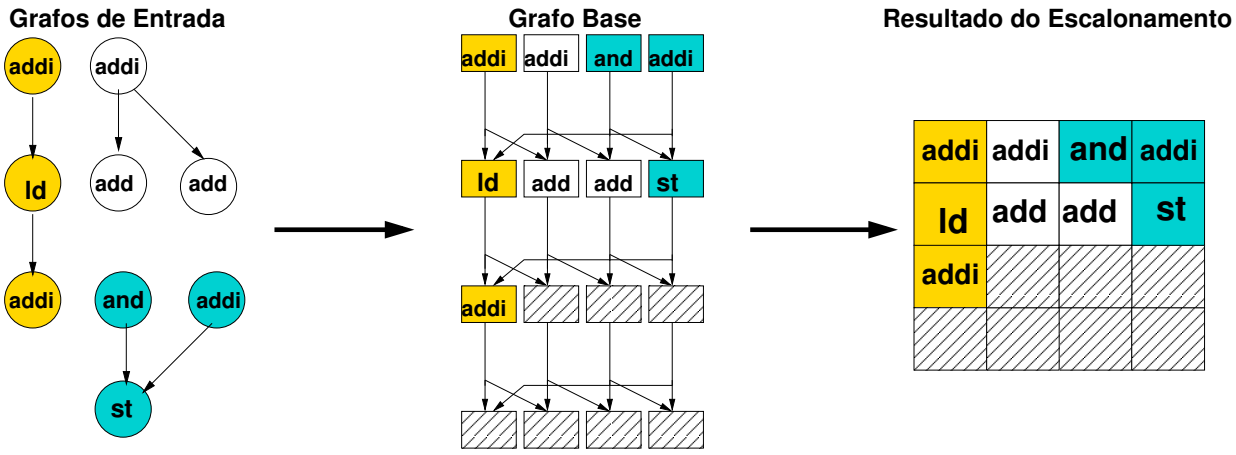


Figura 4.2: Escalonamento de instruções baseado em isomorfismo de subgrafos.

de três heurísticas que procuram facilitar a determinação de um isomorfismo, uma das otimizações principais consiste em restringir o tempo de processamento do algoritmo na tentativa de forçá-lo a apontar uma solução dentro de um limite de tempo estabelecido. Caso o isomorfismo não possa ser determinado dentro dessa restrição de tempo, heurísticas visando à melhoria do desempenho do algoritmo devem ser executadas. Através de experimentos realizados, foi possível observar que essas otimizações foram bastante úteis ao escalonar DAGs com mais de 100 nós.

Os parâmetros de entrada para o escalonamento de instruções 2D-VLIW baseado em isomorfismo de subgrafos são DAGs, gerados pelo compilador Trimaran, e um grafo base. Além da indicação da dependência entre vértices, esses DAGs carregam consigo informação de latência das operações e indicação de parâmetros de entrada para chamadas de funções e procedimentos. De posse desses parâmetros, o escalonador 2D-VLIW realiza o isomofismo de cada DAG para o grafo base. O Algoritmo 1 resume os passos principais do algoritmo de escalonamento de instruções 2D-VLIW baseado em isomorfismo de subgrafos.

A função *subg_iso_sched*() (linha 1) encontra um subgrafo G'_2, $G'_2 \subseteq G_2$, isomorfo para G_1. Esse procedimento usa o algoritmo VF para determinar se existe o isomorfismo entre os grafos de entrada e o grafo base. Caso o subgrafo G'_2 não seja encontrado (linhas 2-8), o escalonador escolhe uma heurística (linhas 3-6) com base no valor da

Algoritmo 1 Algoritmo de escalonamento de instruções 2D-VLIW baseado em isomorfismo de subgrafos.

ENTRADA: Grafo de entrada G_1 e grafo base G_2.
SAÍDA: Conjunto de instruções 2D-VLIW.
Sched(DAG: G_1, GRAFO BASE: G_2)
1) $G'_2 = subg\_iso\_sched(G_1, G_2, \&tag)$;
2) **while** $(G'_2 == NULL)$
3) **switch**(tag)
4) case 0 : $topological\_order(G_1)$;
5) case 1 : $base\_graph\_resize(G_2)$;
6) case 2 : $DAG\_global\_vertices(G_1)$;
7) $G'_2 = subg\_iso\_sched(G_1, G_2, \&tag)$;
8) **end while**
9) $create\_2D\text{-}VLIW\_instruction(G'_2)$;

variável *tag* e executa essa heurística sobre um dos parâmetros de entrada. A heurística *topological_order*() (linha 4) realiza a ordenação topológica no DAG de entrada G_1. A heurística *base_graph_resize*() (linha 5) redimensiona o grafo base com o intuito de oferecer mais opções de combinação para o grafo G_1. A última heurística, $DAG\_global\_vertices$() (linha 6), transforma o DAG G_1 em um grafo G'_1 mais flexível. Esse novo grafo tem a característica de poder utilizar regiões de G_2 que não eram consideradas por G_1. Note que apenas uma heurística é executada depois de cada tentativa sem sucesso da função *subg_iso_sched*(). Um dos motivos para utilização dessas heurísticas após a função *subg_iso_sched*() é que, para determinados grafos de entrada, as heurísticas podem acarretar um maior tempo de processamento por parte do isomorfismo (*base_graph_resize*()) ou mesmo, uso demasiado de recursos ($DAG\_global\_vertices$()).

Após o isomorfismo ser finalmente determinado, as operações do grafo G_1 (ou G'_1), junto com as UFs do subgrafo G'_2, irão compor a(s) instrução(ões) 2D-VLIW criada(s) pela função $create\_2D\text{-}VLIW\_instruction$() (linha 9). As três heurísticas descritas nesse algoritmo serão detalhadas nas Subseções 4.2.1, 4.2.2 e 4.2.3.

No algoritmo VF, função *subg_iso_sched*() no Algoritmo 1, dados dois grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, um mapeamento $M \subset V_1 \times V_2$ é um isomorfismo se e somente se existe uma função bijetora que preserva a estrutura dos dois grafos, de forma que M mapeia cada ramo do grafo G_1 em um ramo do grafo G_2 e vice-versa. De

maneira mais específica, o processo de casamento entre os grafos pode ser descrito através de uma representação do espaço de estados. Cada estado s, durante o processamento do algoritmo, pode ser associado a uma solução parcial $M(s)$ que contém apenas um subconjunto dos componentes da função de mapeamento M. Uma solução de mapeamento parcial identifica univocamente dois subgrafos de G_1 e G_2, $G_1(s)$ e $G_2(s)$, obtidos por meio de uma seleção dos vértices de G_1 e G_2 incluídos nos componentes de $M(s)$, e os ramos conectados a eles. O algoritmo termina com um subgrafo válido quando todos os vértices de G_1 já foram selecionados e o estado final constitui um isomorfismo entre G_1 e o subgrafo de G_2. Se o algoritmo termina depois de ter selecionado todos os vértices de G_2 e o estado final não constitui um isomorfismo de subgrafos, significa que não existe um subgrafo em G_2 que seja isomorfo para G_1. O Algoritmo 2 apresenta uma versão resumida da função *subg_iso_sched*(). Essa descrição contém os passos principais responsáveis pela determinação do subgrafo isomorfo ao grafo de entrada.

Algoritmo 2 Algoritmo de isomorfismo de subgrafos.

ENTRADA: Grafo de entrada G_1, grafo base G_2 e *tag*.
SAÍDA: Subgrafo G'_2.
subg_iso_sched(DAG: G_1, GRAFO BASE: G_2, TAG: &*tag*)
1) $sched\_nodes = 0$;
2) $node_{G1} = next(G_1)$;
3) $node_{G2} = next(G_2)$;
4) **while** ($sched\_nodes < |V_1|$ && $sched\_nodes \geq 0$)
5) **if** $IsFeasible(node_{G1}, node_{G2})$
6) $G'_2 = AddUsefulPair(node_{G1}, node_{G2})$;
7) $sched\_nodes = sched\_nodes + 1$;
8) $node_{G1} = next(G_1)$;
9) $node_{G2} = next(G_2)$;
10) **else**
11) **if** ($node_{G2}.index < |V_2|$)
12) $node_{G2} = next(G_2)$;
13) **else**
14) $backtrack()$;
15) **end if**
16) **end if**
17) **end while**
18) $tag = evaluate(G'_2, G_1, G_2)$;
19) **return** G'_2;

A cada iteração do laço (linhas 4-17), o algoritmo verifica se o par de vértices selecionados (função $next()$ nas linhas 2-3, 8-9 e 12), $node_{G1}$ e $node_{G2}$ são soluções parciais factíveis para o isomorfismo. A cada par de vértices factível encontrado (função $IsFeasible()$ na linha 5), o subgrafo G'_2 é incrementado (função $AddUsefulPair()$ na linha 6) com as informações sobre os vértices de G_1 e G_2 que fazem parte desse par. Um par de vértices é factível se possuem as mesmas características (mesmo grau de entrada, grau de saída, etc.) e contribuem para uma solução parcial do isomorfismo. Se um par de vértices não é uma solução factível, deve-se verificar se há ainda outros vértices de G_2 que podem ser pesquisados (linhas 11 e 12). Caso todos os vértices de G_2 já tenham sido testados, deve-se realizar uma ação de *backtracking* (linha 14) a fim de que um vértice previamente escalonado de G_1 escolha outro vértice de G_2 como par. O algoritmo executa enquanto existirem vértices de G_1 que ainda não foram escalonados (linhas 4-17). Antes do algoritmo finalizar, a função $evaluate()$ analisa o conteúdo do subgrafo G'_2 a fim de detectar se o isomorfismo não pôde ser concluído e, nesse caso, retornar o resultado para a variável *tag* (linha 18). Caso o isomorfismo não seja detectado, a função $evaluate()$ analisa os grafos G_1 e G_2, assim como o valor atual da variável *tag* para indicar um novo valor de retorno para essa variável e, como conseqüência, propor a utilização de uma das heurísticas do Algoritmo 1.

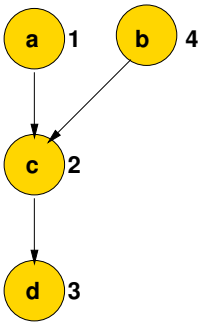
A complexidade de pior caso desse algoritmo é $\mathcal{O}(|V|!|V|), V = max(|V_1|, |V_2|)$. O valor V deve ser o maior entre o número de vértices de G_1 e G_2 pois representará a quantidade máxima de estados visitados pelo algoritmo. Claramente, um algoritmo com essa complexidade fatorial é impraticável para grafos a partir de um determinado número de vértices. No entanto, os autores [22] argumentam que a situação de pior caso acontece se G_1 e G_2 são grafos quase completos. De fato, a manipulação de grafos completos não ocorre no escalonamento de instruções 2D-VLIW uma vez que os grafos de entrada são acíclicos e possuem um limite no número de arestas de entrada. O grafo base também não é um grafo completo, uma vez que cada vértice (as UFs da matriz) possui arestas para apenas dois outros vértices. Nessa situação, a complexidade do algoritmo é limitada por $\mathcal{O}(|V|^2)$.

4.2.1 Heurística 1: Ordenação Topológica

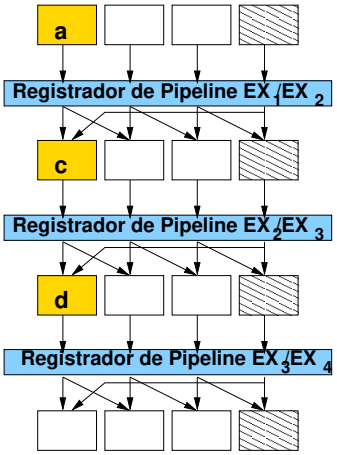
A heurística de ordenação topológica provê uma ordenação dos vértices do grafo G_1 que será considerada durante a execução da função *subg_iso_sched*(). Dependendo da ordenação de G_1, o algoritmo de isomorfismo pode não apresentar um bom desempenho pois terá que executar muitos passos de *backtracking*. A Figura 4.3 mostra um exemplo onde a ordenação topológica pode evitar passos de *backtracking* desnecessários por parte do algoritmo de escalonamento. As Figuras 4.3(a)-4.3(b) mostram um DAG de entrada não ordenado e o escalonamento resultante sem a utilização de *backtrackings*. As Figuras 4.3(c)-4.3(d) mostram o DAG de entrada ordenado a partir dos vértices raízes (vértices com as letras **a** e **b**) para os vértices folhas (vértice com a letra **d**) e o escalonamento resultante sem a realização de qualquer passo de *backtracking*. Os números no lado direito de cada vértice indicam a ordem seguida pelo escalonador. Os retângulos em branco, Figuras 4.3(b) e 4.3(d), representam as UFs disponíveis que podem ser usadas pelo escalonador. Os retângulos hachurados indicam UFs já ocupadas.

A Figura 4.3(b) mostra o estado do algoritmo de isomorfismo de subgrafos depois de escalonar os vértices **a, c** e **d**. Para preservar as dependências entre os vértices, os vértices **d** e **c** devem sofrer o *backtracking* de maneira que o vértice **c** possa ser escalonado novamente sobre uma UF_i que possua uma aresta de entrada de alguma UF_j disponível. Essa UF_j será usada pelo vértice **b**. Observe que esses passos de *backtrackings* não acontecem na Figura 4.3(c) pois o DAG de entrada foi ordenado obedecendo as dependências do grafo. Ao escalonar os DAGs de entrada seguindo as dependências entre os vértices, o resultado final na Figura 4.3(d) é obtido sem a necessidade de realizar qualquer *backtracking*.

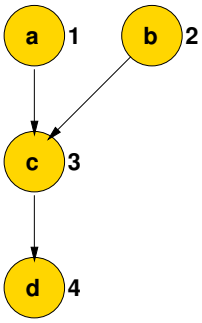
Existem dois algoritmos básicos de ordenação topológica implementados junto ao escalonador de instruções 2D-VLIW. Um algoritmo é baseado na busca em largura no DAG de entrada. O objetivo desse algoritmo é gerar uma ordenação tal que vértices descendentes são mantidos tão próximos quanto possível (na ordenação) de seus ancestrais. O resultado dessa ordenação faz com que um vértice ancestral seja sempre escalonado primeiro e seus descendentes serão escalonados tão logo quanto possível. O segundo algoritmo de ordenação topológica é baseado numa solução do problema de *Bin-*



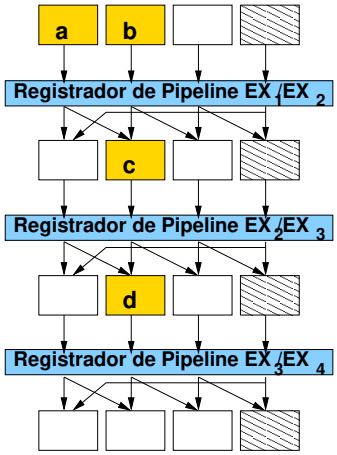
(a) DAG de entrada não ordenado.



(b) Estado do escalonamento sem qualquer *backtracking* para o DAG 4.3(a).



(c) DAG de entrada ordenado.



(d) Estado do escalonamento sem qualquer *backtracking* para o DAG 4.3(c).

Figura 4.3: Exemplo de ordenação topológica.

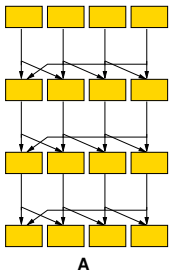
Packing [42] denominado *Best Fit Decreasing*. Nesse algoritmo, os vértices que fazem parte do caminho crítico de um DAG são escalonados primeiro. A motivação para tal algoritmo é evitar situações onde partes de um DAG sejam escalonadas em regiões que poderiam ser ocupadas por vértices do caminho crítico. Ambos os algoritmos de ordenação têm complexidade de pior caso $\mathcal{O}(|V_1| + |E_1|)$.

Embora bastante útil, existem situações onde esta heurística não é suficiente para possibilitar o isomorfismo. Nessas situações, o algoritmo pode lançar mão das heurísticas apresentas nas Subseções 4.2.2 e 4.2.3.

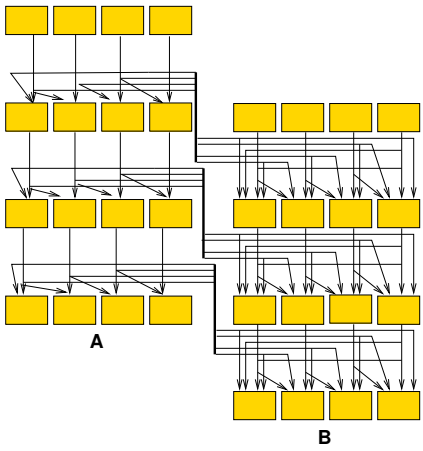
4.2.2 Heurística 2: Redimensionamento do Grafo Base

Esta heurística é executada quando um grafo base G_2 não possui vértices suficientes para comportar um subgrafo isomorfo para G_1. Basicamente, a heurística redimensiona o grafo base, formando assim um novo grafo base maior e com mais possibilidades de comportar um subgrafo isomorfo para G_1. Pode-se entender esta heurística como um desenrolamento do grafo base, uma vez que o aumento do tamanho desse grafo é conseguido através da utilização de mais vértices e arestas ao longo do tempo. O efeito dessa heurística pode ser comparado com uma classe de otimizações de código chamadas de *software pipeline* [4]. A Figura 4.4 exemplifica o desenrolamento de um grafo base, a matriz de UFs da Figura 4.4(a), em um grafo base ainda maior apresentado na Figura 4.4(b). Analisando a matriz de UFs B, pode-se notar que as UFs de uma linha i dessa matriz, $2 \leq i \leq 4$, possuem mais arestas que as UFs de mesma linha na matriz A. Essas arestas indicam que uma UF de uma matriz desenrolada B pode ler valores produzidos por UFs da matriz A e da própria matriz B. A Figura 4.4(c) mostra os detalhes de interconexão de dois vértices da matriz A para dois vértices da matriz B.

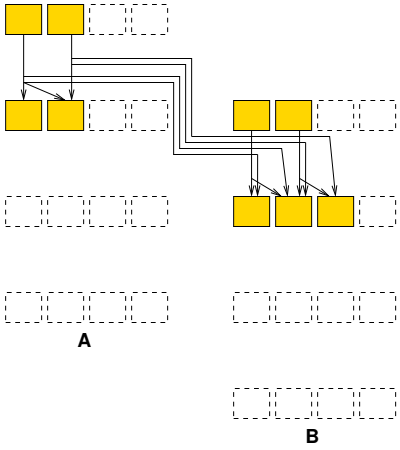
Observe que o DAG na Figura 4.5(a) não pode ser escalonado sobre um grafo base composto por apenas uma matriz de UFs. Isso ocorre porque o vértice **2** possui 3 descendentes e os vértices dos grafo base possuem apenas 2 descendentes. A solução é desenrolar o grafo base até que se consiga vértices com 3 descendentes. Analisando novamente a Figura 4.4(b), é possível notar que um grafo base formado por 2 matrizes de UFs possuem vértices com mais de dois descendentes. Após desenrolar um grafo base,



(a) Grafo base=uma matriz de UFs.

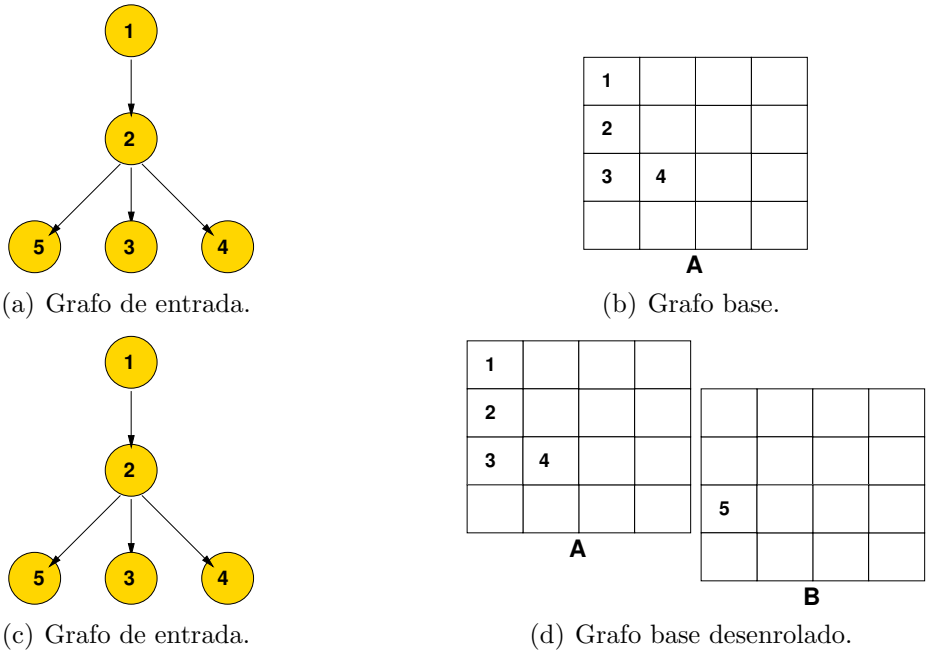


(b) Grafo base desenrolado=duas matrizes de UFs.



(c) Detalhe da interligação entre a matriz A e a matriz B.

Figura 4.4: Exemplo de desenrolamento do grafo base.



(a) Grafo de entrada. (b) Grafo base.

(c) Grafo de entrada. (d) Grafo base desenrolado.

Figura 4.5: Exemplo de redimensionamento do grafo base.

o resultado final do escalonamento na Figura 4.5(d) é obtido. Por simplicidade, o grafo base é representado sem as interligações entre as UFs. Mesmo assim, deve estar claro que essas interligações existem na matriz de UFs e são consideradas durante o escalonamento.

Diferente das demais heurísticas utilizadas durante a etapa de escalonamento, esta heurística modifica o grafo base, ao invés do grafo de entrada. Portanto, a complexidade de pior caso é limitada pelas características do grafo base e não mais pelo grafo de entrada. Nesse caso, a heurística de dimensionamento tem complexidade $\mathcal{O}(|E_2|^2)$. A razão para essa complexidade é que o desenrolamento de um grafo base aumenta o número de vértices de maneira linear, enquanto que a quantidade de arestas é incrementada de maneira quadrática.

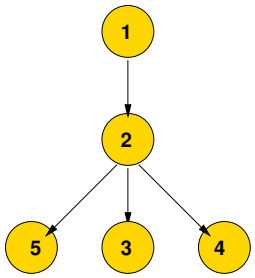
4.2.3 Heurística 3: Vértices Globais no Grafo de Entrada

A heurística de inserção de vértices globais no grafo de entrada funciona como um mecanismo para flexibilizar esse grafo na tentativa de acelerar o escalonamento sobre a matriz de unidades funcionais.

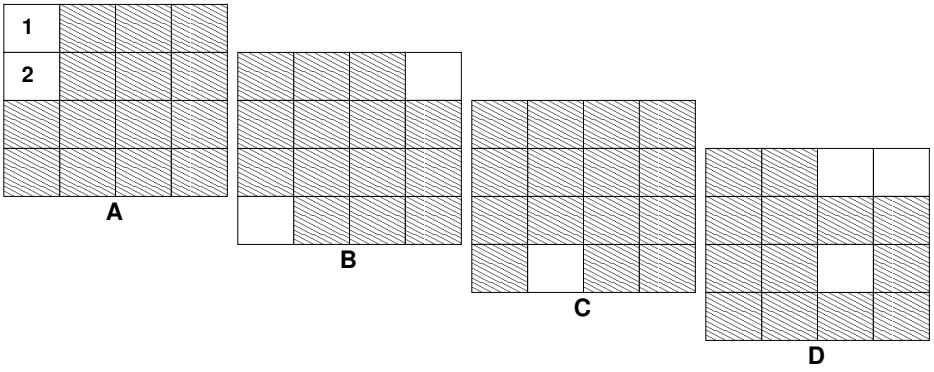
Durante o escalonamento, o escalonador tenta associar os DAGs de entrada com o grafo base. Se o DAG de entrada é complexo ao ponto do escalonador não conseguir realizar o isomorfismo dentro de um intervalo de tempo, o algoritmo de escalonamento pode utilizar esta heurística para flexibilizar o grafo G_1 a fim de encontrar soluções mais rápidas. Basicamente, a heurística converte alguns vértices de G_1 em uma nova classe de vértices, denominados vértices globais, com o intuito de permitir ao escalonador utilizar regiões diferentes do grafo base para realizar o escalonamento. Os vértices escolhidos para serem convertidos são aqueles com o maior número de arestas de saída (maior número de descendentes). A razão para essa escolha está baseada em alguns experimentos empíricos que investigaram a causa de *backtrackings*. Nesses experimentos foi possível notar que os vértices com o maior número de arestas de saída são a causa mais comum para chamadas ao procedimento de *backtracking*. A Figura 4.6 resume como essa heurística funciona.

Na Figura 4.6(a) o vértice **2** possui 3 descendentes e o grafo base, Figura 4.6(b), tem vértices disponíveis suficientes para realizar o isomorfismo com o DAG de entrada. No entanto, o mapeamento de arestas do DAG sobre as arestas do grafo base faz com que o algoritmo de isomorfismo esteja propenso a explorar apenas os vértices conectados através de registradores temporários. A solução encontrada é converter o vértice **2** em um vértice **2g** (vértice global). O termo vértice global indica que o resultado da operação de **2g** será armazenada em um registrador global. Essa mudança possibilita maior flexibilidade para o escalonador, uma vez que valores armazenados em registradores globais podem ser lidos por qualquer vértice do grafo base. Obviamente, desde que os atrasos causados pela escrita e posterior leitura sejam respeitados.

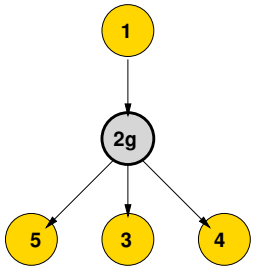
Pode-se observar que o resultado dessa heurística torna o grafo de entrada muito mais flexível com relação ao escalonamento de seus vértices, permitindo-o usar outros vértices do grafo base que não estavam disponíveis sem a inserção do vértice global. A complexidade de pior caso dessa heurística está limitada por $\mathcal{O}(|V_1|)$, já que examina, uma única vez, cada um dos vértices do grafo de entrada à procura daqueles com maior grau de saída.



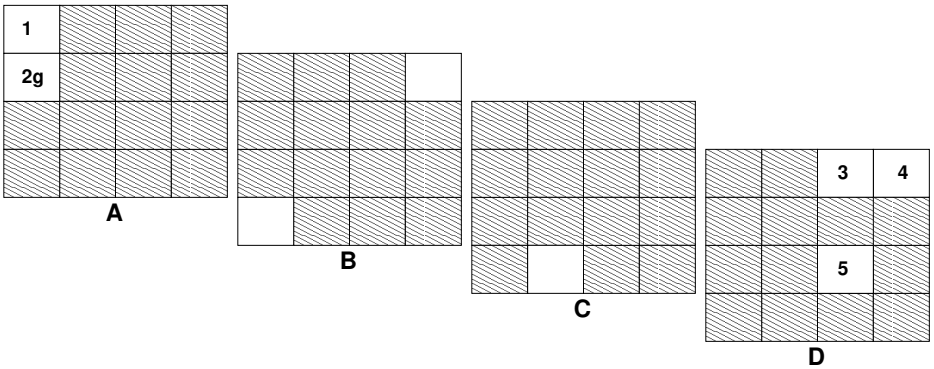
(a) DAG de entrada.



(b) Escalonamento resultante.



(c) DAG de entrada com um vértice global **2g**.



(d) Escalonamento resultante depois da inserção do vértice global.

Figura 4.6: Exemplo da heurística que inser vértice global no DAG.

4.3 Algoritmo de Escalonamento Guloso

O algoritmo de *list scheduling* para 2D-VLIW utiliza técnicas gulosas para realizar o escalonamento considerando cada vértice do grafo de entrada separadamente. Como em qualquer algoritmo de *list scheduling*, este algoritmo escalona cada vértice de acordo com algum critério pré-estabelecido no conjunto de vértices candidatos. Dada uma operação op_i a ser escalonada, o algoritmo tenta escolher uma UF disponível de maneira a agrupar a operação op_i na mesma instrução que seus vértices ancestrais, com o intuito de aumentar o número de operações por instrução. Se não for possível encontrar uma UF que satisfaça essa condição, o algoritmo escolhe a primeira UF disponível, obedecendo a dependência e as restrições de interligação das UFs. Observe que, nesse caso, a operação op_i é armazenada em uma instrução 2D-VLIW diferente de seus antecessores. Da mesma forma que o escalonamento baseado em isomorfismo, a estratégia de escalonamento executa um procedimento de *backtracking* toda vez que não há uma solução de escalonamento possível, de acordo com o estado do escalonamento e as restrições de tempo, para o vértice atual. Mesmo acarretando uma complexidade exponencial, o procedimento de *backtracking* é necessário para garantir que o escalonamento obedeça as dependências entre os vértices do grafo de entrada.

A estratégia de escalonamento guloso baseada em *list scheduling* é resumida através do Algoritmo 3 que apresenta uma estrutura de escalonamento similar àquela apresentada no Algoritmo 1. Exceto pelas linhas 1 e 7, os passos desse algoritmo são os mesmos do Algoritmo 1 discutido na Seção 4.2, com exceção apenas da retirada da heurística de redimensionamento do grafo base, uma vez que essa heurística não é mais necessária na estratégia de *list scheduling*. As mesmas heurísticas para o problema de isomorfismo de subgrafos podem ser aplicadas nos parâmetros de entrada desse novo algoritmo. Além disso, as mesmas restrições de tempo usadas no algoritmo de isomorfismo são também usadas no Algoritmo 3.

O Algoritmo 4 detalha os passos principais da função *greedy_ls_sched*() utilizada no Algoritmo 3.

O laço principal (linhas 3-12) escolhe os vértices (operações) do conjunto $Cand$ de acordo com sua ordenação topológica. A matriz M é usada para armazenar o mapeamento

Algoritmo 3 Estratégia de escalonamento baseado no algoritmo guloso de *List Scheduling*.

ENTRADA: Um grafo de entrada G_1 e um grafo base G_2.
SAÍDA: Conjuntos de instruções 2D-VLIW.
LS_Sched(DAG: G_1, BASE GRAPH: G_2)
1) $G'_2 = greedy\_ls\_sched(G_1, G_2, \&tag)$;
2) **while** $(G'_2 == NULL)$
3) **switch**(tag)
4) case 0 : $topological\_order(G_1)$;
5) case 1 : $DAG\_global\_vertices(G_1)$;
6) $G'_2 = greedy\_ls\_sched(G_1, G_2, \&tag)$;
7) **end while**
8) $create\_2D\text{-}VLIW\_instruction(G'_2)$;

Algoritmo 4 Algoritmo de escalonamento guloso baseado em *List Scheduling*.

ENTRADA: Grafo de entrada G_1 e grafo base G_2.
SAÍDA: Grafo $G2$ representando todas as UFs utilizadas pelos vértices de G_1.
greedy_ls_sched(DAG: G_1, BASE GRAPH: G_2, TAG: &tag)
1) $Cand = G_1.root$;
2) $M[|V_1|][|V_2|]$;
3) **while** $(Cand \neq \emptyset)$
4) $i = position(M, \text{Cand.top})$;
5) $FU = find\_available\_fu(M, i)$;
6) **if**$(FU == NULL)$
7) $backtrack()$;
8) **else**
9) $inst = find\_minimum\_inst(M, FU, i)$;
10) $M[i][FU] = inst$;
11) **end if**
12) **end while**
13) $G'_2 = create\_G'_2(M)$;
14) $tag = evaluate(G'_2, G_1, G_2)$;
15) **return** G'_2;

entre uma operação op_i e uma UF UF_j e, principalmente, saber qual instrução conterá a operação op_i. O número de linhas de M é dado por $|V_1|$ enquanto que o número de colunas é dado por $|V_2|$. A posição $M[i][j]$ armazena o número da instrução onde a operação op_i será armazenada e, além disso, indica que essa operação será executada na UF_j. A função $find\_available\_fu()$ (linha 5) encontra a primeira UF disponível para a operação op_i, de acordo com as restrições de dependência e interligação das UFs. Se nenhuma UF é encontrada, o procedimento *backtrack*() (linha 7) é chamado de maneira que o(s) ancestral(is) direto(s) de op_i ocupe uma nova posição na tentativa de possibilitar um novo escalonamento para essa operação. Quando uma UF disponível é encontrada (linha 8), determina-se uma instrução que possa armazenar op_i através da função $find\_minimum\_inst()$. Após isso, a matriz M é atualizada com a informação da instrução que contém op_i. A função $create\_G'_2()$ recebe a matriz M como entrada e cria um grafo G'_2 com informações sobre as UFs utilizadas no escalonamento.

Ao considerar apenas a informação de cada vértice individualmente, algoritmos baseados em *list scheduling* obtêm um desempenho muito aquém das expectativas em arquiteturas com diversas restrições como 2D-VLIW. Um exemplo disso é que o tratamento local dado por um algoritmo desse tipo gera um número considerável de *backtrackings* que, por sua vez, têm um impacto direto no desempenho do escalonador. Como exemplo, numa situação onde o grafo de entrada é uma árvore, há possivelmente $\mathcal{O}(2^h)$, h = altura da árvore, passos de *backtracking*. Essa complexidade domina o desempenho do algoritmo e acaba por determinar sua complexidade final. Mesmo com a possibilidade de ocorrência de *backtrackings* também no algoritmo de isomorfismo, a possibilidade é menor pois o algoritmo garante, a cada tentativa de escalonamento, que um vértice antecessor seja escalonado numa UF que tenha arestas livres para comportar os vértices sucessores.

O exemplo a seguir mostra como o algoritmo guloso baseado em *list scheduling* escalona os vértices de um grafo de entrada sobre a matriz de unidades funcionais. Considere o DAG apresentado originalmente na Figura 4.3(c) e mostrado novamente na Figura 4.7 como grafo de entrada. A solução apresentada pelo algoritmo pode ser vista na Tabela 4.1.

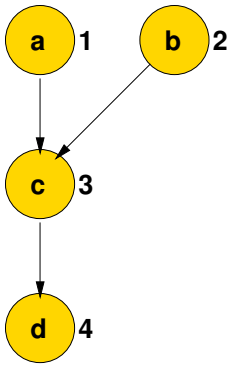


Figura 4.7: Grafo de entrada.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a(0,0)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b(0,0)	-1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c(a,b)	-1	-1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
d(c,0)	-1	-1	0	0	0	-1	0	0	0	1	0	0	0	0	0	0

Tabela 4.1: Resultado do escalonamento baseado no algoritmo guloso de *list scheduling*.

A primeira linha da Tabela 4.1 representa as unidades funcionais presentes na arquitetura. Os números colocados em cada coluna remetem à Figura 3.1. A primeira coluna mostra os vértices que compõem o grafo de entrada, de acordo com a ordem ditada pelo algoritmo de ordenação topológica. Junto a cada vértice, há a informação de dependência, entre parênteses, indicando os ancestrais (x, y) do vértice. Se $x = 0$ ou $y = 0$, significa que o vértice atual não possui ancestral(is). Antes de iniciar o algoritmo, a matriz é preenchida com zeros indicando que nenhuma UF foi escolhida. Conforme especificado no Algoritmo 4, a operação op_i, da linha i, é escalonada em uma UF_j disponível. O escalonamento da operação op_i sobre UF_j deve obedecer às seguintes condições:

- $M[i, j] = 0$.
- caso $op_k \rightarrow op_i$ e op_k é escalonada previamente em $M[k, l]$ e op_i em $M[i, j]$, então deve existir uma aresta de comunicação $e_{lj} = (v_l, v_j) \in E_2$.

Se essas condições são satisfeitas, o algoritmo marca a posição $M[i,j] = 1$ para indicar que a operação op_i utilizará a UF_j na instrução 1. Depois que a UF_j está ocupada, o algoritmo coloca uma marca $-n$, onde n é o número da última instrução 2D-VLIW que utilizou essa UF, nas posições $M[i+1,j],\ M[i+2,j] \ldots, M[|V_1|,j]$. Caso todas as UFs da linha i estejam em uso no momento do escalonamento de op_i, o algoritmo escolhe a primeira posição $M[i,j]$ que satisfaça as dependências de op_i e remarca essa posição, informando a utilização dessa UF_j em uma nova instrução.

É interessante observar que o resultado obtido pelo algoritmo de *list scheduling* para o grafo da Figura 4.7 foi o mesmo resultado do algoritmo de isomorfismo de subgrafos para o mesmo grafo de entrada. No entanto, deve-se atentar que o algoritmo de *list scheduling* realiza um número bem maior de *backtrackings* que o algoritmo baseado em isomorfismo de subgrafos uma vez que não considera a topologia completa do grafo de entrada e sim as dependências individuais dos vértices. Os resultados dos experimentos do Capítulo 5 confirmam as diferenças de desempenho e a qualidade do resultado ao comparar o algoritmo de *list scheduling* com os resultados obtidos pelo algoritmo de isomorfismo de subgrafos.

4.4 Alocação de Registradores

As fases de alocação de registradores e o escalonamento de instruções podem ser realizados conjuntamente na arquitetura 2D-VLIW. Na verdade, a solução encontrada para lidar com o problema de alocação de registradores é usar a mesma técnica baseada em isomorfismo de subgrafos para ambas as tarefas de escalonamento e alocação. Com a adição dessa característica, os grafos de entrada e o grafo base devem sofrer alterações já que precisam considerar a alocação dos resultados/operandos em registradores.

Na solução do problema de alocação de registradores baseada em isomorfismo de subgrafos, o escalonador deve passar a considerar três restrições adicionais durante o escalonamento:

1. Em uma cadeia de dependência $op_j, op_k \rightarrow op_i$, a operação op_i pode ser escalonada em uma UF_m se e somente se op_j e op_k já foram escalonadas e seus resultados estão vivos na entrada da UF_m.

2. Uma operação op_i pode ser escalonada em uma UF_m apenas se, pelo menos, um dos registradores temporários de UF_m estiver livre. Caso op_i armazena seu resultado em um registrador global, o escalonador deve verificar se, na linha de UF_m da matriz de UFs, existe uma aresta livre para o banco de registradores globais.

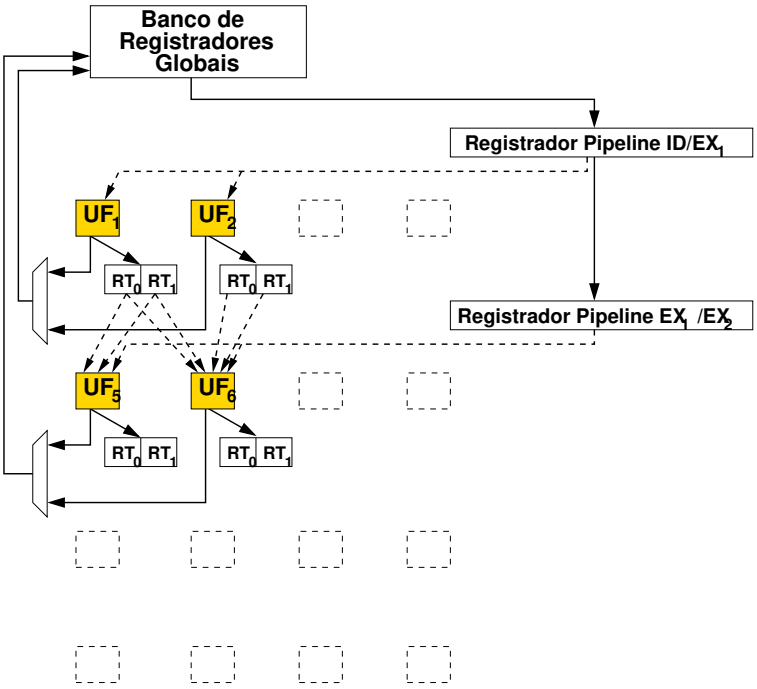
3. Se não existirem registradores temporários livres no momento do escalonamento da operação op_i, um código de *spill*[3] deve ser inserido e escalonado a fim de garantir a semântica correta de execução para op_i e, conseqüentemente, para o programa. Outrossim, a mesma ação deve ocorrer caso o registrador em questão seja global.

As duas primeiras restrições já estão implementadas em uma ferramenta para escalonamento e alocação de registradores considerando a arquitetura 2D-VLIW. Sobre a terceira restrição, o uso e manipulação de código de *spill* durante a alocação de registradores 2D-VLIW assim como melhorias no algoritmo de alocação constituem o foco de investigação de um trabalho de mestrado iniciado no 2° semestre/2006 e que está em desenvolvimento no LSC/IC-UNICAMP pelo estudante Luís Guilherme Pereira. Para os experimentos envolvendo alocação de registradores no Capítulo 5, foi a manipulação de *spill code* fornecida pelo compilador Trimaran.

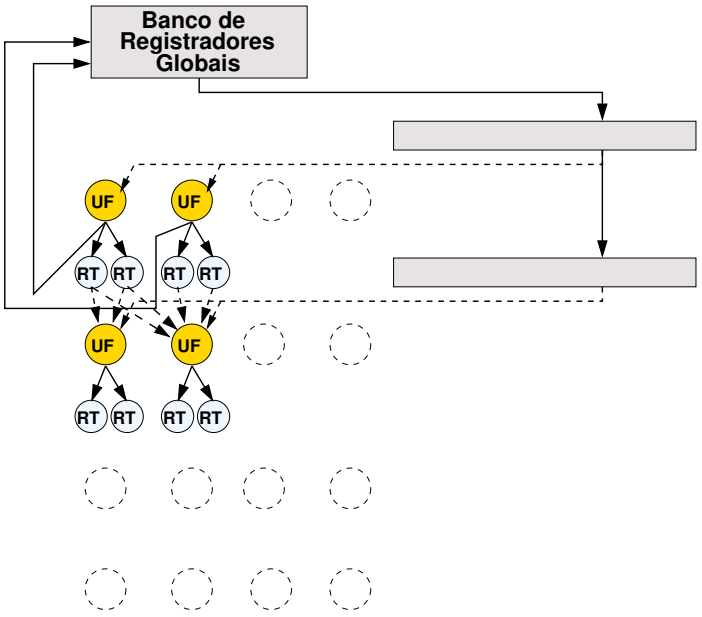
A Figura 4.8 exibe a representação dos bancos de registradores temporários como vértices do grafo base. A Figura 4.8(a) detalha como uma UF é interligada aos seus registradores e na Figura 4.8(b) há a representação desses registradores no grafo base. Por simplicidade, a Figura 4.8 omite os detalhes de interconexão entre os registradores globais e as UFs. Deve estar claro que qualquer registrador global pode ser lido e escrito por qualquer UF desde que o limite do uso de portas de leitura e escrita sejam obedecidos conforme discutido na SubSeção 3.1.2.

Na Figura 4.8, uma aresta contínua $e_{ij} = (UF_i, rt_j)$ representa uma porta de escrita da UF_i para o registrador rt_j. A aresta tracejada representa uma porta de leitura que pode ser usada pela UF. Cada UF escreve seus resultados em apenas um banco de registradores temporários mas pode realizar a leitura a partir de de dois bancos diferentes.

[3]O código de *spill* possibilita que o valor de um registrador seja armazenado na memória, liberando assim um registrador para ser usada pela operação que está executando atualmente.



(a) Interligação entre UFs e os registradores.



(b) Representação dos registradores no grafo base.

Figura 4.8: Registradores globais e temporários como vértices do grafo base.

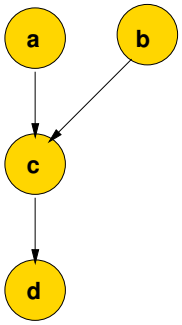
Outro ponto importante a ser observado é que os valores lidos do banco de registradores globais são enviados para as UFs através de registradores de *pipeline*.

A Figura 4.9(b) exibe o mesmo grafo de entrada apresentado na Figura 4.9(a) agora caracterizado com vértices representando os registradores. Nessa figura, pode-se observar a presença de novos vértices entre os pares de vértices $a \rightarrow c$, $b \rightarrow c$ e $c \rightarrow d$. O resultado do escalonamento e da alocação de registradores considerando esse grafo de entrada sobre a matriz de UFs com seus registradores temporários é apresentado na Figura 4.9(c).

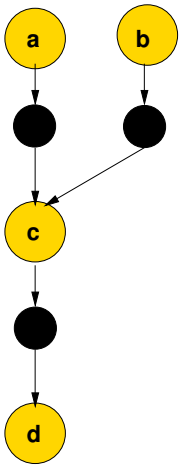
O resultado do escalonamento e alocação de registradores, Figura 4.9(c), utiliza as mesmas UFs da solução obtida na Figura 4.3(d) mas, agora, considera a alocação de registradores. A vantagem adicional nesta abordagem de alocação de registradores é a determinação de quais registradores serão utilizados e, principalmente, garantir que o resultado de uma operação esteja vivo na entrada das UFs das operações dependentes. Uma consideração importante é a respeito da complexidade do algoritmo de isomorfismo com a introdução da alocação de registradores. Na realidade, essa nova atividade não aumenta a complexidade do problema já que os vértices representando registradores são definidos como uma classe especial de vértices, diferente da classe dos vértices que são operações e UFs. O algoritmo de escalonamento considera apenas vértices de mesma classe como potenciais para o isomorfismo e, com isso, não há confusão entre vértices representando operações e vértices representando registradores. De fato, uma conseqüência gerada com a introdução desses novos vértices é o aumento do número de vértices e arestas dos dois grafos o que, de certa forma, pode impactar o tempo de processamento do algoritmo de escalonamento e alocação. No entanto, as heurísticas apresentadas nas SubSeções 4.2.1-4.2.3 procuram justamente flexibilizar os grafos de entrada e o grafo base na tentativa de facilitar o isomorfismo e, como conseqüência, melhorar o desempenho do escalonador.

4.5 Codificação de Instruções

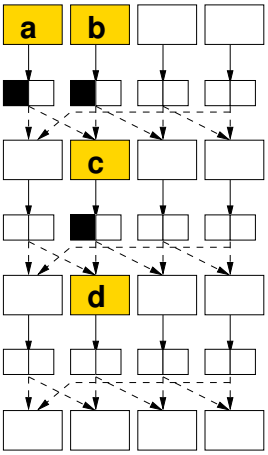
De maneira geral, programas que utilizam instruções longas ocupam uma quantidade total de memória maior do que aqueles que adotam instruções de tamanhos menores. Na tentativa de reduzir esse uso de memória e, ao mesmo tempo, maximizar o desempenho da atividade de busca de instruções, algumas arquiteturas adotam técnicas de compressão



(a) Representação original do grafo de entrada.



(b) Representação do grafo de entrada com vértices indicando o uso de registradores.



(c) Resultado do escalonamento e da alocação de registradores.

Figura 4.9: Grafo de entrada representando as operações e registradores e o resultado final do escalonamento com alocação de registradores.

do tamanho das instruções. Originalmente, a arquitetura 2D-VLIW armazena e busca na memória instruções longas formadas por operações que serão executadas sobre a matriz de UFs. Sendo assim, a abordagem adotada por 2D-VLIW para reduzir a área de memória ocupada pelas instruções é baseada em uma estratégia de codificação de instruções. Essa técnica visa à redução do tamanho da instrução que é armazenada na memória ao mesmo tempo em que mantém a via de dados tão simples quanto possível.

A estratégia de codificação é formada por um algoritmo que fatora os padrões das instruções e obtém uma instrução codificada e um padrão que, junto com essa codificação, permite recriar a instrução completa.

Definição 8 *Uma instrução codificada I contém operandos não redundantes de uma instrução original e não codificada S e um campo de endereço para o padrão que representa a instrução original.*

Definição 9 *Um padrão P de uma instrução não codificada S é formado pelo conjunto das operações $op_1, op_2, \ldots, op_n \in S$,$n$ =quantidade de UFs da matriz, juntamente com ponteiros para os operandos presentes na instrução codificada I.*

Com a introdução das instruções codificadas e padrões de intruções, o estágio de decodificação passa a ter uma atividade adicional. Antes de apresentar essa nova atividade, deve-se ter em mente que as instruções codificadas são armazenadas na memória e na cache de instruções da mesma forma como acontecia com as instruções não codificadas. Os padrões de instruções são também armazenados na memória. A diferença é que os padrões utilizam uma cache específica denominada cache de padrões. Diante desse contexto, o processo de busca e decodificação passa a ser:

- No estágio de busca de instrução, uma instrução codificada é buscada na memória utilizando a hierarquia: cache de instruções no nível 1, cache de nível 2, memória. Obviamente, a cache de nível 2 e a memória só serão utilizadas, respectivamente, caso a instrução procurada não seja encontrada na cache de nível 1 e na cache de nível 2.

- No estágio de decodificação, enquanto os registradores globais são lidos, busca-se um padrão de acordo com o campo de endereço presente na instrução codificada. A busca do padrão segue a hierarquia: cache de padrões, cache de nível 2, memória.
- De posse do padrão e da instrução codificada, a decodificação da instrução é realizada e, a partir de então, tem-se uma instrução completa cujas operações serão executadas na matriz de UFs.

De maneira parecida com as técnicas tradicionais de compressão de código baseadas em fatoração de operandos [5, 30, 32], a técnica de codificação de instruções 2D-VLIW extrai padrões a partir de operandos redundantes na instrução não codificada. Essa estratégia gera uma considerável redução no tamanho da instrução codificada, uma vez que dados redundantes não estarão mais presentes em seu conteúdo. Essa técnica está pautada na sobrejeção intrínseca entre instruções e padrões. Pesquisas anteriores [18, 77] focando a busca de padrões de instruções, descobriram que a função que mapeia o conjunto de instruções de um programa CI para o conjunto de padrões dessas instruções CP obedece ao comportamento de uma função sobrejetora. Em outras palavras, dado um conjunto de instruções codificadas $CI = \bigcup_{j=1}^{n} I_j$, um conjunto de padrões $CP = \bigcup_{i=1}^{m} P_i$, existe um mapeamento f tal que

$$\begin{aligned} P_l = f(I_i) = f(I_j) = f(I_k), \quad \text{onde} \\ i \neq j \neq k, \\ \{I_i,\ I_j,\ I_k\} \subset CI, \\ P_l \in CP \end{aligned} \tag{4.1}$$

Observe que a função f na Equação 4.1 é sobrejetora pois mapeia instruções diferentes sobre um mesmo padrão P_l.

Como já apresentado no Capítulo 3, existem algumas restrições na arquitetura 2D-VLIW envolvendo o número de portas disponíveis nos registradores globais. Essas restrições devem ser levadas em consideração durante o processo de codificação de instruções já que também representam uma limitação sobre os tipos de operações que podem fazer parte

de uma instrução. Uma conseqüência da aplicação dessas restrições arquiteturais durante a codificação de uma instrução é que o número de instruções codificadas é geralmente maior que o número de instruções não codificadas. Assim, a existência da sobrejeção entre o conjunto de instruções e o conjunto de padrões é uma condição necessária para que essa estratégia possa reduzir a quantidade de memória utilizada por um programa. Os resultados apresentados no Capítulo 5 comprovam a existência dessa sobrejeção e, mais importante, mostram a redução no tamanho do programa acarretada por essa técnica.

Semelhante à analogia proposta no Capítulo 3, envolvendo registradores e chamadas de funções, pode-se entender a técnica de codificação como uma chamada de função. Os parâmetros de entrada da função são os operandos que estão na instrução codificada. A função é o próprio padrão da instrução e o conteúdo da função são as operações que são armazenadas pelo padrão. Quando uma nova instrução é buscada na memória, isso significa que os parâmetros estão sendo repassados para a função. A chamada à função é equivalente à busca do padrão na cache. A execução da função corresponde à execução das operações individuais que compõem o padrão.

Um ponto importante aqui é entender como funciona o algoritmo para criar as instruções codificadas. Em linhas gerais, o algoritmo para codificação analiza todas as instruções não codificadas geradas pelo escalonador. Para cada instrução não codificada S, a primeira tarefa do algoritmo é criar uma instrução codificada vazia I e um padrão vazio P. Considere uma instrução S_j onde, para cada operação $op_k \in S_j$, $1 \leq k \leq 16$ o algoritmo armazena o código da operação op_k em um campo de P_j e verifica se os operandos de op_k estão na instrução codificada I_j. Em caso positivo, o algoritmo adiciona um ponteiro para esses operandos em P_j. Considerando que operandos dos registradores globais são lidos no estágio de decodificação (em paralelo com a decodificação da instrução), esses operandos são armazenados nas posições iniciais de I_j. A quantidade de posições reservadas para os operandos de registradores globais depende do número de portas de leitura disponível no RG. Obviamente, devido às restrições arquiteturais, algumas operações que pertenciam à instrução S_j podem não estar em I_j. Nesse caso, as operações que não puderam fazer parte de I_j devem compor uma nova instrução I_{j+1}. Ao terminar a análise sobre todas as operações de S_j, obtem-se então uma nova instrução codificada I_j e um padrão P_j. No

entanto, antes de considerar efetivamente a existência desse padrão P_j, faz-se uma busca no conjunto de padrões CP já criado com o intuito de averiguar se $P_j = P_i, \forall P_i \in CP$. Em caso positivo, a instrução I_j deve ser atualizada com a informação do endereço do padrão P_i enquanto que o padrão P_j será descartado. Em caso negativo, P_j é adicionado ao conjunto CP e tem seu endereço inserido em I_j.

O Algoritmo 5 apresenta os passos principais do algoritmo de codificação. Alguns procedimentos do algoritmo, por exemplo, detecção e descarte de um padrão existente, manipulação de registradores de destino, entre outros, foram omitidos em favor da simplicidade. A entrada para o algoritmo é o conjunto de instruções não codificadas CS, obtido após as fases de escalonamento de instrução e alocação de registradores.

Para cada instrução do conjunto CS (linha 1), o algoritmo analisa o conteúdo da instrução (linha 4) verificando se os operandos *op.opnd*1 e *op.opnd*2 da operação atual *op* estão em alguma posição da instrução codificada I (linhas 5 e 16, respectivamente). Se estiverem, ponteiros devem ser acrescentados no padrão P a fim de indicar quais são os operandos corretos da operação *op* (linhas 14, 15 e 25). Se, devido à alguma restrição da arquitetura, operações não podem ser codificadas na instrução atual I, essas operações serão codificadas em uma nova instrução (linhas 7-9, 18-20). O algoritmo possui dois laços importantes: o primeiro é repetido até que todas as instruções de CS sejam analisadas; o segundo verifica cada operação presente na instrução S selecionada no laço anterior. Embora não apresentando, é importante entender que antes da execução da inserção do padrão ao conjunto de padrões (linha 28), deve-se verificar se o referido padrão já existe nesse conjunto. Como o número de operações de uma instrução é constante, a complexidade do algoritmo é limitada pelo tamanho do conjunto CS e pela complexidade da consulta ao conjunto de padrões. Em uma situação de pior caso $|CP| = |CS|$ e, portanto, $\mathcal{O}(|CS|^2)$.

A Figura 4.10 exemplifica a utilização do algoritmo de codificação sobre uma instrução 2D-VLIW sem codificação. Por simplicidade, considere uma instrução não codificada formada apenas por 4 operações, isto é, uma arquitetura 2D-VLIW com 2×2 UFs.

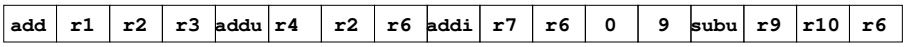
Algoritmo 5 Algoritmo de codificação.

ENTRADA: Conjunto de instruções não codificadas CS.
SAÍDA: Conjunto de instruções codificadas CI e padrões CP.
Encoding(SET_INST: CS**)**

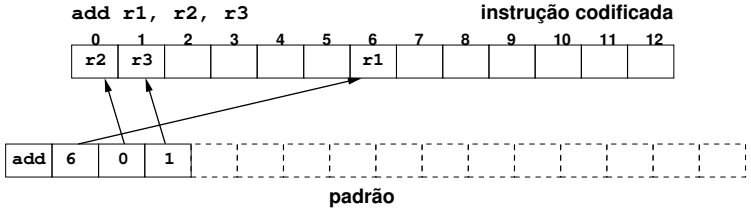
```
1) for each S ∈ CS
2)    create new I;
3)    create new P;
4)    for each op ∈ S
5)       if op.opnd1 ∉ I
6)          if free_space(I) < 1
7)             CI = CI ∪ I;
8)             CP = CP ∪ P;
9)             create new I;
10)            create new P;
11)         end if
12)         I.add(op.opnd1);
13)      end if
14)      P.add_op(op.opcode);
15)      P.add_opnd = location of op.opnd1 in I;
16)      if op.opnd2 ∉ I
17)         if free_space(I < 1)
18)            CI = CI ∪ I;
19)            CP = CP ∪ P;
20)            create new I;
21)            create new P;
22)         end if
23)         I.add(op.opnd2);
24)      end if
25)      P.add_opnd = location of op.opnd2 in I;
26)   end for
27)   CI = CI ∪ I;
28)   CP = CP ∪ P;
29) end for
```

```
add r1, r2, r3
addu r4, r2, r6
addi r7, r6, 9
subu r9, r10, r6
```

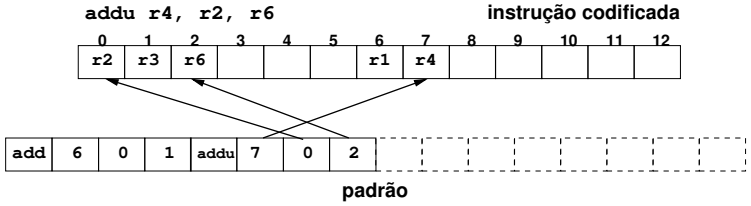
(a) Código que será escalonado.



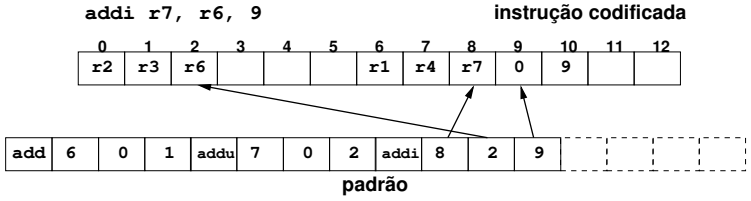
(b) Instrução não codificada S.



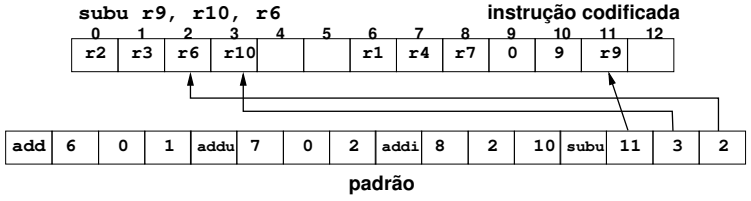
(c) Instrução codificada I e o padrão P após codificar a operação add.



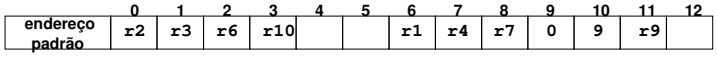
(d) Instrução codificada I e o padrão P após codificar a operação addu.



(e) Instrução codificada I e o padrão P após codificar a operação addi.



(f) Instrução codificada I e o padrão P após codificar a operação subu.



(g) Instrução codificada I após todos os passos do algoritmo.

Figura 4.10: Exemplo de execução da técnica de codificação.

A Figura 4.10 mostra os passos para codificar a instrução definida em 4.10(b) usando a estratégia de codificação 2D-VLIW. As Figuras 4.10(c)-4.10(f) mostram o estado atual da instrução codificada e o conteúdo do padrão após cada passo do algoritmo. Em cada etapa, o algoritmo atualiza e instrução codificada e seu respectivo padrão. As setas indicam os operandos e o resultado utilizados por cada operação inserida no padrão. Considere, por exemplo, a operação `add r1, r2, r3` no passo 4.10(c). Primeiro, o código da operação `add` é inserido no padrão. Depois disso, o primeiro registrador (`r1`), que é um registrador de saída, é armazenado no campo 6 da instrução codificada (os campos 0 até 5 são reservados para leitura dos operandos do banco de registradores globais). Um ponteiro para o campo 6 é armazenado em uma linha da cache de padrões. A codificação dos dois registradores de leitura (`r2` e `r3`) segue o mesmo procedimento mas usa os campos reservados para leitura dos registradores globais. Observe que a segunda operação `addu r4, r2, r6`, Figura 4.10(d), também utiliza o registrador `r2`. Nesse caso, o campo 0 será reusado. No último passo, Figura 4.10(f), tem-se uma instrução codificada e um padrão. A instrução codificada é armazenada na cache de instruções enquanto que o padrão será armazenado na cache de padrões. Na Figura 4.10(g), a informação sobre o endereço do padrão é adicionada na instrução codificada.

É importante ter em mente que a instrução original (não codificada), Figura 4.10(b), e o padrão armazenado na cache de padrões, Figura 4.10(f), são bem diferentes embora tenham quase o mesmo número de campos. A diferença marcante está na representação dos dados pois os campos de um padrão armazenam ponteiros para campos da instrução codificada, enquanto os campos de uma instrução não codificada armazenam índices dos registradores e valores imediatos.

O processo de decodificação de uma instrução 2D-VLIW é simples: depois que o padrão é buscado na cache de padrões, o hardware de decodificação utiliza os ponteiros que estão armazenados dentro do padrão, juntamente com a instrução codificada trazida da memória, para reconstruir a instrução 2D-VLIW não codificada. Durante a decodificação, a instrução codificada funciona como um dicionário de operandos para os ponteiros do padrão. Essa característica, atrelada à capacidade de decodificação enquanto a leitura dos registradores globais é processada, possibilita que essa técnica de codificação tenha

uma grande diferença para técnicas tradicionais de compressão que armazenam padrões ou dados para decodificação em tabelas. Em geral, técnicas de compressão de código acrescentam um estágio específico para decodificação da palavra junto à via de dados, além de utilizarem uma ou mais tabelas (dicionários) para reconstruir a palavra completa.

Considerando uma arquitetura 2D-VLIW com 4×4 UFs, uma instrução 2D-VLIW sem codificação possui 512 bits (considerando um conjunto de instruções cujas operações possuem 32 bits). A instrução codificada definida para essa arquitetura possui 128 bits, um decremento da ordem de $4\times$ sobre o tamanho original, enquanto que um padrão possui 496 bits. Como já mencionado no Capítulo 3, as pesquisas envolvendo técnicas de codificação para instruções 2D-VLIW têm suas origens a partir de estudos iniciais realizados no escopo desta Tese. Atualmente, as atividades relacionadas com codificação de instruções, padrões e cache de padrões na arquitetura 2D-VLIW são parte de um trabalho de mestrado em desenvolvimento no LSC/IC-UNICAMP.

4.6 Considerações Finais

Este capítulo apresentou as características principais do processo de geração de código para a arquitetura 2D-VLIW. Dentre as diversas etapas que fazem parte do processo de geração de código para uma arquitetura de processador, as técnicas utilizadas para escalonar e codificar instruções e alocar registradores assumem papel de destaque em virtude de seus impactos sobre o projeto da arquitetura. Deve-se, inclusive, destacar que a investigação em torno de ferramentas, técnicas e algoritmos para geração de código nessa arquitetura são também objetos de estudos de dois trabalhos em nível de mestrado atualmente em desenvolvimento no LSC/IC-UNICAMP.

Capítulo 5

Experimentos e Resultados

A tarefa de avaliar e comparar experimentalmente projetos em torno de arquiteturas de processadores tem se mostrado bastante árdua. Muitas vezes, esses projetos não disponibilizam recursos e informações suficientes para que seja possível realizar tal avaliação. Mesmo as iniciativas que procuram implementar ferramentas para simular o comportamento dos processadores não possuem um considerável conjunto de casos de entrada, de forma que é difícil avaliá-las e compará-las com outras propostas.

Este capítulo cobre os experimentos realizados a fim de validar empiricamente as considerações que compõem esta Tese e que foram discutidas nos capítulos anteriores. Ao longo deste capítulo, procura-se apresentar e justificar os experimentos escolhidos assim como a infraestrutura utilizada na realização desses experimentos. Por não possuir ainda uma implementação em hardware, os experimentos envolvendo o processador 2D-VLIW são baseados na simulação do modelo de execução na tentativa de determinar o desempenho do processador. Além disso, também há experimentos sobre as atividades de escalonamento de instruções, alocação de registradores e codificação de instruções que, por afetarem diretamente a qualidade do código gerado, possuem também influência no desempenho final da arquitetura.

5.1 Experimentos e Medidas de Desempenho

Os experimentos envolvendo a arquitetura 2D-VLIW cobrem três características básicas: o modelo de execução, o escalonamento de instruções juntamente com a alocação

de registradores e, por fim, o esquema de codificação de instruções. Os experimentos apresentados neste capítulo foram realizados com base na simulação de uma especificação da arquitetura e na execução de algoritmos que geram código tendo como alvo as características arquiteturais definidas no Capítulo 3.

Em todos os experimentos, procurou-se utilizar o mesmo conjunto de programas divididos entre três benchmarks diferentes, com o intuito de facilitar análises comparativas e, ao mesmo tempo, utilizar um conjunto de programas que seja representativo sobre a característica avaliada. Esses programas são provenientes dos pacotes de benchmarks SPEC2000 [41] (programas que lidam com números inteiros e de ponto-flutuante) e MEDIABench [51]. A escolha dos programas teve como base o trabalho de Joshi e outros [43] que consistiu na análise dos diversos programas que compõem os pacotes SPEC2000 e MEDIABench, onde pôde-se concluir que apenas um subconjunto de todos esses programas é suficiente para mensurar o desempenho de um processador. Os experimentos aqui apresentados cobrem aproximadamente 75% do conjunto de programas do pacote SPEC definidos pelos seus autores. A não-cobertura total de todos os programas foi devido à ausência do programa 176.gcc. Ao compilar esse programa com Trimaran, não foi possível confirmar a correção da execução do programa e, por esse motivo, o 176.gcc foi retirado do conjunto de programas dos experimentos. Os programas escolhidos e utilizados nos experimentos discutidos neste capítulo são apresentados na Tabela 5.1.

Programas	**Benchmark**	**Descrição**
168.wupwise	SPECfp	Cálculos de cromodinâmica quântica
175.vpr	SPECint	Análise e síntese de circuitos digitais
179.art	SPECfp	Redes neurais e reconhecimento de imagens
181.mcf	SPECint	Otimização de veículos em transporte público
183.equake	SPECfp	Simulação da propagação de ondas sísmicas
197.parser	SPECint	*Parser* de sentenças em inglês
256.bzip2	SPECint	Compressão e descompressão de arquivos
epic	MEDIABench	Compressão de imagens por meio de *wavelets*
g721.decode & g721.encode	MEDIABench	Codificação e decodificação de sinais de voz

Tabela 5.1: Programas utilizados nos experimentos.

Todos os programas foram compilados com o compilador Trimaran (versão 3.7) e as opções de formação de hiperblocos e desenrolamento de laços (em até 32 vezes) habilitadas. Nos experimentos apresentados na Seção 5.3, o modelo de execução 2D-VLIW foi especificado através da linguagem de descrição de arquiteturas denominada HMDES [37]. Nesse conjunto de experimentos, o modelo de execução 2D-VLIW foi comparado considerando duas medidas de desempenho: o *speedup* e a escalabilidade. A primeira medida considera a proporção em que um sistema é melhor que outro segundo algum critério previamente definido. No experimento em questão, o *speedup* mede quanto o modelo de execução 2D-VLIW possui melhor desempenho que o modelo EPIC, considerando o tempo de execução de EPIC, T_B, em ciclos de relógio, sobre o tempo de execução de 2D-VLIW, T_A, conforme a Equação 5.1.

$$Sp = \frac{T_B}{T_A} \tag{5.1}$$

A escalabilidade informa se há ocorrência de *speedup* de uma configuração A_1 de um processador, sobre uma configuração B_1 do mesmo processador. O intuito é verificar se à medida que se aumenta o número de unidades funcionais, o desempenho continua aumentando.

As latências consideradas para as operações existentes nos programas são apresentadas na Tabela 5.2. A informação sobre a latência representa a quantidade de ciclos de relógio necessários para computar a operação no estágio de execução.

Os experimentos descritos na Seção 5.4 utilizam duas medidas que avaliam o preenchimento de uma instrução 2D-VLIW, Operações Por Instrução - OPI, e o preenchimento da matriz de unidades funcionais, Operações Por Ciclo - OPC. Essas medidas permitem indicar se uma estratégia de escalonamento e/ou alocação de registradores está utilizando os recursos de hardware de maneira eficiente. Obviamente, as melhores soluções são aquelas que provêem os níveis mais altos de OPIs e OPCs.

Por fim, os experimentos descritos na Seção 5.5 procuram averiguar como a estratégia de codificação afeta o tamanho do código dos programas quando comparada a uma estratégia sem codificação. Nesse caso, duas medidas merecem destaque: o Reuso e o

Operações	**Latência (em ciclos)**
Operações com inteiros (excluindo mult. e divisão)	1
Multiplicação	3
Divisão	8
Operações em ponto-flutuante	3
Busca na cache de nível 1	2
Busca na cache de nível 2	7
Busca na memória	35
Escrita na memória	1
Operações de desvio	1

Tabela 5.2: Latências das operações consideradas.

Fator de Redução. A primeira medida indica a quantidade média de instruções codificadas que utilizam um mesmo padrão. Em outras palavras, essa é a medida que verifica se há sobrejeção entre o conjunto de instruções e o conjunto de padrões. A segunda medida informa quanto foi a redução da área de código de um programa a partir do uso da técnica de codificação. É importante observar que, quanto maior o reuso e o fator de redução, mais eficiente é a técnica de codificação.

5.2 Infraestrutura para Geração dos Experimentos

A fim de determinar o desempenho de algumas características que compõem a arquitetura 2D-VLIW, deve-se lançar mão de ferramentas que possibilitam avaliar essas características através de experimentos. Como não há uma implementação em hardware do processador 2D-VLIW, toda a infraestrutura para execução de experimentos é composta por ferramentas de software. Algumas dessas ferramentas são partes do compilador Trimaran, enquanto outras foram adicionadas ao fluxo de execução desse compilador. A Figura 5.1 mostra a organização do compilador Trimaran por meio do fluxo de execução de seus principais elementos. A Figura 5.1(a) mostra o fluxo de execução normal do compilador Trimaran, onde é possível observar quatro elementos distintos:

- Impact: é o elemento responsável pela tradução do programa em uma linguagem fonte para uma linguagem intermediária com otimizações independentes de máquina;

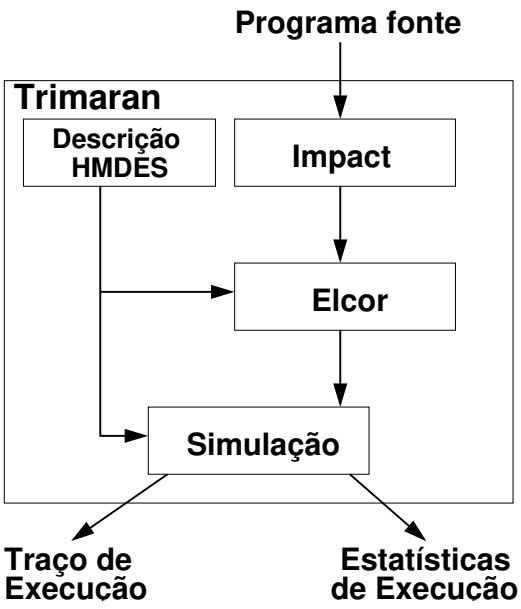
- Elcor: é o elemento responsável pelas otimizações dependentes de máquina do compilador e onde as fases de escalonamento e alocação de registradores são realizadas;

- Descrição HMDES: é a descrição do processador que deverá ser considerada pelas otimizações dependentes de máquina e na etapa de simulação;

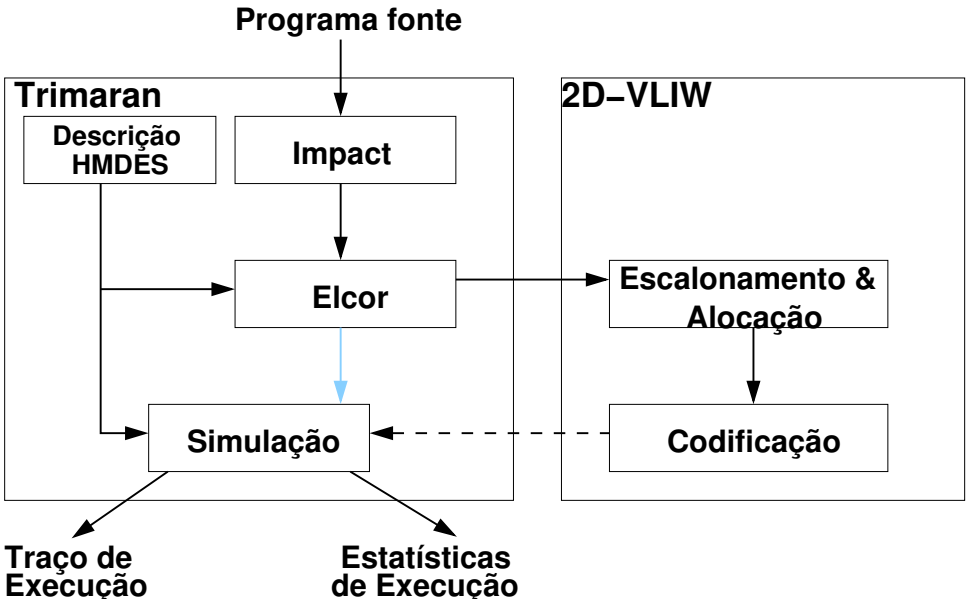
- Simulação: esse elemento recebe uma descrição de máquina e o código com as otimizações dependentes de máquina e realiza a simulação do código. A fase de simulação permite obter um traço das operações do programa que foram executadas e estatísticas gerais sobre a execução. É possível obter também um programa executável independente.

A Figura 5.1(a) representa o fluxo de execução das ferramentas nos experimentos envolvendo o desempenho do modelo de execução (Seção 5.3). Para esse experimento, o algoritmo de escalonamento e o de alocação de registradores utilizados foram aqueles já implementados na infraestrutura Trimaran. A Figura 5.1(b), por outro lado, mostra o fluxo de execução para os experimentos das Seções 5.4 e 5.5. As setas com cores escuras indicam o fluxo de execução seguido nesses experimentos, a seta clara indica o fluxo de execução normal do compilador Trimaran mas que não é utilizado nesses experimentos, a seta tracejada indica um fluxo de execução ainda não implementado na arquitetura. A ausência de um fluxo automatizado entre as ferramentas de escalonamento & alocação e codificação, desenvolvidas para a arquitetura 2D-VLIW e o compilador Trimaran, não prejudicou a obtenção dos resultados nos experimentos das Seções 5.4 e 5.5 pois, esses experimentos avaliam o resultado das ferramentas e não o desempenho final da aplicação. Mesmo para o experimento que utiliza informação dinâmica (Tabela 5.4), o resultado é baseado no traço de execução obtido previamente com o fluxo normal de ferramentas do compilador Trimaran. Deve-se ressaltar que a integração entre essas ferramentas já foi iniciada e está em fase de desenvolvimento.

A linguagem HMDES possibilita descrever diversas características arquiteturais em um estilo hierárquico, de forma que estas podem ser adicionadas a uma especificação em função de outras definidas previamente. Uma das características que chama a atenção nessa linguagem e, conseqüentemente, motivou sua adoção para descrever o modelo de



(a) Fluxo de execução do compilador Trimaran.



(b) Fluxo de execução do compilador Trimaran com as ferramentas desenvolvidas na arquitetura 2D-VLIW.

Figura 5.1: Organização e fluxo de execução do compilador Trimaran com e sem as ferramentas específicas para a arquitetura 2D-VLIW.

execução 2D-VLIW, reside na capacidade de especificar unidades funcionais, a interligação das operações do conjunto de instruções HPL-PD com UFs e com bancos de registradores da arquitetura e a definição do estágios de execução para a(s) unidade(s) funcional(is). Essas características da linguagem vão ao encontro de algumas particularidades do modelo de execução 2D-VLIW já que, como apresentado no Capítulo 3, esse modelo adota unidades funcionais, interligadas através de registradores, que são ativadas em diferentes estágios de execução. Obviamente, os recursos da linguagem não capturam diretamente todas as características do modelo de execução. Por exemplo, não é possível modelar canais compartilhados assim como também não é possível modelar a utilização de mais de um banco de registradores por operação. Essas restrições impedem a modelagem de determinadas características como o acesso compartilhado das unidades funcionais, de uma linha da matriz, para o banco de registradores e a utilização de registradores temporários e globais por parte de uma operação. De fato, a capacidade de utilizar barramentos (canais compartilhados) por parte das unidades funcionais que estão em uma linha da matriz não foi coberta pela modelagem utilizando HMDES. De qualquer forma, as ferramentas de compilação tomam conhecimento dessa restrição da linguagem HMDES e não geram código considerando essa característica. A restrição envolvendo a utilização de mais de um banco de registradores foi resolvida através da criação de bancos globais distribuídos do mesmo tamanho dos bancos de registradores temporários e da interligação desses bancos às unidades funcionais.

O modelo de execução 2D-VLIW especificado na linguagem HMDES é utilizado no experimento envolvendo o desempenho dessa especificação. Essa modelagem não foi, no entanto, utilizada nos experimentos sobre o escalonamento de instruções assim como naqueles sobre codificação de instruções, uma vez que as ferramentas adotadas nesses experimentos foram construídas levando em conta todas as características da arquitetura, inclusive aquelas que não puderam ser especificadas pela linguagem HMDES.

5.3 Desempenho do Modelo de Execução

O primeiro experimento compara o *speedup* obtido pelo modelo de execução 2D-VLIW com o modelo de execução utilizado pelo processador HPL-PD apresentado na

SubSeção 2.2.6. Os dois modelos de execução foram especificados utilizando a linguagem HMDES. O número de registradores, unidades funcionais, conjunto de instruções e latência das operações foram os mesmos para ambos os modelos. A correção desse experimento é garantida pela comparação entre traços de execução executados no processador hospedeiro[1] sem a utilização do modelo avaliado (2D-VLIW ou HPL-PD) e com a utilização do modelo. Os programas aqui apresentados foram aqueles cujos traços são iguais tanto antes quanto depois da inclusão dos modelos. Esse mecanismo de correção procura garantir que o código simulado por um modelo 2D-VLIW ou HPL-PD seja equivalente ao executado por um processador "real".

A Figura 5.2 mostra o *speedup* (em ciclos de processamento) obtido pelo modelo 2D-VLIW com diversas configurações da matriz de unidades funcionais sob as mesmas configurações do modelo EPIC adotado pelo processador HPL-PD. Por exemplo, as colunas correspondentes à legenda **2D-VLIW 3×3** indicam o *speedup* de uma configuração 2D-VLIW com 3×3 unidades funcionais sobre uma configuração HPL-PD com 9 unidades funcionais. Observe que o *speedup* foi alcançado em todas as configurações, com ganhos na faixa de 5% até 63%.

Note que o modelo 2D-VLIW obtém melhores resultados que o modelo HPL-PD em todos os programas. Depois de uma análise do código gerado para ambas as arquiteturas, pode-se observar que a razão por trás do *speedup* 2D-VLIW sobre HPL-PD foi devido à organização das unidades funcionais. A adição de registradores entre as unidades funcionais tornou possível manipular dependências escrita-após-leitura[2] através da matriz sem incorrer em atrasos extras por causa da leitura de um banco com muitos registradores globais. Deve-se destacar que ganhos maiores que esses seriam esperados caso a especificação pudesse capturar todas as características de 2D-VLIW e o simulador levasse em consideração as latências decorrentes do número de portas do banco de registradores globais.

Outro experimento relacionado ao desempenho refere-se à escalabilidade obtida ao utilizar configurações diferentes de unidades funcionais tanto na arquitetura 2D-VLIW quanto em HPL-PD. A Figura 5.3 exibe os resultados com o experimento de escalabilidade.

[1] Processador que executa o compilador Trimaran.
[2] Do inglês: *Write-After-Read* (WAR).

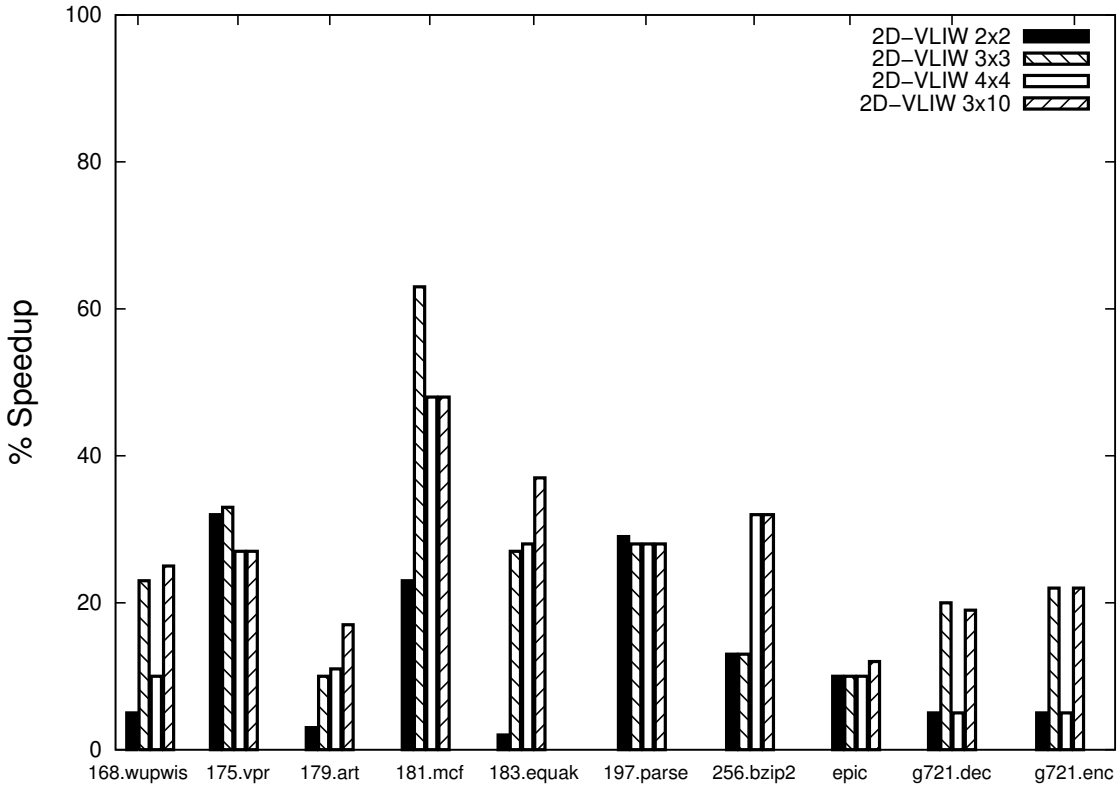


Figura 5.2: *Speedup* do modelo de execução 2D-VLIW sobre HPL-PD.

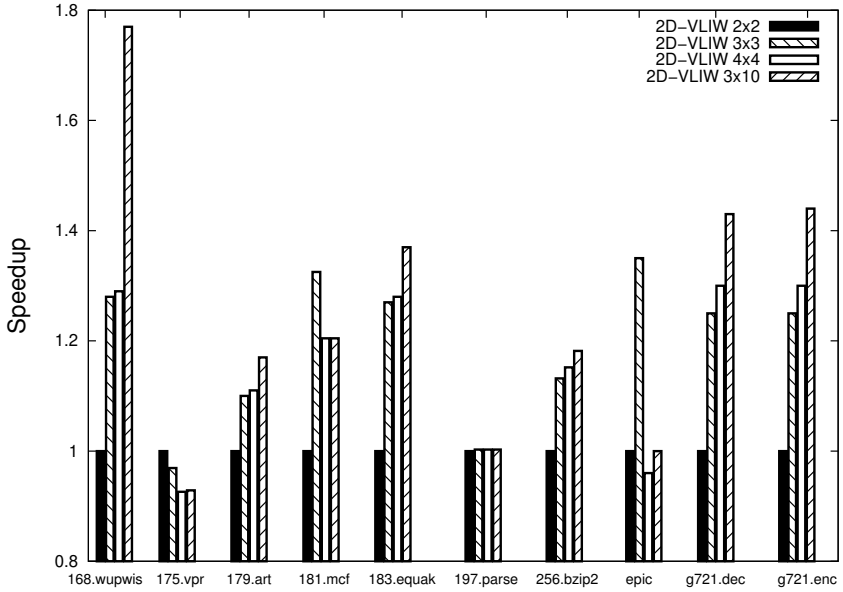
Observe que nos gráficos das Figuras 5.3(a) e 5.3(b), as barras cuja configuração indica 4 UFs (**2D-VLIW** 2×2 e **HPL-PD 4**) possuem *speedup*=1 e são chamadas de configurações base. As demais barras indicam o *speedup* de sua respectiva configuração sobre a configuração base. Assim, se uma determinada barra atinge, por exemplo, um *speedup*=1.2, isso signfica que sua respectiva configuração possui um desempenho 20% melhor que a configuração base. Da mesma forma, se uma configuração possui um *speedup*$<$ 1, essa configuração tem um desempenho pior que a configuração base.

Para alguns programas (por exemplo, epic e 175.vpr), o desempenho é decrementado a medida que o número de unidades funcionais é aumentada. Esse fenômeno, aumentar o número de unidades funcionais decrementa o desempenho, é devido principalmente ao aumento dos ciclos de *stalls* por parte das operações de memória. Pelo fato da configuração de memória não ser modificada a medida que se aumenta o número de unidades funcionais, há mais possibilidade de encontrar gargalos no acesso à memória já que mais operações estão tentando o acesso simultâneo. Apesar de ser encontrado em diferentes proporções, esse fenômeno acontece tanto no modelo 2D-VLIW quanto no modelo HPL-PD.

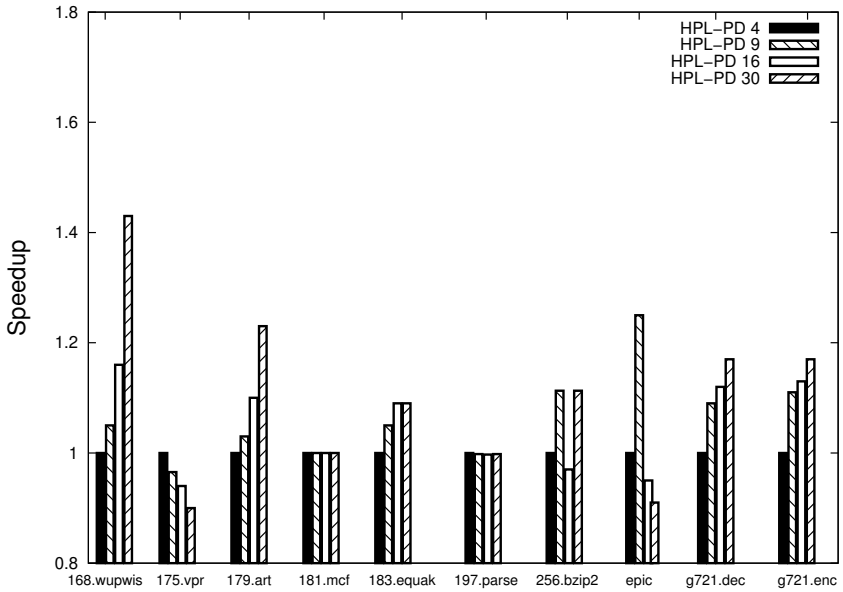
5.4 Escalonamento e Alocação

O primeiro experimento considerando o escalonamento de instruções utilizando os algoritmos definidos nas Seções 4.2 e 4.3 apresenta o número de instruções 2D-VLIW obtidos por cada algoritmo. O intuito deste experimento é determinar qual abordagem de escalonamento pode oferecer maiores ganhos no desempenho de programas sobre a arquitetura 2D-VLIW com uma configuração com 4×4 UFs. Deve-se ressaltar que para os experimentos envolvendo o escalonamento, os mesmos programas, hiperblocos, operações e latências de operações são considerados por ambos os algoritmos. Mais ainda, a correção é garantida através da verificação dos resultados de saída de cada algoritmo. Nesse caso, os dois algoritmos devem ter conseguido escalonar o mesmo número de programas, hiperblocos e operações.

Os resultados são apresentados na Tabela 5.3 onde os valores entre parênteses indicam os ganhos obtidos com a técnica de isomorfismo de subgrafos sobre a técnica de *list scheduling*. Os valores da linha **Média**, incluindo o valor correspondente ao ganho,



(a) Escalabilidade 2D-VLIW.



(b) Escalabilidade HPL-PD.

Figura 5.3: Escalabilidade dos modelos de execução 2D-VLIW e HPL-PD.

representam a média entre os valores resultantes de cada algoritmo sobre o conjunto de programas.

Programas	***List Scheduling*** **Número de Instruções**	**Escalonamento de Isomorfismo em Subgrafos Número de Instruções (Ganho %)**
168.wupwise	13.396	6.111 (54)
175.vpr	83.165	36.634 (56)
179.art	17.704	6.953 (61)
181.mcf	13.764	7.333 (47)
183.equake	18.268	6.761 (63)
197.parser	80.036	40.437 (49)
256.bzip2	16.297	16.208 (2)
epic	6.720	2.533 (62)
g721.decode	3.523	1.489 (58)
g721.encode	3.572	1.503 (58)
Média	**25.644,5**	**12.596,2** (51%)

Tabela 5.3: Número de instruções 2D-VLIW.

Os resultados da Tabela 5.3 mostram claramente que a estratégia baseada em isomorfismo de subgrafos gera menos instruções 2D-VLIW que o algoritmo guloso de *list scheduling* em todos os programas avaliados. A estratégia de isomorfismo de subgrafos obteve ganhos 2× melhores que o algoritmo de *list scheduling* em 7 (70%) dos 10 programas. O número médio de instruções que o escalonamento baseado em isomorfismo gerou é aproximadamente 51% menor que o número médio alcançado pelo *list scheduling*. Esses resultados têm um impacto direto no tamanho final do código dos programas e, conseqüentemente, em seu desempenho. Programas que possuem mais instruções terão que dedicar mais ciclos de busca para trazer da memória todas as operações que necessitam executar.

A razão principal para a diferença entre resultados obtidos pelo algoritmo de *list scheduling* e o algoritmo de isomorfismo reside na estratégia usada para garantir o escalonamento das operações dentro de um tempo pré-determinado. No caso de *list scheduling*, se o algoritmo não está conseguindo escalonar o vértice atual após uma quantidade de tentativas, a heurística de inserção de vértices globais é logo chamada. Com isso, o vértice consegue ser escalonado ao custo de considerar os ciclos de relógio para escrita e leitura do banco de registradores globais. O algoritmo de isomorfismo,

de outra forma, procura utilizar a heurística de redimensionamento do grafo base pois, em algumas situações, com mais "espaço disponível", o algoritmo converge rapidamente para uma solução. Apenas quando essa estratégia não surte os efeitos desejados é que o algoritmo de isomorfismo usará a inserção de vértices globais. É importante deixar claro que um escalonamento realizado apenas com a primeira estratégia (redimensionamento do grafo base) utiliza menos instruções já que não há necessidade de aguardar por escritas e leituras nos registradores globais. Os vértices do DAG continuam armazenando seus resultados e recuperando seus operandos diretamente dos registradores temporários.

A Tabela 5.4 compara o OPC e o OPI alcançado por ambos os algoritmos de escalonamento sobre núcleos de programas. Esses núcleos correspondem às funções e hiperblocos responsáveis por, aproximadamente, 90% do tempo de execução do programa. Os valores apresentados nas colunas 2 e 3 indicam o OPC e OPI, este último entre parênteses, respectivamente.

Programas	***List Scheduling*** **OPC (OPI)**	**Escalonamento de Isomorfismo em Subgrafos OPC (OPI)**
168.wupwise	1,44 (2,10)	2,03 (2,83)
175.vpr	1,20 (2,01)	5,40 (7,10)
179.art	1,10 (1,50)	3,80 (6,80)
181.mcf	1,60 (2,00)	2,65 (2,79)
183.equake	1,30 (2,01)	2,70 (2,95)
197.parser	1,85 (2,04)	6,80 (8,50)
256.bzip2	2,10 (2,40)	4,53 (5,36)
epic	1,10 (1,40)	1,50 (3,12)
g721.decode	1,50 (1,73)	2,70 (3,00)
g721.encode	1,60 (1,80)	2,70 (3,00)
Média	**1,48 (1,90)**	**3,48 (4,55)**

Tabela 5.4: OPC e OPI alcançados pelos algoritmos de escalonamento.

O algoritmo de isomorfismo de subgrafos obtém melhores resultados que o algoritmo de *list scheduling* em todos os programas considerados. Os ganhos com os valores de OPC e OPI por parte do algoritmo de isomorfismo chegam a ser até 4 vezes melhores que os do algoritmo guloso de *list scheduling*. Um resultado interessante surge quando os resultados obtidos neste experimento são comparados com aqueles obtidos em [21] que utiliza a arquitetura TRIPS apresentada na Subseção 2.2.2. Ao comparar

um subconjunto de programas (179.art, 256.bzip2, 183.equake e 197.parser) em comum entre os experimentos realizados com 2D-VLIW e em [21], observa-se que o OPC médio obtido em [21] foi de 3.62 enquanto que o OPC médio pelo algoritmo de isomorfismo foi de 4.45. Além disso, o algoritmo de escalonamento descrito em [21] não considera restrições de tempo em suas heurísticas enquanto que, tanto a abordagem de isomorfismo de subgrafos quanto a técnica de *list scheduling* obtêm o desempenho de pico tendo como base restrições no tempo de execução do algoritmo.

Um outro experimento avaliou o *speedup* obtido pelo algoritmo de isomorfismo sobre o algoritmo de *list scheduling*, levando em consideração o tempo gasto no escalonamento do código de um programa. Nesse experimento, o desempenho médio alcançado pelo algoritmo de isomorfismo foi em torno de 57.5 vezes melhor que o algoritmo de *list scheduling*. Esse experimento também tomou como base uma configuração 2D-VLIW com 4×4 UFs.

O experimento a seguir mostra alguns resultados considerando os ganhos em termos da redução de código para manipular *spills* no corpo dos programas ao adotar registradores temporários. Esse experimento consistiu na inclusão (ou retirada) de registradores temporários junto à especificação HMDES do modelo de execução 2D-VLIW. A Tabela 5.5 mostra a quantidade de código de *spill* gerado em cada programa, em tempo de compilação, utilizando uma configuração com registradores globais apenas (32 registradores) e quando um banco de registradores globais (32 registradores) e 32 registradores temporários (16 bancos de RTs $\times$ 2 registradores por RT) é adotado. Deve estar claro que esse experimento não utiliza o algoritmo de escalonamento de instruções e alocação de registradores integrado, conforme apresentado na Seção 4.4. Nesse sentido, o objetivo aqui é mostrar que há ganhos significativos com a inclusão de registradores temporários em uma configuração 2D-VLIW com 4×4.

É importante entender que, mesmo que a redução seja aparentemente pequena, isso pode resultar em um ganho considerável em tempo de execução. Por exemplo, nos programas g721.decode e g721.encode o único código de *spill* reduzido (de 18 para 17) se encontra, em ambos os programas, em laços interiores. Como, de maneira geral, laços

Programas	Registradores Globais	Registradores Globais + Bancos de Regs. Temporários
168.wupwise	14	9
179.art	40	24
181.mcf	16	14
183.equake	20	10
197.parser	77	34
256.bzip2	19	13
epic	12	6
g721.decode	18	17
g721.encode	18	17

Tabela 5.5: Números de códigos de *spill* inseridos no corpo do programa com diferentes configurações de registradores.

interiores são as partes mais executadas dos programas, uma economia no número de *spills* dentro desses laços melhora consideravelmente o desempenho final da aplicação.

5.5 Codificação de Instruções

Este experimento mostra o impacto da técnica de codificação sobre o tamanho do programa comparando os resultados com a estratégia de não codificar as instruções. A codificação das instruções e seus respectivos padrões são obtidos de acordo com as definições apresentadas na Seção 4.5 e considerando uma configuração 2D-VLIW com 4×4 UFs. A correção deste experimento é garantida por meio de um procedimento que simula a decodificação das instruções codificadas e seus padrões resultando em instruções não codificadas. Deve-se aqui atentar para o fato que os valores de saída desse algoritmo devem estar presentes no conjunto de instruções não codificadas. Há, no entanto, uma exceção para os casos de instruções codificadas que foram originadas a partir de "quebras" das instruções não codificadas. Essas "quebras" foram devido à extrapolação dos recursos arquiteturais e/ou à extrapolação das restrições de tamanho para uma instrução codificada. Nesse caso, o número de instruções decodificadas pelo algoritmo de decodificação será maior que o número de instruções não codificadas. Mesmo assim, a correção ainda pode ser realizada pois a diferença entre o número de instruções decodificadas e o número

de instruções não codificadas deve ser igual ao número de "quebras" computadas pelo algoritmo de codificação.

A Tabela 5.6 mostra os resultados da estratégia de codificação. As três primeiras colunas, $|S|$, $|I|$ e $|P|$, mostram o número de instruções não codificadas, instruções codificadas e padrões para cada programa. As instruções não codificadas foram obtidas através do escalonamento baseado em isomorfismo de subgrafos, como pode ser observado na Tabela 5.3. A coluna **Reuso** é a taxa de reuso dos padrões P por parte das instruções codificadas I. Em outras palavras, o **Reuso** informa quantas instruções codificadas utilizam o mesmo padrão. A coluna **Fator de Redução** indica a porcentagem de redução do tamanho de um programa que utiliza instruções codificadas quando comparado com um programa usando instruções não codificadas. Por exemplo, a versão codificada do programa 168.wupwise é 53% menor que o mesmo programa sem codificação. Os valores das colunas **Reuso** e **Fator de Redução** são calculados de acordo com as Equações 5.2 e 5.3, respectivamente.

$$\textbf{Reuso} = \frac{|I|}{|P|} \tag{5.2}$$

$$\textbf{Fator de Redução} = 1 - \frac{((|I| \times \alpha) + (|P| \times \beta))}{(|S| \times \gamma)} \tag{5.3}$$

sendo:

- α é o tamanho da instrução codificada. No exemplo considerado neste experimento, cada instrução possui 128 bits.
- β é o tamanho de cada padrão de instrução. Cada padrão possui 496 bits.
- γ é o tamanho da instrução não codificada. Cada instrução 2D-VLIW agrupa 16 operações, cada uma possuindo 32 bits. O tamanho total de uma instrução não codificada é 512 bits.

Os resultados da Tabela 5.6 confirmam algumas premissas declaradas na Seção 4.5. Os valores da Coluna **Reuso** representam a sobrejeção entre instruções codificadas e seus padrões. Como já mencionado, a sobrejeção é uma condição necessária para decrementar

Programas	$\|S\|$	$\|I\|$	$\|P\|$	**Reuso**	**Fator de Redução** (%)
168.wupwise	6.111	7.746	957	8,09	53
175.vpr	36.634	40.811	5.066	8,06	59
179.art	6.953	8.880	1.025	8,66	54
181.mcf	7.333	8.028	1.160	6,92	57
183.equake	6.761	8.383	1.159	7,23	52
197.parser	40.437	41.959	5.345	7,85	61
256.bzip2	16.208	18.194	2.212	8,23	59
epic	2.533	3.066	674	4,55	44
g721.decode	1.489	1.607	391	4,11	48
g721.encode	1.503	1.624	399	4,07	47
Média	12.596,2	14.029,8	1.838,8	6,78	53,4

Tabela 5.6: Resultados com a técnica de codificação. $|S|$ =número de instruções não codificadas; $|I|$ =número de instruções codificadas; $|P|$ =número de padrões.

o tamanho do programa, uma vez que o tamanho da instrução codificada e o tamanho do padrão juntos é maior que o tamanho de uma instrução não codificada, $\alpha + \beta > \gamma$. Além disso, a sobrejeção reduz as possibilidades de perdas de desempenho devido à freqüência de *misses* simultâneos na cache de instruções e de padrões.

A Tabela 5.6 também revela outro resultado interessante: o impacto das restrições arquiteturais (uso restrito de portas de leitura e escrita nos registradores globais) e do algoritmo de codificação (menor número de operações com imediatos) não causaram um aumento significativo no número de instruções codificadas. O aumento no número de instruções codificadas foi de $8\% - 27\%$, tendo um aumento médio em torno de 15%. Contudo, como pode ser observado pela coluna **Fator de Redução**, esse aumento não foi significativo a ponto de impactar sobre o tamanho do programa.

Na realidade, essa técnica de codificação de instruções é factível mesmo quando comparada com uma codificação bastante compacta como é o caso da técnica utilizada por arquiteturas x86. Tomando como exemplo os valores das colunas $|I|$ e $|P|$ para o programa 181.mcf, pode-se observar que esse programa possui 128.488 bytes ($8.028 \times (128/8)$) utilizados pelas instruções codificadas e 71.920 bytes ($1.160 \times (496/8)$) pelos padrões. Considerando que instruções e padrões compõem o segmento de código de um programa, conclui-se que esse programa possui, aproximadamente, 200.408 bytes ($128.488 + 71.920$) de código. Na arquitetura x86, utilizando o compilador gcc versão 4.1.1 com a opção de

desenrolamento de laços ativada e a opção de ligação dinâmica desativada, esse mesmo programa possui 477.738 bytes de código. O menor tamanho do segmento de código obtido com a técnica de codificação 2D-VLIW comparado com o tamanho do segmento de código obtido com a codificação x86 pôde ser notado em um subconjunto dos programas SPEC e em todos os programas MEDIABench avaliados.

Deve-se também deixar claro que os valores apresentados na Tabela 5.6 cobrem uma avaliação estática da técnica de codificação. O efeito dessa codificação considerando a busca de instruções e padrões em suas respectivas memórias cache fazem parte de uma avaliação dinâmica. Tal avaliação é implementada no trabalho de mestrado que estuda algoritmos e técnicas para codificação de instruções 2D-VLIW, atualmente em desenvolvimento no LSC/IC-UNICAMP.

5.6 Considerações Finais

Vários experimentos envolvendo os conceitos e proposições abordados ao longo desta Tese foram apresentados neste capítulo. Além de mostrar os resultados dos experimentos, procurou-se detalhar a metodologia adotada assim como as ferramentas e critérios utilizados. Realizar experimentos em projetos envolvendo processadores é uma atividade árdua pois envolve a necessidade de construir e integrar diversas ferramentas. Nos casos onde o hardware ainda não está disponível, como é o caso de 2D-VLIW, ferramentas de simulação são utilizadas, o que não minimiza a dificuldade e o esforço despendido para conseguir realizar experimentos. É interessante observar que, com o passar dos anos, a preocupação em ter um conjunto de instâncias que avaliam, de forma representativa, projetos na área de arquitetura de computadores têm aumentado significativamente. O trabalho de Joshi e outros [43] discute a questão da quantidade de programas realmente necessários para validar o desempenho de processadores e propõe o uso de um conjunto reduzido, mas representativo, de programas para essa validação. Outro ponto de muita discussão no que concerne à avaliação de trabalhos envolvendo arquiteturas de processadores diz respeito à modelagem e simulação. O trabalho de Skadron e outros [75] apresenta essa questão e discute resultados interessantes sobre o crescimento de avaliações experimentais baseadas em simulação. Um de seus resultados

mostra uma crescente adoção em experimentos que utilizam simulação, em contraposição aqueles que possuem uma implementação final do sistema. Esse resultado tem como base o os artigos aceitos no Simpósio Internacional de Arquiteturas de Computadores (ISCA - *International Symposium on Computer Architecture*) no período de 1985-2001. Como exemplo, um dado revelado pelos autores é que mais de 90% dos artigos aceitos, nesse simpósio, no ano de 2001 utilizaram simulação, enquanto que essa porcentagem foi de apenas 28% em 1985. Além de SPEC e MEDIABench, um outro benchmark bastante utilizado em avaliações experimentais na área de arquitetura de computadores é o MiBench [36].

O próximo capítulo retoma as motivações e objetivos principais que nortearam o desenvolvimento desta Tese. As contribuições alcançadas com o desenvolvimento deste trabalho, na forma de artigos e relatórios publicados, também são listadas. Por fim, diversas sugestões para o desenvolvimento de trabalhos futuros são apresentadas e discutidas.

Capítulo 6

Conclusões

A proposta de um novo modelo de execução para arquiteturas de processadores, assim como algoritmos que suportam esse modelo foram apresentados e discutidos nesta Tese. O objeto de estudo nesta Tese vai ao encontro de uma lacuna existente na área de arquitetura de processadores que é a proposição de modelos de execução e algoritmos para geração de código que consideram a geometria dos grafos de dependência entre operações de programas. A motivação para lidar com essa abordagem se dá pela necessidade crescente em investigar novas alternativas de organização do hardware na tentativa de sobrepor algumas questões abertas e barreiras tecnológicas já conhecidas pela comunidade da área de arquitetura de computadores. Dentre essas, destacam-se a barreira térmica devido à miniaturização contínua dos transistores e a disparidade entre o desempenho do processador e da memória gerando o problema conhecido como *Memory Wall*.

Este trabalho iniciou com uma busca visando dimensionar e caracterizar o estado da arte em termos de arquiteturas que exploram paralelismo em nível de instrução através de novas organizações dos elementos de hardware, técnicas avançadas de geração de código e novos paradigmas de execução. Um selecionado dessas arquiteturas faz parte do Capítulo 2. Ao elaborar esse capítulo, o intuito não foi apenas apresentar trabalhos relacionados mas também realizar uma análise crítica e compará-los qualitativamente com a proposta desta Tese.

O Capítulo 3 abordou as características de hardware da arquitetura 2D-VLIW. Na perspectiva do entendimento geral da arquitetura, os principais conceitos e características provenientes do projeto da arquitetura 2D-VLIW são comentados. O capítulo apresentou

os elementos mais relevantes que compõem essa arquitetura, além de justificar as decisões de projeto. Procurou-se também mostrar os ganhos obtidos com esse projeto através da análise da redução da complexidade de vários elementos. Além de ser o ponto de partida para o entendimento dos elementos que fazem parte da arquitetura, esse capítulo exemplificou o modelo de execução 2D-VLIW.

As questões envolvendo a geração de código para a arquitetura 2D-VLIW foram discutidas no Capítulo 4. É nesse capítulo que a infraestrutura de compilação e simulação de código foi definida. Além de apresentar as lacunas existentes para geração de código em arquiteturas com restrições arquiteturais, como é o caso de 2D-VLIW, o Capítulo 4 descreveu as soluções adotadas para resolvê-las. Houve uma preocupação adicional em mostrar não apenas os algoritmos desenvolvidos mas também analisá-los em termos de suas complexidades de tempo no pior caso. Da mesma forma que no capítulo que apresentou os elementos que compõem a arquitetura, procurou-se aqui justificar as decisões tomadas e mostrar os ganhos obtidos.

O Capítulo 5 apresentou os experimentos realizados na validação dos conceitos principais desta Tese. É nesse capítulo que se tem uma análise quantitativa dos ganhos advindos do melhor desempenho de 2D-VLIW. A definição de quais experimentos, o detalhamento de como os experimentos foram realizados e as justificativas em torno dos resultados formaram o cerne desse capítulo. Outrossim, o ponto importante foi demonstrar que a arquitetura 2D-VLIW pode trazer ganhos significativos para o desempenho final das aplicações. Mais ainda, o capítulo ratificou a existência de algumas contribuições tanto no campo de projeto de arquiteturas de processadores quanto no projeto e desenvolvimento de algoritmos que compõem as etapas da geração de código em um compilador.

6.1 Contribuições desta Tese

Durante o seu desenvolvimento, esta Tese foi caracterizada como uma frente de trabalho em diversas sub-áreas envolvendo o projeto de arquiteturas de processadores. Além deste trabalho, em nível de doutorado, dois outros trabalhos em nível de mestrado estão sendo desenvolvidos a partir de problemas investigados inicialmente através desta Tese.

Justamente por abordar atividades em diversas sub-áreas, as contribuições desta Tese podem ser agrupadas em dois grandes grupos: as contribuições relacionadas à área de arquitetura de computadores e aquelas relacionadas à geração de código. Em se tratando de arquitetura de computadores as seguintes contribuições foram realizadas:

- A proposição de uma nova arquitetura de processador que explora o paralelismo em nível de instrução através de uma nova organização dos elementos arquiteturais como unidades funcionais e registradores;
- A adoção de um modelo de execução que captura as dependências de dados entre operações de um programa e obedece essas dependências sem a introdução de elementos de hardware com propósito exclusivo para esse fim;
- A introdução de um esquema de codificação de instruções que utiliza uma memória cache de padrões no estágio de decodificação, aproveitando assim um paralelismo inerente ao esquema de execução e excluindo a necessidade de adicionar mais um estágio de *pipeline*;
- A comprovação empírica, através de experimentos e comparações com arquiteturas que exploram paralelismo em nível de instrução, dos ganhos obtidos com esse novo modelo de execução;
- A ratificação que a redução no número de portas do banco de registradores é uma alternativa factível para o projeto de arquiteturas de processadores e pode trazer ganhos efetivos em termos de área e latência.

A despeito da geração de código, mais algumas contribuições foram realizadas:

- A demonstração das possibilidades de geração de código para processadores com diversas restrições arquiteturais em termos de portas de registradores e interligação entre unidades funcionais;
- A proposição e implementação de heurísticas de escalonamento e alocação de registradores que exploram a geometria dos grafos de entrada assim como elementos de hardware;

- A utilização de técnicas de codificação de instruções por meio de um algoritmo baseado em fatoração de operandos e operadores. Além disso, a implementação e a utilização do mesmo junto a uma infraestrutura de compilação;
- A proposição, implementação e utilização de diversas heurísticas que facilitam e aceleram a convergência dos algoritmos de escalonamento e alocação baseados em isomorfismo de subgrafos;
- O tratamento dos problemas clássicos de escalonamento de instruções e alocação de registradores de maneira integrada;

A busca constante em divulgar os resultados obtidos permeou todo o desenvolvimento deste trabalho. Antes mesmo da primeira publicação em evento ou periódico, houve a preocupação em publicar um relatório técnico que servisse como referência básica para os primeiros artigos da arquitetura 2D-VLIW. Dessa forma, a listagem a seguir apresenta, em ordem crescente pela data de submissão, o relatório técnico e as publicações provenientes deste trabalho e fornece uma descrição rápida sobre cada uma. Além dos artigos já publicados, há também os artigos submetidos recentemente e que ainda estão sob avaliação.

1. R. Santos, R. Azevedo, G. Araujo. *The 2D-VLIW Architecture*. Technical Report IC-03-06. Institute of Computing, State University of Campinas, 2006, Campinas-São Paulo, Brazil.

   Esse relatório técnico foi o primeiro trabalho descrevendo a arquitetura 2D-VLIW. O texto focou a apresentação detalhada dos recursos existentes na arquitetura e de seu modelo de execução.

2. R. Santos, R. Azevedo, G. Araujo. *Exploiting Dynamic Reconfiguration Techniques: The 2D-VLIW Approach*. In Proceedings of the 13th Reconfigurable Architectures Workshop (RAW) joint with the 24th IEEE International Parallel Distributed Processing Symposium (IPDPS). Rhodes-Island, Greece, 2006, IEEE Computer Society.

   Essa publicação tratou da adoção da arquitetura 2D-VLIW voltada para reconfiguração dinâmica. Considerou-se aqui a possibilidade de utilizar a memória cache de

padrões como uma memória cache de configurações. O artigo também realizou uma comparação de características entre 2D-VLIW e arquiteturas com reconfiguração dinâmica.

3. R. Santos, R. Azevedo, G. Araujo. *2D-VLIW: An Architecture Based on the Geometry of the Computation*. In Proceedings of the 17th IEEE International Conference on Application-specific, Systems, Architectures and Processors (ASAP). Steamboat Springs-Colorado, USA, 2006, IEEE Computer Society.

   Essa foi a publicação que detalhou as características principais da arquitetura 2D-VLIW. Os recursos de hardware presentes na arquitetura foram apresentados e discutiu-se as razões dos ganhos de 2D-VLIW em comparação com o modelo EPIC, representado pelo processador HPL-PD.

4. R. Santos, R. Azevedo, G. Araujo. *The 2D-VLIW Architecture Model*. ACM Transactions on Architectures and Compilation Techniques (TACO). 2007, ACM Press. (submetido)

   Esse artigo abordou detalhadamente a arquitetura, o modelo de execução e a infraestrutura para geração de código. Além da descrição minuciosa de todos esses elementos, houve a preocupação em apresentar experimentos que ratificaram os conceitos principais da proposta 2D-VLIW.

5. R. Santos, R. Azevedo, R. M. O. Santos. *A DAGs-Packing Heuristic for a High Performance Processor Architecture*. In Proceedings of the Brazilian Symposium on Operational Research (SBPO). Fortaleza-Ceará, Brazil. 2007. (submetido)

   Esse artigo apresentou uma abordagem mais teórica sobre o projeto e desenvolvimento de uma das heurísticas utilizadas durante o escalonamento de instruções 2D-VLIW. O objetivo foi mostrar que a heurística permite a modelagem e resolução do problema de Empacotamento de Faixas[1], bem conhecido pela comunidade da área de otimização combinatória, como uma fase do escalonamento de instruções baseado em isomorfismo de subgrafos.

[1]Do inglês: *Strip Packing*.

6. R. Santos, R. Azevedo, G. Araujo. *An Instruction Scheduling Algorithm Based on Subgraph Isomorphism*. In Proceedings of the 16th ACM International Symposium on Parallel Architectures and Compilation Techniques (PACT). Brasov, Romania, 2007, ACM Press. (submetido)

   Nesse artigo, o objetivo foi detalhar a resolução do problema de escalonamento de instruções através de uma heurística que resolve o problema de isomorfismo de subgrafos. Uma das motivações foi mostrar que mesmo um algoritmo para resolução de um problema $\mathcal{NP}$-completo, acoplado com heurísticas eficientes, fornece soluções muito boas para o problema de escalonamento de instruções. Ademais, discutiu-se também a questão do emprego de algoritmos de escalonamento clássicos que tratam o grafo de entrada de maneira local, como é o caso de algoritmos baseados em *list scheduling*.

Além desses artigos já publicados e sob avaliação, há um outro artigo, em fase final de escrita, que discorre sobre a técnica de codificação e apresenta resultados estáticos e dinâmicos com diversos programas. Esse artigo será submetido ao Brazilian Symposium on Computer Architectures (Simpósio Brasileiro de Arquiteturas de Computadores - SBAC) de 2007.

6.2 Propostas de Trabalhos Futuros

Como mencionado na Seção 6.1, uma das características marcantes desta Tese foi a abertura de trabalhos em várias sub-áreas. Diante desse contexto e com a complexidade inerente à pesquisa envolvendo a proposição de novos paradigmas e conceitos na área de arquitetura de computadores, há diversas possibilidades de extensão deste trabalho. A listagem a seguir, fornece algumas sugestões para desenvolvimento de pesquisas futuras:

- Estudar o impacto das operações com latências variáveis na arquitetura 2D-VLIW e propor soluções que minimizam o número de *stalls* do processador;

  O intuito aqui é investigar o impacto de operações com latências variáveis sobre o número de *stalls* de um processador baseado em 2D-VLIW e, principalmente, propor otimizações de código de forma que esse número seja minimizado. Pode-se, inclusive,

estudar pequenas modificações ou acréscimos de funcionalidades em uma unidade funcional da matriz de UFs a fim de atender a demanda das operações com latências variáveis. Contudo, deve-se ter a preocupação em manter o modelo de execução e a organização das UFs o mais simples possível pois essa é uma característica intrínseca ao projeto 2D-VLIW.

- Integrar as ferramentas de escalonamento de instruções, alocação de registradores e codificação de instruções junto ao compilador Trimaran com o intuito de possibilitar um fluxo de execução completo e, ainda, aproveitar as características de simulação desse compilador;

  O desenvolvimento dessa proposta deve fechar o trabalho a respeito de uma infra-estrutura de compilação para a arquitetura 2D-VLIW. Além disso, há aqui uma contribuição pujante para pesquisas futuras que propõem novas otimizações de código ou melhorias no processo de geração de código. Parte dessa proposta está sendo discutida e, pretende-se implementá-la no âmbito do projeto de mestrado do aluno Luís Guilherme Pereira que investiga técnicas de *spill code* junto ao algoritmo de escalonamento e alocação de registradores.

- Realizar experimentos com a técnica de codificação visando à determinação de parâmetros para o tamanho da instrução codificada, tamanho do padrão e organização da cache de padrões;

  Através de experimentos com padrões de instruções e com base na taxa de ocupação de instruções codificadas, deve-se buscar o menor tamanho para uma instrução codificada e o padrão da instrução sem sacrificar o tamanho do programa como um todo. Ademais, os experimentos podem ir além e auxiliar na determinação da organização da cache de padrões com relação à política de substituição de objetos, tamanhos de blocos e conjuntos, entre outros parâmetros. Parte desse trabalho está sendo desenvolvido no projeto de mestrado do aluno Rafael Fernandes Batistella e há, inclusive, um artigo submetido para uma conferência contendo resultados estáticos e dinâmicos envolvendo a técnica de codificação.

- Comparar quantitativamente o modelo de execução da arquitetura 2D-VLIW com modelos de outras arquiteturas baseadas em matrizes de unidades funcionais como TRIPS, RAW e até com processadores comerciais como os processadores TMS e a arquitetura Itanium;

  O foco desse trabalho é a construção de uma infraestrutura de simulação que possibilite a avaliação e comparação de modelos de execução próximos ao modelo 2D-VLIW. Apesar do objetivo ser bem definido, a execução desse trabalho exige um estudo considerável a fim de determinar parâmetros de comparação, características das arquiteturas, programas e instâncias de entrada a serem utilizadas, entre outros.

- Realizar um estudo teórico sobre as características dos DAGs e do grafo da arquitetura e propor algoritmos de isomorfismo de subgrafos que tirem proveito dessas características e obtenham melhores desempenhos. Além disso, verificar a otimalidade da solução proposta pelo algoritmo de escalonamento adotado atualmente;

  A característica principal desse projeto é fazer uma análise teórica sobre a otimalidade das soluções obtidas pelo algoritmo de escalonamento baseado em isomorfismo de subgrafos. Há uma intuição de que os grafos G'_2 - obtidos como resultado do isomorfismo - sejam boas soluções para o escalonamento. No entanto, não se sabe quão distante da solução ótima está a solução fornecida pelo algoritmo. Em acréscimo a esse estudo, é possível realizar um estudo mais detalhado considerando a qualidade da solução das heurísticas utilizadas. Em particular, uma das heurísticas que merece atenção especial é aquela que infere o tamanho do grafo base a partir do tamanho do grafo de entrada. Acredita-se que uma formulação do problema de Empacotamento de Faixas baseada em programação linear inteira pode fornecer melhores resultados.

- Estudar e realizar experimentos considerando potência e área consumida por um processador 2D-VLIW;

  Essa proposta está voltada para a questão da análise dos recursos de hardware presentes em um processador 2D-VLIW. O objetivo aqui é avaliar quantitativamente as características de potência e área consumida pela arquitetura 2D-VLIW. Além

disso, uma contribuição relevante seria usar informações de consumo de potência das operações e agregar essas informações junto às ferramentas de geração de código. Nesse sentido, pode-se tomar como base trabalhos anteriores que realizaram análises de potência sobre arquiteturas que exploram paralelismo em nível de instrução [11, 48] ou trabalhos que adicionaram algoritmos para avaliação de potência junto a compiladores [6, 83].

- Dotar a linguagem HMDES com recursos para possibilitar a descrição da interconexão entre UFs e registradores e inserir características no simulador Trimaran para realizar a simulação de arquiteturas com diferentes modelos de execução e organizações arquiteturais;

  Em adição à proposta envolvendo a integração das ferramentas de escalonamento e alocação no compilador Trimaran, esse trabalho teria por objetivo adicionar novas características à linguagem de descrição de hardware HMDES na tentativa de aumentar o leque de arquiteturas que podem ser descritas através dessa linguagem. Apesar de estar fortemente relacionado com a linguagem HMDES, essa proposta ainda exige o conhecimento e implementação de novas funcionalidades no cerne do compilador Trimaran.

- Realizar a implementação em hardware de uma instância da arquitetura 2D-VLIW;

  Esse trabalho faria a implementação do protótipo de um processador 2D-VLIW em uma tecnologia de hardware. Inicialmente, a tecnologia de FPGAs (*Field Programmable Gate Arrays*) seria usada. Essa implementação deve seguir as especificações da arquitetura e possibilitar mais experimentos envolvendo uma implementação real da arquitetura.

Referências Bibliográficas

[1] A. Adl-Tabatabai, P. Dubey, D. Dunning, M. Espig, E. Grochowski, A. Gonzalez, S. Hahn, R. Huggahalli, J. Jayasimha, A. Kumar P. Kundu, T. Mattson, D. McAuley, A. Munoz, and C. Narad. Intel Tera-Scale Research Program Focuses on Moving from a Few Cores to Many. *Technology@Intel Magazine*, pages 1–7, 2006.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques, & Tools*. Addison Wesley, 2 edition, 2007.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, 1986.

[4] A. Aiken and A. Nicolau. Optimal Loop Parallelization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–317. ACM Press, April 1988.

[5] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code Compression Based on Operand Factorization. In *Proceedings of the* 31^{st} *Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 194–201. IEEE Computer Society, 1998.

[6] R. Banakar, M. Ekpanyopang, K. Puttaswamy, R. Rabbah, M. Balakrishnan, V. Mooney, and K. Palem. A System Level Energy Model for HPL-PD Microarchitecture in Trimaran Framework (TRIREME). Technical Report GIT-CC-02-35, Georgia Institute of Technology, 2002.

[7] F. Barat, R. Lauwereins, and G. Deconinck. Reconfigurable Instruction Set Processors from a Hardware/Software Perspective. *IEEE Transactions on Software Engineering*, 28(9):847–860, 2002.

[8] R. Bittner and P. Athanas. WormHole Run-Time Reconfiguration. In *Proceedings of the* 5^{th} *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 79–85. ACM Press, February 1997.

[9] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. The Macmillan Press, 1976.

[10] P. Briggs, K. D. Cooper, K. kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI)*, pages 275–284. ACM Press, 1989.

[11] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 83–94. ACM Press, 2000.

[12] D. Burger and J. Goodman. Billion-Transistor Architectures: There and Back Again. *IEEE Computer*, 37(3):22–28, 2004.

[13] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burril, R. G. Mcdonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, 2004.

[14] A. Capitanio, N. Duit, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proceedings of the* 25^{th} *IEEE International Symposium on Microarchitecture (MICRO)*, pages 292–300. IEEE Computer Society, 1992.

[15] G. J. Chaitin, M. A. Auslander, A. K. Shandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, 1981.

[16] L. N. Chakrapani, J. Gyllenhaal, W. Mei, W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran - An Infrastructure for Research in Instruction-Level Parallelism. *Lecture Notes in Computer Science*, 3602:32–41, 2004.

[17] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen. PipeRench Implementation of the Instruction Path Coprocessor. In *Proceedings of the* 33^{rd} *IEEE/ACM Annual International Symposium on Microarchitecture (MICRO)*, pages 147–158. IEEE Computer Society, December 2000.

[18] D. Citron and D. G. Feitelson. Revisiting Instruction Level Reuse. In *Proceedings of the Workshop on Duplication, Deconstructing and Debunking (WDDD)*, pages 62–70, May 2002.

[19] R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In *Proceedings of the* 2^{nd} *International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS)*, pages 180–192. IEEE Computer Society, 1987.

[20] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):170–210, 2002.

[21] K. E. Coons, X. Chen, S. K. Kushwaha, D. Burger, and K. S. McKinley. A Spatial Path Scheduling Algorithm for EDGE Architectures. In *Proceedings of the* 12^{th} *ACM Annual International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 40–51. ACM Press, October 2006.

[22] L. P. Cordella, P. Foggia, C. Sansona, and M. Vento. An Improved Algorithm for Matching Large Graphs. In *Proceedings of the* 3^{rd} *IAPR TC15 Workshop on Graph-Based Representations*, pages 149–159, Ischia - Italy, 2001.

[23] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Evaluating Performance of the VF Graph Matching Algorithm. In *Proceedings of the* 10^{th} *International Conference on Image Analysis and Processing*, pages 1172–1177. IEEE Computer Society, 1999.

[24] H. Corporaal. ILP Architectures: Trading Hardware for Software Complexity. In *Proceedings of the* 3^{rd} *International Conference on Algorithms and Architectures for Parallel Processing*, pages 1412–154. IEEE Computer Society, 1997.

[25] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture - A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1999.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the* 16^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35, Austin-TX, 1989. ACM Press.

[27] Carole Dulong. The IA-64 Architecture at Work. *IEEE Computer*, 31(7):24–32, 1998.

[28] H. El-Rewini and M. Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing*. John Wiley & Sons, New Jersey, 2005.

[29] D. Eppstein. Subgraph Isomorphism in Planar Graphs and Related Problems. *Journal of Graph Algorithms and Applications*, 3(3):1–27, 1999.

[30] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code Compression. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 358–365. ACM Press, 1997.

[31] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the* 10^{th} *International Symposium on Computer Architecture (ISCA)*, pages 140–150. IEEE Computer Society, 1983.

[32] M. Franz and K. Thomas. Slim Binaries. *Communications of the ACM*, 40(2):87–94, 1997.

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

[34] P. H. Gibbons and S. S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 11–16. ACM Press, 1986.

[35] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4):70–76, 2000.

[36] M.R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the 2001 IEEE International Workshop on Workload Characterization*, pages 3–14. IEEE Computer Society, 2001.

[37] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3, Center for Reliable and High-Performance Computing - University of Illinois at Urbana-Champaign, Urbana-Champaign-Illinois, 1996.

[38] S. Hack. Interference Graphs of Programs in SSA-Form. Technical Report ISSN 1432-7864, Universitat Karlsruhe, 2005.

[39] T. Hara, H. Ando, C. Nakanishi, and M. Nakaya. Reconfigurable Computing: A Survey of Systems and Software. *ACM SIGARCH Computer Architecture News*, 24(2):213–224, 1996.

[40] J. L. Henessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[41] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, 2000.

[42] D. S. Johnson. Fast Algorithms for Bin Packing. *Journal of Computer and System Sciences*, 8:272–314, 1974.

[43] A. Joshi, A. Phansalkar, L. Eeckhout, and K. John. Measuring Benchmark Similarity Using Inherent Program Characteristics. *IEEE Transactions on Computers*, 55:769–782, 2006.

[44] N. P. Jouppi. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the* 3^{rd} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 272–282. ACM Press, 1989.

[45] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4):589–603, 2005.

[46] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh Architecture Specification: Version 1.1. Technical Report 93-80, Hewlett Packard Laboratories Palo Alto, 1994.

[47] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL-PD Architecture Specification: Version 1.1. Technical Report 93-80, Hewlett Packard Laboratories Palo Alto, 2000.

[48] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. A Framework for Energy Estimation of VLIW Architecture. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 40–45. IEEE Computer Society, 2001.

[49] C.E. Kozyrakis and D. A. Patterson. A New Direction for Computer Architecture Research. *IEEE Computer*, 31(11):24–32, 1998.

[50] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the* 19^{th} *International Symposium on Computer Architecture (ISCA)*, pages 46–57. ACM Press, 1992.

[51] C. Lee, M. Potkonjak, and W. H. Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the* 30^{th} *Annual International Symposium on Microarchitecture (MICRO)*, pages 330–335, Research Triangle Park - North Carolina, 1997. IEEE Computer Society.

[52] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the* 8^{th} *International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 46–57. ACM Press, 1998.

[53] 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} *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–54. IEEE Computer Society, December 1992.

[54] J. Markoff. Intel's Big Shift After Hitting Technical Wall. *The New York Times*, May 2004.

[55] S. McGeady. The 1960CA Superscalar Implementation of the 80960 Architecture. *IEEE Computer*, pages 232–240, 1990.

[56] B. Mei, A. Lambrechts, J-Y Mignolet, D. Verkest, and R. Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design & Test of Computers*, 22(2):90–101, 2005.

[57] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, pages 166–173. IEEE Computer Society, 2001.

[58] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Adres: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proceedings of the* 13^{th} *International Conference on Field-Programmable Logic and Applications (FPLA)*, pages 61–70, 2003.

[59] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Instruction Scheduling for a Tiled Dataflow Architecture. In *Proceedings of the* 16^{th} *International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 141–150. ACM Press, 2006.

[60] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Modeling Instruction Placement on a Spatial Architecture. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 158–169. ACM Press, 2006.

[61] E. Mirsky and A. DeHon. A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE Computer Society, 1996.

[62] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[63] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. Keckler. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *Proceedings of the* 13^{th} *ACM Annual International Parallel Architecture and Compilation Techniques (PACT)*, pages 74–84. IEEE Computer Society, 2004.

[64] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of the* 34^{th} *IEEE/ACM Annual International Symposium on Microarchitecture*, pages 40–51. IEEE Computer Society, 2001.

[65] F. M. Q. Pereira and J. Palsberg. Register Allocation via Coloring of Chordal Graphs. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, pages 315–329, 2005.

[66] D. A. Pucknell and K. Eshraghian. *Basic VLSI Design: Systems and Circuits*. Prentice Hall, Sydney, 2 edition, 1988.

[67] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the* 27^{th} *Annual International Symposium on Microarchitecture (MICRO)*, pages 63–74. ACM Press, December 1994.

[68] R. Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the* 27^{th} *IEEE/ACM Annual International Symposium on Microarchitecture (MICRO)*, pages 172–180. IEEE Computer Society, November 1994.

[69] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the* 30^{th} *IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 422–434. ACM Press, May 2003.

[70] M. S. Schlansker and B. R. Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. Technical Report 99-111, Hewlett Packard Laboratories Palo Alto, February 2000.

[71] M. S. Schlansker and B. R. Rau. EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer*, 33(2):37–45, 2000.

[72] H. Schmit. Incremental Reconfiguration for Pipelined Applications. In *Proceedings of the* 5^{th} *IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM)*, pages 47–55. IEEE Computer Society, 1997.

[73] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. 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, Florida, May 2002. IEEE Computer Society.

[74] R. Sethi. Complete Register Allocation Problems. In *Proceedings of the* 5^{th} *Annual ACM Symposium on Theory of Computation*, pages 182–195. ACM Press, 1973.

[75] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in Computer Architecture Evaluation. *IEEE Computer*, 36(8):30–36, 2003.

[76] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on Multiple Instruction Issue. In *Proceedings of the* 3^{rd} *International Conference on Architectural Support*

for Programming Languages and Operating Systems (ASPLOS), pages 290–302. ACM Press, 1989.

[77] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the* 24^{th} *Annual International Symposium on Computer Architecture (ISCA)*, pages 194–205. ACM Press, 1997.

[78] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Dataflow: The Road Less Complex. In *Proceedings of the* 30*th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, 2003.

[79] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the* 36^{th} *IEEE International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2003.

[80] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. Eggers. The WaveScalar Architecture. *ACM Transactions on Computer Systems*, 2007.

[81] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the* 31^{st} *Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13. IEEE Computer Society, 2004.

[82] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, 1976.

[83] N. Vijaykrishnan, M. Kandemir, A. Sivasubramanian, and M. J. Irwin. *Power Aware Design Methodologies*, chapter Tools and Techniques for Integrated Hardware-Software Energy Optimizations, pages 277–296. Kluwer Academic Publishers, 2002.

[84] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Argawal. Baring it All to Software: RAW Machines. *IEEE Computer*, 30(9):86–93, 1997.

[85] D. W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the* 4^{th} *ACM Annual International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 176–188. ACM Press, 1991.