Dissertação de Mestrado

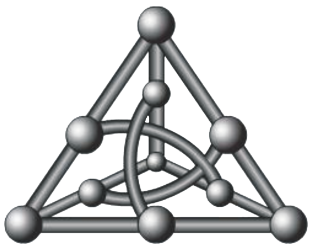
Algoritmos para Escalonamento de Instruções e Alocação de Registradores na Infraestrutura LLVM

Lucas da Costa Silva

Orientação: Prof. Dr. Ricardo Ribeiro dos Santos

Área de Concentração: Sistemas Computacionais

Palavras-chave: Compiladores, LLVM, Escalonamento de Instruções, Alocação de Registradores



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
27 de março de 2013.

Algoritmos para Escalonamento de
Instruções e Alocação de Registradores
na Infraestrutura LLVM

Campo Grande, 27 de março de 2013.

Banca Examinadora:

- Prof. Dr. Ricardo Ribeiro dos Santos (FACOM/UFMS) - orientador
- Prof. Dr. Sandro Rigo (IC/UNICAMP)
- Prof. Dr. Amaury Antonio de Castro Júnior (CPPP/UFMS)

Resumo

O objetivo deste trabalho é apresentar uma proposta integrada para Escalonamento de Instruções e Alocação de Registradores baseada em Isomorfismo de Subgrafos implementada no compilador LLVM. A contribuição principal deste trabalho é mapear as fases de escalonamento instruções e alocação de registradores como um problema de Isomorfismo de Subgrafos. A resolução é baseada na modelagem dos recursos de *hardware* (unidades funcionais, banco de registradores e suas interconexões) como um grafo base e a representação do programa de entrada como um *Directed Acyclic Graph* (DAG) com vértices representando instruções, operandos de entradas e saída, e as arestas as dependências entre esses vértices. O DAG de entrada possui atributos especiais para indicar a latência de instruções, operandos especiais (registradores de instruções específicas/dedicadas) e dependências de controle. As entradas para o algoritmo são compostas por um DAG G_1 e um grafo base G_2 que representa a arquitetura alvo. A saída é um grafo G'_2 isomórfico para G_1. Outra contribuição é a definição de grafo base como uma ferramenta para modelar diferentes modelos arquiteturais de processadores. A técnica tem-se mostrado particularmente viável para arquiteturas com restrições de registradores e interconexões. No entanto, pode-se vislumbrar extensões para arquiteturas *Very Long Instruction Word* (VLIW), máquinas escalares e até mesmo processadores que exploram paralelismo em nível de instrução utilizando outros modelos de execução. Experimentos foram realizados visando a validação, caracterização do algoritmo e a comparação com outras técnicas existentes na infraestrutura de compilação LLVM. Os resultados mostram que o algoritmo proposto, apesar de necessitar melhorias quanto ao tempo de compilação, pode obter ganhos de desempenho no código final gerado.

Abstract

This work aims at providing an integrated Instruction Scheduling and Register Allocation algorithm based on Subgraph Isomorphism theory implemented on the LLVM Compiler. The main contribution of this work is the mapping of instruction scheduling and register allocation as a Subgraph Isomorphism problem. The approach is based on the modelling of the hardware resources (functional units, register files and their interconnections) as a base graph, and the representation of the input program as Directed Acyclic Graphs (DAG) with vertices representing program instructions, input and output operands, and edges representing the dependences between two vertices. This input DAG has also special attributes to indicate the instruction latency, special operands (specific/dedicated instruction registers), and control dependencies. The algorithm input comprises a DAG G_1 of the program and a base graph G_2 that represents the target architecture. The output is a graph G'_2 isomorphic to G_1. Another contribution is the definition of base graphs as a tool to model different computer architectures to be used by the technique. The technique has shown to be a viable alternative for constrained register and interconnection processor architectures. In addition, the technique can be also extensible for architectures ranging from Very Long Instruction Word (VLIW), scalar, and even processors exploiting instruction level parallelism by using non-conventional execution models. Experiments have been performed to validate and characterize the algorithm. Other experiments have been performed to compare our proposal to classical techniques existing in the LLVM compiler. The results show that proposed algorithm, in spite being a subject for future investigations regarding compilation time, can provide performance gains for programs.

Sumário

Lista de Figuras **7**

Lista de Tabelas **8**

Lista de Algoritmos **9**

Lista de Acrônimos **10**

1 Introdução **11**

2 Fundamentos Teóricos e Trabalhos Relacionados **13**

2.1 Escalonamento de Instruções e Alocação de Registradores 13

2.2 Integração de Escalonamento e Alocação de Registradores 17

2.2.1 Algoritmo *Integrated Prepass Scheduling* e *DAG-Driven Register Allocation* 18

2.2.2 Algoritmo *Parallelizable Interference Graph* 19

2.2.3 Algoritmo *Combined Register Allocation and Instruction Scheduling Problem* 20

2.2.4 Algoritmo *Unified Resource Allocation Using Reuse DAGs and Splitting* 20

2.2.5 Algoritmo *Register Allocation Sensitive Region* 21

2.3 Considerações Finais 22

3 Algoritmo de Escalonamento de Instruções e Alocação de Registradores **23**

3.1 Definições Básicas 23

3.2 Algoritmo de Escalonamento de Instruções Baseado em Isomorfismo de Subgrafos 25

3.3 Algoritmo para Escalonamento de Instruções e Alocação de Registradores Baseado em Isomorfismo de Subgrafos 29

3.4 Exemplos de Processadores Modelados como Grafos Base 31
3.4.1 Grafo Base para o Processador MIPS 32
3.4.2 Grafo Base para o Processador HPL-PD 34
3.4.3 Grafo Base para o Processador ρ-VEX 35
3.5 Considerações Finais 36

4 Infraestrutura de Compilação LLVM 38
4.1 Compilador LLVM 38
4.2 Escalonamento de Instruções no LLVM 41
4.3 Alocação de Registradores no LLVM 42
4.4 Integração do Algoritmo de Escalonamento e Alocação de Registradores Baseado em Isomorfismo de Subgrafos no LLVM 43
4.5 Considerações Finais 45

5 Experimentos e Resultados 47
5.1 Análise de Desempenho do Algoritmo Proposto 47
5.2 Análise dos Impactos do Algoritmo de Isomorfismo sobre o Desempenho dos Programas 50
5.3 Considerações Finais 51

6 Conclusões e Trabalhos Futuros 54
6.1 Dificuldades Encontradas 55
6.2 Propostas para Trabalhos Futuros 55

Referências Bibliográficas 57

Lista de Figuras

2.1 Fases do algoritmo de coloração de vértices. 15

2.2 Exemplo de alocação de registradores. 16

3.1 Exemplo do problema de isomorfismo de subgrafos. 24

3.2 Grafo de entrada com duas instruções. 24

3.3 Grafo base da arquitetura com duas unidades funcionais e quatro registradores. 25

3.4 Grafo base da arquitetura desenrolado duas vezes. 28

3.5 Exemplo de desenrolamento do grafo de entrada. 29

3.6 Exemplo de grafo de entrada e grafo base com a modelagem de registradores e o grafo resultante $G_2^{'}$ do isomorfismo entre G_1 e G_2. 31

3.7 Grafo base da arquitetura MIPS representado com 4 registradores. 34

3.8 Exemplo de Isomorfismo de Subgrafos na arquitetura MIPS. 34

3.9 Exemplo de grafo base do HPL-PD. 35

3.10 Grafo base do processador HPL-PD desenrolado. 35

3.11 Exemplo de grafo base do ρ-VEX. 36

3.12 Exemplo de desenrolamento do grafo base do ρ-VEX. 36

4.1 Visão em alto nível da infraestrutura LLVM. 39

4.2 Exemplo baseado em SSA. 39

4.3 Exemplo de código em C e em LLVM IR. 39

4.4 Fases do LLVM na geração de código. 40

4.5 Exemplo do SelectionDAG (fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z). . . . 41

4.6 Fases do LLVM na geração de código com acréscimo do algoritmo de isomorfismo de subgrafos. 43

Lista de Tabelas

2.1 Comparação de *speedup* entre os algoritmos IPS e URSA. 21

5.1 Tempo médio de compilação, em segundos, dos programas utilizando escalonamento de instruções *List Scheduling* e alocação de registradores com o algoritmo *Greedy* e *Linearscan*. 48

5.2 Tempo médio de compilação, em segundos, dos programas utilizando escalonamento de instruções e alocação de registradores com o algoritmo de isomorfismo de subgrafos. 48

5.3 Quantidade de *spills* gerados com 16 registradores disponíveis na máquina MIPS. 50

5.4 Tradução das instruções *movn* e *movz*. 51

5.5 Resultados da execução dos programas compilados com os algoritmos isomorfismo, *greedy* e *linearscan*, com o simulador do processador MIPS. 52

Lista de Algoritmos

1 Algoritmo para exemplo de alocação de registradores. 17

2 Algoritmo para exemplo de alocação de registradores com *spill code*. 17

3 Algoritmo para exemplo de alocação de registradores com registradores físicos. 17

4 Algoritmo de escalonamento de instruções 2D-VLIW baseado em isomorfismo de subgrafos [1]. 26

5 Algoritmo de *match* entre o grafo do programa e o grafo base da arquitetura. 27

6 Algoritmo para construção do grafo base. 28

7 Algoritmo de escalonamento de instruções e alocação de registradores baseado em isomorfismo de subgrafos. 30

8 Algoritmo para construção do grafo do programa. 32

9 Algoritmo para construção do grafo base com unidades funcionais e registradores. 33

10 Algoritmo de runOnFunctionMachine método herdado da classe *Pass*. 44

11 Algoritmo que verifica a compatibilidade de um vértice do grafo do programa com um vértice do grafo base. 45

12 Algoritmo de visitor. 46

Lista de Acrônimos

ABI *Application Binary Interface*.

BSD Berkeley Software Distribution.

DAG Grafo Dirigido Acíclico.

Hexagon DSP *Qualcomm's VLIW processor*.

HPL-PD Hewlett-Packard Laboratories PlayDoh.

MIPS Microprocessor without Interlocked Pipeline Stages.

RISC *Reduced Instruction Set Computer*.

SIGPLAN *Special Interest Group on Programming Languages*.

SSA *Static Single Assignment*.

UF Unidade Funcional.

UIUC *University of Illinois at Urbana-Champaign*.

ρ-VEX *Reconfigurable and Extensible VLIW Processor*.

VLIW Very Long Instruction Word.

Capítulo 1

Introdução

O escalonamento de instruções e a alocação de registradores são atividades que impactam diretamente no desempenho final de uma aplicação, uma vez que determinam precisamente como o *hardware* será utilizado. Tais atividades devem ser executadas levando em consideração os recursos físicos existentes no processador alvo.

Desde a década de 70, a comunidade da área de compiladores tem dedicado esforço considerável em busca de soluções para os problemas de escalonamento e alocação de recursos em processadores [2–4]. Muito desse esforço se deve ao fato que tais problemas pertencem à classe NP-completo [5, 6]. Com isso, surge a necessidade de pesquisas em busca de técnicas e melhores heurísticas [7–12]. Aliado a isso, o surgimento de processadores com múltiplos núcleos (Multicores) e processadores baseados no modelo de execução *Single Instruction Multiple Threads* (SIMT) aumenta a complexidade dessas atividades, pois há de se considerar não apenas os múltiplos recursos, mas também a maneira como estão interconectados e suas restrições de utilização [1, 12–20].

Nesse cenário, este trabalho estende uma técnica definida em [1] para escalonamento de instruções baseada em isomorfismo de subgrafos para geração de código exclusiva para um processador experimental com modelo de execução VLIW. Nessa técnica, dado um grafo G_1 representando o fluxo de dados de um programa e um grafo G_2 representando as unidades funcionais/elementos de processamento de um processador e suas interconexões, o objetivo é encontrar um subgrafo G'_2 de G_2 que seja isomorfo a G_1. O grafo resultante G'_2 possui as unidades funcionais/elementos de processamento que precisam ser escalonados para as instruções de G_1 e em que ordem esse escalonamento precisa ser feito.

Diante dessa situação, o objetivo principal deste trabalho de mestrado é estender o algoritmo definido em [1] para tratar os problemas de escalonamento de instruções e alocação de registradores de maneira integrada. Além disso, projetar a implementação deste novo algoritmo sobre uma infraestrutura robusta de compilação como *Low-Level Virtual Machine* (LLVM) [21]. Entende-se que o algoritmo aqui abordado não se constitui apenas numa extensão da proposta apresentada em [1]. De outra forma, é uma nova alternativa para lidar com ambos os problemas de forma integrada e, ainda, lidar com situações como *spill code*. Neste novo algoritmo, o grafo G'_2 representa não apenas as unidades funcionais para execução das instruções mas, também, os registradores necessários para possibilitar a execução eficiente dessas instruções.

Ressalta-se ainda que esta abordagem se constitui numa alternativa inovadora para resolução de ambos os problemas nas etapas de geração de código de um compilador. Tal resolução ocorre em um único passo, de tal forma que, por meio de uma função de mapeamento das instruções do programa sobre os recursos arquiteturais do processador, tem-se, ao final, um código gerado já escalonado e com os recursos de armazenamento (registradores) já efetivamente reservados para a computação. Assim, pode-se organizar as principais contribuições deste trabalho como:

- Proposta de um novo algoritmo de escalonamento e alocação de registradores baseado em isomorfismo de subgrafos;
- Extensão da representação de grafo base para representar recursos de execução (unidades funcionais/de processamento e registradores) para diferentes arquiteturas de processadores e modelos de execução;
- Projeto e implementação do novo algoritmo junto a uma infraestrutura de compilação amplamente utilizada pela comunidade acadêmica e pela indústria;
- Validação, caracterização e comparação do algoritmo proposto com abordagens clássicas existentes na literatura da área.

O texto dessa desta dissertação está organizado da seguinte forma:

- no Capítulo 2 apresenta-se a fundamentação teórica dos conceitos mais relevantes usados ao longo do texto, assim como uma revisão detalhada da literatura. Referências bibliográficas serão fornecidas para o leitor interessado em conhecer mais detalhes.
- no Capítulo 3 será apresentado o algoritmo para o isomorfismo em uma visão em alto nível. Também será apresentado outros exemplos de grafo base de diversas arquiteturas.
- O Capítulo 4 descreve o compilador LLVM e a integração da técnica com a geração de código para a arquitetura *Microprocessor without Interlocked Pipeline Stages* (MIPS).
- O Capítulo 5 apresenta e discuti todos os experimentos realizados e resultados obtidos visando à validação e avaliação da técnica.
- As conclusões deste trabalho assim como proposições para desenvolvimentos de trabalhos futuros são apresentadas no Capítulo 6.

Capítulo 2

Fundamentos Teóricos e Trabalhos Relacionados

O escalonamento de instruções e a alocação de registradores são atividades que impactam diretamente no desempenho final de uma aplicação, uma vez que determinam precisamente como o hardware será utilizado. Tais atividades devem ser executadas levando em consideração os recursos físicos existentes no processador alvo.

Este capítulo apresenta as definições básicas para a resolução dos problemas de escalonamento de instruções e alocação de registradores na Seção 2.1. Além disso, apresenta também outros trabalhos na literatura da área que propõem técnicas para resolução de ambos os problemas de maneira integrada na Seção 2.2.

2.1 Escalonamento de Instruções e Alocação de Registradores

A área de compiladores possui uma vasta quantidade de trabalhos cujo foco reside na resolução dos problemas de escalonamento de instruções e alocação de registradores de maneira integrada. Uma vez que há vantagens e desvantagens da ordem de utilização do escalonamento de instruções em relação à alocação de registradores, faz-se necessário estudar detalhadamente o assunto a fim de que a integração entre essas duas fases possa ser melhor aproveitada em um compilador. Os trabalhos de [3, 12, 18, 19] apresentam propostas de algoritmos para resolução desses problemas. Nesta Seção, apresenta-se os conceitos fundamentais envolvidos nos problemas de escalonamento e alocação assim como algumas estratégias de resolução.

O escalonamento de instruções consiste em atribuir operações do programa a recursos físicos (elementos de processamento, unidades funcionais), aptos a executá-los de acordo com uma ordem específica. O problema de escalonar instruções sobre recursos de hardware é um problema amplamente estudado na área de compiladores [22–24]. Em sua forma original, o escalonamento de instruções consiste em definir uma ordem de execução para as instruções de um programa. Essa ordenação é realizada pelas instruções que compõem cada bloco básico do programa, respeitando, necessariamente, as dependências de dados

entre essas instruções. Um algoritmo clássico, baseando-se nessa ordenação, para realizar o escalonamento é denominado *list scheduling* [5]. Historicamente, a medida que máquinas que oferecem paralelismo em nível de instrução (do inglês ILP – *Instruction Level Parallelism*) foram surgindo, essa fase passou a ser adotada como uma otimização, uma vez que a ordenação das instruções pode considerar a minimização de conflitos de dados, de controle e estrutural. A minimização desses conflitos reduz o número de *stalls* gerados pelo processador e, como consequência, impacta diretamente no desempenho do programa.

A alocação de registradores é uma tarefa importante na otimização de geração de código que afeta diretamente o desempenho do programa gerado. Há um vasto número de trabalhos propondo soluções para o problema de alocação de registradores [2–4, 6]. O objetivo desta tarefa é mapear os registradores virtuais (utilizados pelas instruções) nos registradores físicos da máquina alvo. Caso não hajam registradores disponíveis suficientes, há necessidade de armazenar um dos registradores virtuais na memória. Esse evento é denominado *spill* [25]. De maneira geral, o *spill* é resolvido por meio da adição de instruções para escrever e ler um valor (que não poderia ser atribuído para um registrador) da memória.

Em virtude da complexidade inerente à atividade de alocação de registradores, apresenta-se a seguir definições básicas que serão usadas pelas técnicas e algoritmos descritos na sequência.

Definição 1 *Alocação de Registradores é a otimização que faz o mapeamento dos registradores virtuais em registradores físicos da arquitetura alvo.*

Definição 2 *A* live range *de uma variável inicia no ponto onde a variável é definida e termina no ponto onde a variável é redefinida ou em seu último uso [5].*

Definição 3 *Bloco Básico é uma sequência de instruções, tal que a execução flui, sem desvios, da primeira até a última instrução do Bloco Básico [26].*

Definição 4 *Grafo de interferência é um grafo* $G = (V, E)$*, em que os vértices de* V *representam as* live ranges *existentes no código analisado e as arestas de* E *representam a interseção dessas* live ranges*.*

A alocação de registradores pode ser realizada utilizando o algoritmo de *Coloração de Vértices* [5]. Essa coloração é feita a partir do grafo de interferência. O algoritmo procura colorir o grafo G com um número de cores K (representa o número de registradores físicos) de maneira que nenhum par de vértices conectados por uma aresta tenha a mesma cor. Se uma K-coloração não for encontrada então ocorre o *spill*. *Spill* gera novas instruções e um novo grafo de interferência. O pressuposto para a K-coloração de um grafo G, é que qualquer vértice de G deve possuir menos que K vizinhos. Suponha que G tem menos do que K vizinhos, remova o vértice i e seus vizinhos de G obtendo um subgrafo G'. Uma K-coloração de G' pode ser estendida para uma K-coloração de G atribuindo ao vértice i uma cor que não foi atribuída a nenhum de seus vizinhos [5].

O algoritmo que realiza uma K-coloração tem as seguintes fases, representadas na Figura 2.1:

- **renumber**: descobre as *live ranges* (variáveis com interferência)

- ***build***: constrói o grafo de interferência
- ***coalescing***: une *live ranges* não interferentes
- ***spill costs***: calcula o custo do *spill*
- ***simplify***: método de coloração de vértices
- ***spill code***: insere novas instruções por causa do *spill* (*load* e *store*)
- ***select***: mapeia os registradores virtuais nos registradores físicos

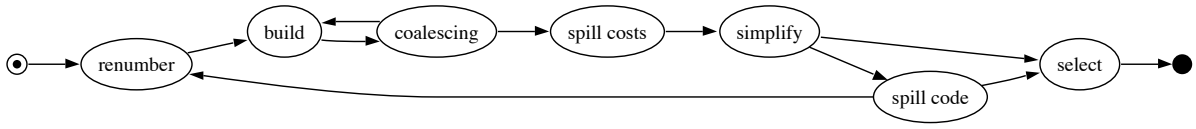


Figura 2.1: Fases do algoritmo de coloração de vértices.

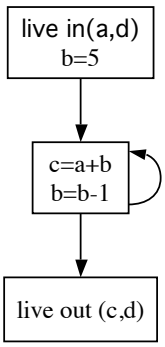
Deve-se destacar que a fase *coalescing* [5,26] pode minimizar o *spill*. O *coalescing* consiste na ação de unir duas variáveis, não interferentes, que são relacionadas por uma cópia de instrução para o mesmo registrador.

Destaca-se também a possibilidade de utilizar uma nova fase denominada *freeze*: se não puder remover vértices por *simplify* nem por *coalescing*, então o passo *freeze* deve marcar um vértice para não ser considerado pelo *coalescing* [26].

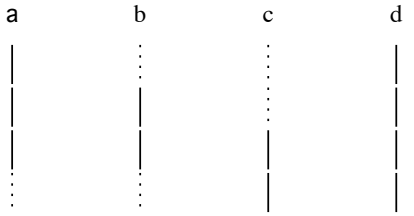
Definição 5 *Dada uma operação do tipo* $a := b$ *(*move*) em que as* live ranges *de* a *e* b *são não conflitantes, então, os vértices que representam* a *e* b *no grafo de interferência podem ser unidos e a operação pode ser eliminada. Essa união de vértices é denominada* coalescing.

Definição 6 Spill code *é a inserção de instruções* load *e* store *no programa devido a falta de registradores. A necessidade de* spill code *é detectada na fase* simplify*, pois não há remoção de vértices do grafo de interferência [26].*

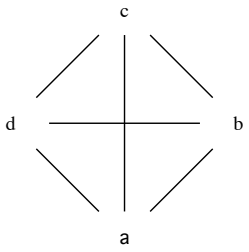
A Figura 2.2 apresenta um exemplo de utilização do algoritmo de alocação de registradores através da coloração de vértices utilizando 3 registradores ($K = 3$). Dado o trecho de código do Algoritmo 1, tem-se a organização em blocos básicos na Figura 2.2a, as *live ranges* das variáveis a, b, c e d na Figura 2.2b (fase *renumber*) e o grafo de interferência na Figura 2.2c (fase *build*). As fases *coalescing e simplify* não podem ser executadas, então uma variável é selecionada para *spill*, fase *spill cost*. A variável d foi escolhida para ser *spilled*. Na Figura 2.2d, continuando a alocação, só restam três variáveis, cada uma pode ser colorida usando três cores. Seguindo a alocação, a variável d realmente não pode voltar ao grafo e ser colorida, então na fase *spill code* o código é transformado (trecho de código do Algoritmo 2), de forma que todo o processo de alocação precisa ser realizado novamente (*renumber*). O novo grafo de interferência é representado na Figura 2.2e, que agora contém $d1$ e $d2$, ao invés de d. A fase *simplify* primeiro remove a variável $d1$, como na Figura 2.2f, e depois $d2$, b, a e c empilhando-as. Na Figura 2.2g, a fase *select* desempilha as variáveis c, a, b e $d2$ atribuindo cores, ou seja, mapeando para os registradores físicos (no exemplo $r1$, $r2$ e $r3$) de maneira que nenhum par de vértices conectados por uma aresta tenha o mesmo registrador. Por último, desempilha a variável $d1$, atribuindo o registrador $r1$ como na Figura 2.2h. O código depois da alocação é representado no Algoritmo 3.



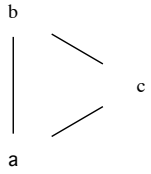
(a) Blocos básicos.



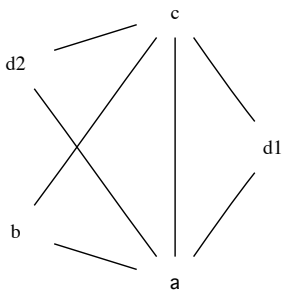
(b) *Live range*.



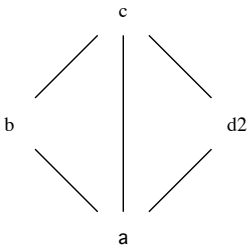
(c) Grafo de Interferência.



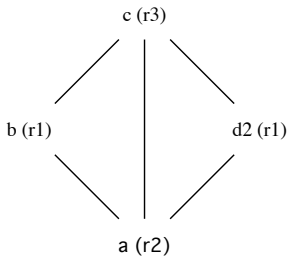
(d) *Spill* na variável d.



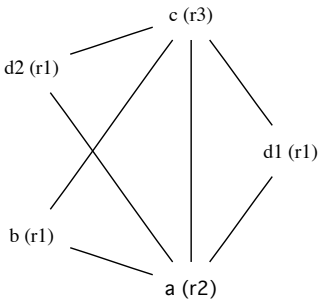
(e) Constrói o Grafo de Interferência depois do *Spill code*.



(f) *Simplify* da variável $d1$.



(g) *Select* nas variáveis a, b, c e $d2$.



(h) *Select* na variável $d1$.

Figura 2.2: Exemplo de alocação de registradores.

Algoritmo 1: Algoritmo para exemplo de alocação de registradores.

Entrada: live in (a,d)
Saída: live out (c,d)

```
1 início
2 |  b=5
3 |  loop: c=b+a
4 |  b=b-1
5 |  if b ≤ 0 goto loop
6 fim
```

Algoritmo 2: Algoritmo para exemplo de alocação de registradores com *spill code*.

Entrada: live in (a,d1)
Saída: live out (c,d2)

```
1 início
2 |  M[Dloc] = d1
3 |  b=5
4 |  loop: c=b+a
5 |  b=b-1
6 |  if b ≤ 0 goto loop
7 |  d2=M[Dloc]
8 fim
```

Algoritmo 3: Algoritmo para exemplo de alocação de registradores com registradores físicos.

Entrada: live in (r2,r1)
Saída: live out (r3,r1)

```
1 início
2 |  M[Dloc] = r1
3 |  r1=5
4 |  loop: r3=r1+r2
5 |  r1=r1-1
6 |  if r1 ≤ 0 goto loop
7 |  r1=M[Dloc]
8 fim
```

2.2 Integração de Escalonamento e Alocação de Registradores

A literatura da área de Compiladores dispõe de várias propostas de algoritmos para escalonamento de instrução e alocação de registradores de forma separada ou conjunta. Uma das principais motivações para desenvolver novos algoritmos que tratam ambos os problemas de maneira integrada é a possibilidade de maximizar o desempenho do programa, pois pode-se invocar a resolução de um problema a partir de situações existentes no outro. Também, a medida que processadores mais avançados (com maior disponibilidade de recursos) estão

disponíveis para utilização, a necessidade para gerar código eficiente para essas máquinas se torna ainda mais importante. Algumas propostas de algoritmos integrados são apresentadas a seguir.

O escalonamento pode ser aplicado a uma linguagem intermediária tanto antes da alocação de registradores (*prepass*) quanto depois (*postpass*). A vantagem do *prepass* é a exploração do paralelismo e, por outro lado, implica no aumento da *live range* das variáveis possibilitando o aumento de *spill code*. *Postpass* não aumenta o *spill code*, pois a alocação já aconteceu, entretanto faz o escalonamento mais restrito.

2.2.1 Algoritmo *Integrated Prepass Scheduling* e *DAG-Driven Register Allocation*

Em [2] apresenta-se as diferenças de cooperação entre escalonamento local e global com alocação local e global. Local é o rearranjo das instruções dentro de um bloco básico isolado do resto do programa. O global considera o movimento de instruções entre os blocos. Dois métodos (*Integrated Prepass Scheduling* — integrado no *prepass* — e *DAG-Driven Register Allocation* — integrado no *postpass*) foram propostos como solução para escalonamento e alocação de uma forma cooperativa.

No método *Integrated Prepass Scheduling* foram combinadas duas técnicas de escalonamento: uma para reduzir o atraso do paralelismo (CSP) — *Code Scheduling for Pipelined processors* — e outra para minimizar o uso de registradores (CSR) — *Code Scheduling to minimize Registers usage*. A ideia básica é manter um número de registradores disponíveis durante o escalonamento. Quando há registradores suficientes, o escalonador usa CSP para reduzir o atraso do paralelismo. Quando o número de registradores disponíveis alcança um limite inferior, o escalonador troca para o CSR para controlar o uso de registradores. As simulações foram realizadas variando a quantidade de registradores de 4 a 30 e em uma máquina com alto paralelismo (considera 11 ciclos de *clock* — CC — para *load*, 3 CC para adição de inteiros, 6 CC para para multiplicação de inteiros) e médio paralelismo (similar a máquina com alto paralelismo exceto pelo *clock rate* menor — considera 6 CC para *load*, 2 CC para adição e 3 CC para multiplicação). Como medida de comparação é utilizado o número de ciclos para executar o programa. Foram utilizados os 20 primeiros programas do *benchmark Livermore loops*. Com 4 registradores foi observada maior diferença de desempenho (CSR obtém melhor desempenho). A medida que registradores são incrementados, essa diferença diminui. O número de ciclos é o mesmo com 30 registradores.

A técnica *DAG-Driven* utiliza um grafo de dependência. Dois termos são introduzidos: largura e altura. A largura é definida pelo máximo de vértices mutuamente independentes que precisam de um registrador de destino. A altura do DAG é o comprimento do seu caminho mais longo. Se o número de registradores é maior que a largura do DAG, a geometria permanece inalterada durante a alocação de registradores. Caso contrário, a alocação reduzirá a largura do DAG para um número menor ou igual ao número de registradores. Enquanto a largura é reduzida, a altura é aumentada. Quanto maior a altura, maior é o caminho crítico. Quanto maior o caminho crítico, menos eficiente é o código do escalonamento. As estratégias para controlar o crescimento da altura são: exploração das dependências *Write-After-Read* (WAR) e o balanceamento de crescimento do DAG. O *Benchmark* utilizado é o *Livermore*

loops (utilizou-se os doze primeiros programas). A medida utilizada é o número de ciclos do *clock* necessários para executar o programa. A quantidade de registradores variou de 4 a 30. *Prepass* tem um número menor de ciclos de *clock* comparado com o *postpass*, mas gera um programa com mais instruções.

Ambos algoritmos se mostram efetivos em resolver o problema de interdependência entre o escalonamento e a alocação de registradores. Em um modelo de alto paralelismo o *Integrated Scheduling* gera programas com melhor desempenho. O *DAG-driven allocator* apresenta-se como melhor opção em um modelo de médio paralelismo. O *Integrated Scheduling* executa o escalonamento com um número limitado de registradores e oscila entre a utilização das heurísticas CSP e CSR. Se há registradores suficientes então dá preferência para instruções que aumentam o paralelismo, senão opta por instruções que diminuem o número de variáveis utilizadas no mesmo instante. No *DAG-driven allocator* a alocação local é executada e usa a informação de dependências de dados do DAG enquanto o escalonamento é restringido pelas dependências adicionadas pela alocação.

2.2.2 Algoritmo *Parallelizable Interference Graph*

Em [9] é apresentada uma abordagem baseada na construção de um grafo de interferência paralelo (*parallelizable interference graph*). As definições a seguir são necessárias para o entendimento do processo de construção do grafo de interferência paralelo. Nesse trabalho, considera-se G_r o grafo de interferência (conforme Definição 4).

Definição 7 *Seja $G_s = (V_s, E_s)$ o grafo de escalonamento. Cada vértice $v \in V_s$ corresponde a uma instrução do código intermediário. Há uma aresta direcionada $(u, v) \in E_s$, de u para v se u deve ser executado antes de v.*

Em seguida é gerado um grafo contendo todas as falsas dependências (G_f). As arestas do grafo serão usadas na geração do grafo de interferência paralelo.

Definição 8 *Seja $G_f = (V_f, E_f)$. $V_f = V_s$. Definido o conjunto E_t contendo exatamente restrições da máquina no escalonamento. O par $(u, v) \in E_f$ é dado por $u, v \in V_f, u \neq v$ e $(u, v) \notin E_t$.*

Com o grafo de falsas dependências é definido o grafo de interferência paralelo. Basicamente, esse inclui ambas arestas do grafo de falsa dependência e aquelas do grafo de interferência.

Definição 9 *Seja $G = (V, E)$ o grafo de interferência paralelo. $V = V_r$. $E = E_r \cup \{\{u, v\} : \{u, v\} \in E_f$ e $u, v \in V\}$.*

O escalonamento varre o grafo pelo incremento do número EP (número que representa o possível tempo para escalonar o vértice) do vértice. Quando as operações com o mesmo número EP não podem ser escalonadas juntas então elas são postergadas e o número EP é atualizado. O procedimento de coloração é realizado primeiro gerando o grafo de interferência paralelo depois segue com a coloração exemplificada a seguir (considere r o número de registradores físicos da máquina):

1. Constrói-se o grafo de interferência paralelo G_s.
2. Executa-se o procedimento de coloração:
   (a) *Simplify*: remove v e atualiza G_s se há um vértice $v \in V_s$ com grau menor do que r. Usa considerações do escalonamento para remover de G_s e $G_s^{'}$ uma aresta de falsa dependência não em E_r.
   (b) *Spill* Cost: se V_s não estiver vazio escolhe um vértice $v \in V_s$ com menor custo e coloca v na lista de *spill*.
   (c) *Select*: colore o grafo se não há *spill*.
   (d) *Spill*: para cada item na lista de *spill* refazer o procedimento de coloração.

Essa solução foi avaliada utilizando os programas do *SPEC92 benchmark*. Os resultados da comparação são dados pelo tempo de execução. A melhora mais significativa é do programa 023.*eqntott* de 0.33% e a menos significativa é do programa 050.*ear* de 0.05%.

2.2.3 Algoritmo *Combined Register Allocation and Instruction Scheduling Problem*

Em [7] é apresentada a técnica *Combined Register Allocation and Instruction Scheduling Problem* (CRISP) cujo objetivo é de minimizar o *spill* sem perder o paralelismo em nível de instrução. O algoritmo combina o escalonamento e a alocação de registradores dentro de um bloco básico como um único problema de otimização.

A solução do problema é encontrar um custo mínimo C onde um escalonamento factível T é somado ao dobro do número de elementos do conjunto de *spills* SVR, dado por: $C = T + 2|SVR|$. Estabelece-se o algoritmo (α, β)-*Combined Heuristic*, em que os parâmetros $\alpha, \beta \in [0, 1] \implies \alpha + \beta = 1$ provêem um peso relativo para controlar a pressão nos registradores γ_r e considerar o paralelismo de instruções γ_s, dado pela função de classificação $\gamma = \alpha\gamma_s + \beta\gamma_r$. Com essa classificação é realizada a ordenação das instruções em uma lista. Então executa o algoritmo *greedy list scheduling*. Observa-se um desempenho de melhora de 16%-21% em relação a execução de escalonamento seguido de alocação. Já para uma alocação seguida de um escalonamento o desempenho é de 4%-21%.

2.2.4 Algoritmo *Unified Resource Allocation Using Reuse DAGs and Splitting*

Em [11] é apresentado o algoritmo *Unified Resource Allocation using Reuse Dags and Splitting* baseado no paradigma medir e reduzir *(measure-and-reduce)* tanto de registradores quanto de unidades funcionais. A técnica constrói DAGs de reúso (*reuse DAGs*) de recursos. O DAG é composto por diversas cadeias de reúso. Cada cadeia representa um conjunto de instruções que podem compartilhar o mesmo recurso. Usando DAGs de reúso, essa abordagem identifica grupos de instruções nas quais o paralelismo requer mais recursos do que há disponível. Então, reduções são executadas para a demanda excessiva. Se um número de cadeias excede o limite de registradores, é adicionado sequências de arestas ou código de

spill no DAG de reúso para reduzir a pressão e, como consequência, reduzir o tamanho do DAG de reúso. O objetivo é modificar o DAG identificando regiões de código que precisam de reduções sobre os recursos requeridos para os níveis suportados pela arquitetura. A técnica é descrita, basicamente em três passos:

1. Representação do programa e requisitos para cada recurso.
2. Algoritmos para mensurar os requisitos dos recursos.
3. Transformações para reduzir os recursos requeridos para os níveis suportados pela arquitetura.

Uma extensão dessa técnica é proposta em [4], baseando-se na técnica *IPS* apresentada em [2]. Essa proposta utiliza o DAG de reúso e *Live range splitting* para dividir a *live range*, reduzindo assim a demanda de registradores. O algoritmo mantém uma lista de todos os registradores sendo escalonados. Quando uma alta pressão nos registradores é detectada e não há instruções prontas para escalonar, então é executado a divisão da *live range*, que reduz o número de *live ranges* ativas. O *live range splitting* é descrito a seguir:

1. Insere instrução *load* na lista de escalonamento pronta.
2. Insere instrução *store* na lista de escalonamento não pronta.
3. Move as dependências de usos, não escalonadas, da definição original para o *load* inserido.

Os experimentos, utilizando *Testsuite Speedup*, foram mensurados pelo *speedup* com registradores variando de 8, 16 a 32. Comparando com IPS de [2] essa abordagem teve melhor desempenho, conforme a Tabela 2.1.

Tabela 2.1: Comparação de *speedup* entre os algoritmos IPS e URSA.

Registradores	IPS	URSA
8	1,25	4,25
16	1,75	3,25
32	2	2,25

2.2.5 Algoritmo *Register Allocation Sensitive Region*

Em [8] propõe-se uma técnica de escalonamento e alocação chamada de *Register Allocation Sensitive Region* (RASER). Esta técnica atua sobre um *Program Dependence Graph – Grafo de Dependência do Programa* (PDG) [10][1] e tenta criar regiões de código com uma quantidade

[1]Um PDG é um grafo direcionado que representa as dependências de controle e as dependências de dados entre as sentenças do programa. Os vértices são sentenças e predicados de expressões que ocorrem em um programa. Uma aresta representa uma dependência de controle ou de dados

igual de paralelismo. O objetivo é combinar o paralelismo estimado em cada região com a quantidade de paralelismo explorável pela arquitetura alvo.

A técnica é baseada no seguinte fluxo de execução:

1. Obtenção de estimativa do paralelismo disponível em cada região do PDG.
2. Execução do *Integrated Prepass Scheduling* (IPS) [2] em cada região.
3. Construção de uma lista de regiões em que o número de variáveis vivas excede o número de registradores físicos.
4. Redução dos *spills* por meio da redução da quantidade de variáveis vivas nestas regiões.
5. Aplicação de transformações para aumentar o paralelismo nas regiões.
6. Eliminação de regiões sem descendentes.

O desempenho do RASER é comparado com o desempenho de um escalonamento de regiões padrão (blocos básicos) seguido pela alocação de registradores. Os experimentos são executados em uma máquina hipotética com médio paralelismo e com unidades funcionais em paralelo. Um simulador é usado para determinar o número de ciclos necessários para executar o código intermediário e o número de *loads* e *stores* executados devido ao *spilling*. Utiliza-se programas do *benchmark Livermore loops*, variando o número de registradores de 3 a 16. No estudo é indicado que para um conjunto pequeno de registradores, em geral, RASER gera código que executa mais rápido, em 17 de 24 experimentos com 3 registradores e em 19 de 24 com 4 registradores. O número de ciclos executados é comparado com a alocação via coloração de vértices e a utilização de um escalonamento *postpass*.

2.3 Considerações Finais

Esse capítulo apresentou os principais conceitos teóricos utilizados ao longo do trabalho. Também apresentou-se uma revisão bibliográfica sobre trabalhos existentes na literatura da área com foco em técnicas integradas para resolução dos problemas de escalonamento de instruções e alocação de registradores. Interessante observar que a maior parte das propostas apresentadas utilizam DAGs organizados a partir de blocos básicos, realizando a integração localmente. A maior parte das propostas apresentam a técnica considerando uma arquitetura alvo e não consideram a extensão da técnica para outras arquiteturas. O próximo capítulo detalha a proposta de algoritmo integrado para escalonamento e alocação de registradores deste trabalho apresentando ainda como outras arquiteturas podem ser modeladas para utilizar o algoritmo proposto.

Capítulo 3

Algoritmo de Escalonamento de Instruções e Alocação de Registradores

Este capítulo apresenta em detalhes a proposta de um algoritmo para resolução dos problemas de escalonamento de instruções e alocação de registradores de maneira conjunta. O algoritmo aqui apresentado modela as instruções do programa e os recursos computacionais a serem utilizados como grafos. A tarefa do algoritmo é então mapear os vértices do grafo de instruções sobre os vértices do grafo dos recursos computacionais. Esse mapeamento é baseado na teoria de isomorfismo de subgrafos para a arquitetura MIPS. Em adição, demonstra-se também como é possível estender o algoritmo para gerar código para outras arquiteturas de processadores.

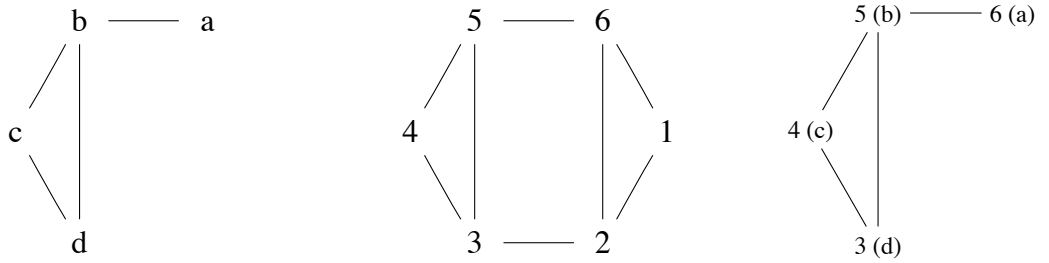
3.1 Definições Básicas

Para o entendimento adequado do algoritmo proposto neste trabalho, apresenta-se algumas definições teóricas envolvendo os principais conceitos utilizados pelo algoritmo.

Definição 10 *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 $\varphi : V_1 \to V_2$ 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 $(\varphi(v_i), \varphi(v_j)) \in E_2$. [1].*

Definição 11 *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 $G_1 \cong G_2^{'}$ [1].*

A Figura 3.1 exemplifica o isomorfismo de subgrafos. As Figuras 3.1a e 3.1b mostram exemplos dos grafos G_1 e grafo G_2, respectivamente. Um possível resultado do mapeamento das arestas e vértices do grafo G_1 sobre o grafo G_2 é o grafo resultante na Figura 3.1c. Observe que o grafo resultado é isomorfo ao grafo G_1 e é um subgrafo de G_2. De acordo com a Definição 11, $\varphi(a) = 6$, $\varphi(b) = 5$, $\varphi(c) = 4$, $\varphi(d) = 3$.



(a) Grafo G_1 (b) Grafo G_2 (c) Grafo resultante $G_2^{'}$

Figura 3.1: Exemplo do problema de isomorfismo de subgrafos.

Ao aplicar a resolução do problema de isomorfismo de subgrafos para tratar os problemas de escalonamento e alocação de registradores faz-se necessário adaptar a nomenclatura utilizada para o contexto de compiladores. Considerando a Figura 3.1, o grafo G_1 passa a ser tratado, no contexto de compiladores, como grafo de dependência de dados entre instruções, enquanto que o grafo G_2 é relacionado ao grafo que representa os recursos arquiteturais necessários para as etapas de escalonamento de instruções e alocação de registradores. O grafo $G_2^{'}$ é subgrafo de G_2 e isomorfo ao grafo G_1, de forma que representa os recursos de hardware para execução das instruções representadas em G_1. Neste trabalho, denomina-se o grafo G_1 como grafo de entrada para o problema de isomorfismo de subgrafos. O grafo G_2 é denominado grafo base.

Definição 12 *Um grafo de entrada $G_1 = (V_1, E_1)$ é um grafo que representa as dependências de dados entre as instruções de um programa. V_1 é o conjunto de vértices de G_1 e representa as instruções. E_1 é o conjunto de arestas de G_1 e representa os operandos e dependências de dados entre instruções.*

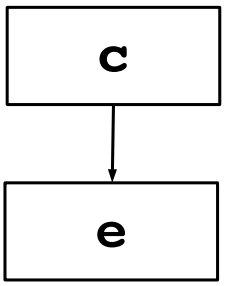


Figura 3.2: Grafo de entrada com duas instruções.

A Figura 3.2 apresenta um exemplo de um grafo de entrada com apenas duas instruções hipotéticas (**c** e **e**). Deve-se atentar que esse grafo representa o fluxo de dados entre as instruções **c** e **e**. Assim, nota-se que a instrução **c** deve ser a primeira instrução a ser executada uma vez que seu resultado é usado por **e**. Uma vez que os vértices representam instruções de um programa, cada vértice possui um atributo peso, que corresponde ao número de ciclos necessários para executar a instrução, e um atributo tipo que representa o tipo dessa instrução, de acordo com a especificação do conjunto de instruções ao qual pertence.

Definição 13 *O grafo base $G_2 = (V_2, E_2)$ é um grafo que representa a topologia das unidades funcionais/elementos de processamento e registradores de uma arquitetura de processador alvo. No grafo G_2, o conjunto de vértices V_2 representa os recursos de processamento/armazenamento do processador enquanto que o conjunto de arestas E_2 representa a interconexão entre os elementos de V_2.*

A Figura 3.3 apresenta um exemplo simplificado de um grafo base para uma arquitetura de processador escalar. Esse grafo representa um processador alvo com suporte a quatro registradores ($R1$, $R2$, $R3$, $R4$) e dois elementos de processamento/unidades funcionais ($FU1$, $FU2$). Os vértices do grafo base possuem um atributo denominado tipo que indica o(s) tipo(s) de vértices do grafo de entrada que pode(m) ser executada(s) e, consequentemente, mapeada sobre esse vértice. Além desse, há também o atributo peso que, para os vértices do grafo base, representam a latência mínima (em ciclos) necessária para executar um vértice do grafo de entrada. No momento do *matching* entre um vértice do grafo de entrada com um vértice do grafo base, esses dois atributos devem ser verificados e devem ser iguais em ambos os vértices para que o *matching* aconteça.

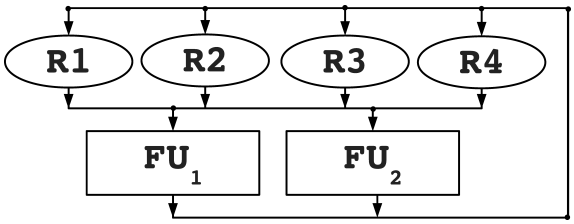


Figura 3.3: Grafo base da arquitetura com duas unidades funcionais e quatro registradores.

Na Figura 3.3 pode-se notar que tanto FU_1 quanto FU_2 podem ler operandos e escrever resultados em qualquer um dos quatro registradores ($R1$, $R2$, $R3$, $R4$) disponíveis, devido às arestas que interligam os registradores com as unidades funcionais (UFs) e das UFs para os registradores. Neste modelo está sendo considerado que os vértices que modelam as UFs são genéricos. Isso significa que qualquer vértice, representando uma operação, do grafo de entrada, pode ser mapeado sobre qualquer um desses vértices de UFs. Da mesma forma, considera-se que qualquer vértice operando, do grafo de entrada, pode ser mapeado sobre qualquer vértice registrador. Para direcionar o mapeamento de vértices operações sobre vértices UFs e de vértices operandos sobre vértices registradores, basta indicar, um valor específico para o atributo tipo de cada tipo de vértice (registrador ou UF). Esses mesmos valores deverão ser colocados como atributos dos vértices operandos e operações do grafo de entrada. Não está sendo considerado a possibilidade de uma UF ler operandos ou escrever dados da/para a memória. Se essa funcionalidade é permitida para uma UF, então deve existir um vértice, com atributo específico, para representar a memória e uma aresta da UF para a memória e vice-versa.

3.2 Algoritmo de Escalonamento de Instruções Baseado em Isomorfismo de Subgrafos

O trabalho de [1] aplica a teoria de isomorfismo de subgrafos para resolver o problema de escalonamento de instruções para um processador, denominado 2D-VLIW [27, 28], cuja

topologia de interconexão de elementos de processamento é a de uma matriz bidimensional. No algoritmo proposto nesse trabalho, um grafo de entrada é representado por um DAG. O algoritmo utiliza as classes da biblioteca VF [29, 30] que disponibiliza algoritmos para isomorfismo de grafos e subgrafos.

De forma geral, o algoritmo (Algoritmo 4) proposto em [1] recebe como entrada um DAG G_1 e um grafo base G_2, representando os recursos e interconexões do processador 2D-VLIW, e fornece como saída um código escalonado a partir do subgrafo $G_2^{'}$.

Algoritmo 4: Algoritmo de escalonamento de instruções 2D-VLIW baseado em isomorfismo de subgrafos [1].

```
Sched(DAG G1, GRAFO BASE G2)
    Entrada: DAG de Entrada G1 e grafo base G2.
    Saída: Conjunto de instruções já escalonadas.
1   topological_order(G1);
2   repeat
3       G2'=subg_iso_sched(G1,G2,&tag);
4       switch tag do
5           case 0: base_graph_resize(G2);
6           case 1: DAG_global_vertices(G1);
7       end
8   until G2' ≠ NULL;
9   create_2D-VLIW_instructions(G2');
```

No Algoritmo 4, a chamada *topological_order()* (linha 1) realiza a ordenação topológica sobre o DAG de entrada G_1. Na linha 3 invoca o procedimento para encontrar o subgrafo $G_2^{'}$ isomorfo ao grafo G_1. Esse procedimento executa o algoritmo de *match* (Algoritmo 5) implementado na biblioteca VF. Caso $G_2^{'}$ não seja determinado (linhas 2-8), o escalonador executa uma das heurísticas de otimização, tendo como base o valor da variável *tag*. Se o valor de *tag* é igual a 0, a heurística *base_graph_resize()* (linha 5) aumenta o tamanho do grafo base a fim de que o mesmo possa acomodar o isomorfismo. Uma vez que o grafo base representa os recursos existentes na arquitetura do processador para escalonamento e alocação, esse processo de desenrolar (redimensionar) o grafo base deve ser realizado de maneira a criar cópias interconectadas desses recursos. O trabalho de [31] discute detalhadamente o problema de aumentar o tamanho do grafo base por meio da modelagem como um problema de empacotamento de DAGs[1]. Por fim, a chamada à heurística *DAG_global_vertices()* (linha 6) altera as dependências de dados entre os vértices de G_1 para que esses possam ser lidos a partir de recursos globais (registradores globais ou hierarquia de memória) do processador alvo. Uma vez que o isomorfismo seja resolvido, instruções 2D-VLIW, provenientes do escalonamento realizado, são construídas (linha 9).

O Algoritmo 5 detalha o processo de *matching* entre os grafos G_1 e G_2 do procedimento anterior, a fim de obter o subgrafo $G_2^{'}$. Esse algoritmo é invocado pelo procedimento *subg_iso_sched()* do Algoritmo 4 e é um dos algoritmos da biblioteca VF. A entrada é

[1]Em [31], o problema de empacotamento de DAGs manipula o incremento do tamanho do grafo base como um "desenrolamento" de cópias desse grafo.

composta por um ponteiro para um grafo de estados (s) — que contém os grafos de entrada G_1 e G_2 — e um procedimento *visitor* (vis). Caso o *match* seja encontrado (linha 2), o Algoritmo 12 de visitor é executado. A cada par de vértices factível (linha 6) o grafo de estados s é incrementado (linha 7). Se o isomorfismo não é encontrado entre o par de vértices (linha 8), um procedimento de *backtracking* (linha 11) é invocado para que um novo par de vértices seja usado.

Algoritmo 5: Algoritmo de *match* entre o grafo do programa e o grafo base da arquitetura.

```
   match(State s, visitor vis)
   |  Entrada: State: grafo de estados que forma o G'_2, visitor: callback executado
   |           quando o match é encontrado
   |  Saída: true or false indicando se o grafo G'_2 é o resultado do match.
   |  /* Verdadeiro se o match entre os grafos G_1 e G_2 é encontrado.    */
1  |  if s->IsGoal() then
2  |  |  return vis(s);
3  |  end
   |  /* declara node_id n1 e n2 que é atribuído em NextPair              */
4  |  node_id n1, n2;
5  |  while s->NextPair(n1, n2) do
6  |  |  if s->IsFeasiblePair(n1, n2) then
7  |  |  |  s->AddPair(n1, n2);
8  |  |  |  if match(s, vis) then
9  |  |  |  |  return true;
10 |  |  |  else
11 |  |  |  |  s->BackTrack();
12 |  |  |  |  delete s;
13 |  |  |  end
14 |  |  end
15 |  end
16 |  return false;
```

A construção do grafo base para um algoritmo de escalonamento de instruções depende da quantidade, característica e organização das unidades funcionais disponíveis pelo processador. O Algoritmo 6 apresenta os passos para gerar o grafo base da arquitetura. O algoritmo recebe, como parâmetro de entrada, um valor inteiro indicando o número de cópias (desenrolamentos) do grafo base que devem ser gerados. Esse valor de entrada é determinado por uma heurística que avalia a geometria do DAG a ser escalonado. Uma apresentação sobre heurísticas e um modelo de PLI para determinação desse valor pode ser obtida em [31]. Se for considerado, por exemplo, que o processador alvo segue o modelo arquitetural de recursos apresentados na Figura 3.3, então um valor de entrada igual a 1 deve gerar o mesmo grafo apresentado na Figura 3.3, enquanto que um valor de entrada igual a 2 deve gerar um grafo semelhante ao apresentado na Figura 3.4.

Nos passos executados pelo Algoritmo 6, primeiro, os vértices representando UFs são criados de acordo com o nível de desenrolamento (linha 2) especificado como entrada. Na

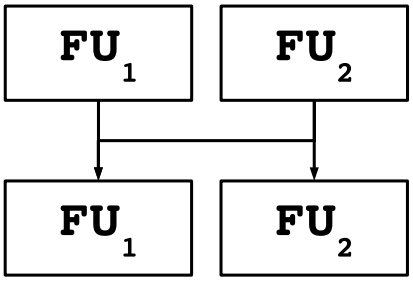


Figura 3.4: Grafo base da arquitetura desenrolado duas vezes.

linha 5 é adicionada uma aresta que conecta cada UF com sua respectiva cópia nos níveis de desenrolamento já construídos ($UF_j \rightarrow UF_i$).

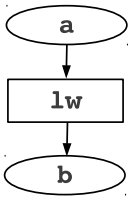
Algoritmo 6: Algoritmo para construção do grafo base.

```
   BaseGraphBuild(int unroll_number)
   |  Entrada: Número de cópias/desenrolamentos.
   |  Saída: Grafo base.
   |  /* cria um mapeamento de UFs e instruções.                    */
1  |  mapUFInst;
2  |  for int i = i_Unroll; i < unroll_number+i_Unroll; i++ do
3  |  |  for each value of the mapUFInst[i] do
   |  |  |  /* cria um vértice de UF.                               */
4  |  |  |  node_id nUF = new WNode(mapUFInst[i]->UF);
5  |  |  |  for int j = 0; j < i; j++ do
6  |  |  |  |  for each value of mapUFInst[j] do
   |  |  |  |  |  /* cria uma aresta conectando cada UFj anterior à UFi do
   |  |  |  |  |     ciclo i.                                        */
7  |  |  |  |  |  new WEdge(mapUFInst[j]->nUF, mapUFInst[i]->nUF);
8  |  |  |  |  end
9  |  |  |  end
10 |  |  end
11 |  end
12 |  i_Unroll = i_Unroll + unroll_number;
```

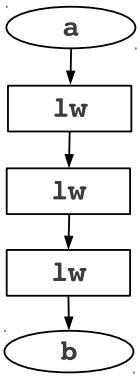
No que concerne às características de um grafo base, deve-se atentar para duas características importantes:

- Todo grafo base é conexo, independente do número de desenrolamentos que sejam especificados, uma vez que as unidades funcionais são, direta ou indiretamente, conectadas entre si.
- As Unidades Funcionais (UFs) possuem pesos que correspondem aos pesos das instruções que podem executar.

O Algoritmo 6 deve ser executado antes do algoritmo de *match* uma vez que uma das entradas do *matching* é um grafo base. O algoritmo de construção do grafo base é polinomial e possui complexidade de pior caso $O(N^2)$, N = unroll number. O conceito de desenrolamento de grafo, embora tratado até aqui apenas para grafos base, também é aplicado sobre grafos de entrada. Nesse caso, a aplicação acontece quando um determinado vértice do grafo de entrada, representando uma instrução, possui atributo de peso maior que 1. O desenrolamento desse grafo é necessário pois o grafo base executa um desenrolamento ciclo-a-ciclo de seus vértices e interconexões, de forma que, para associar com os vértices do grafo base, os vértices do grafo de entrada precisam também ter o comportamento de execução em 1 ciclo. Assim, o desenrolamento em nível de grafo de entrada acontece pelo "desenrolamento" de um ou mais vértices que possuem peso maior que 1. A Figura 3.5 mostra o efeito do desenrolamento de um vértice *lw* (representa a operação *load*) de um grafo de entrada. A Figura 3.5a apresenta o grafo de entrada com a representação de instruções e operandos. Na Figura 3.5b nota-se que o vértice *lw* foi desenrolado 3 vezes a fim de que cada novo vértice desse grafo represente apenas 1 ciclo.



(a) Grafo de entrada com vértice *lw* de peso 3.



(b) Grafo de entrada com desenrolamento do vértice *lw*.

Figura 3.5: Exemplo de desenrolamento do grafo de entrada.

3.3 Algoritmo para Escalonamento de Instruções e Alocação de Registradores Baseado em Isomorfismo de Subgrafos

No Algoritmo elaborado no âmbito deste trabalho, estende-se a proposta anterior [1] de escalonamento de instruções para também resolver o problema de alocação de registradores. Diante desse novo contexto, algumas situações aparecem e precisam ser manipuladas por esse algoritmo. Uma dessas situações é que, com a alocação de registradores, surge a necessidade de tratar o problema de código *spilled* para a memória. Dessa forma, o algoritmo original para tratamento heurístico após o *matching* passa a funcionar conforme o Algoritmo 7.

Algoritmo 7: Algoritmo de escalonamento de instruções e alocação de registradores baseado em isomorfismo de subgrafos.

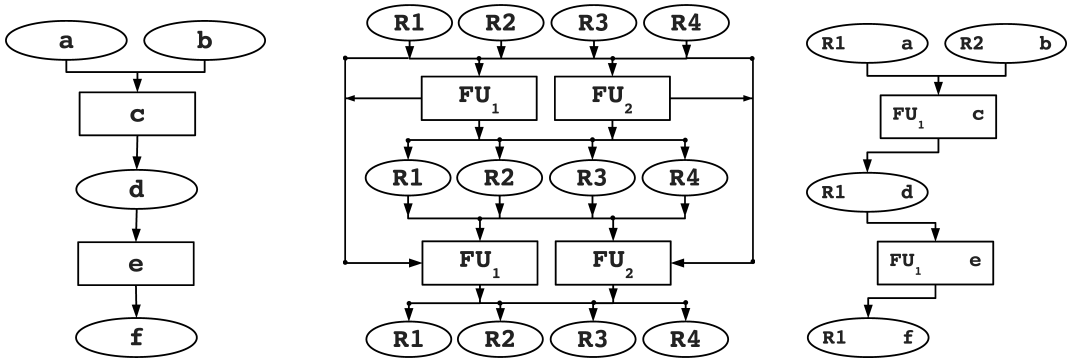
```
Sched(DAG G1, GRAFO BASE G2)
    Entrada: DAG de Entrada G1 e grafo base G2.
    Saída: Conjunto de instruções já escalonadas e alocadas.
1   topological_order(G1);
2   repeat
3       G'2=subg_iso_sched(G1,G2,&tag);
4       switch tag do
5           case 0: base_graph_resize(G2);
6           case 1: Spill_handler_vertices(G1);
7       end
8   until G'2 ≠ NULL;
9   Emit_Schedule();
```

O Algoritmo 7 apresenta os mesmos passos do Algoritmo 4. A única mudança significativa ocorre na linha 6, pois a invocação ao procedimento *Spill_handler()* indica que o *matching* não ocorreu devido a ausência de registradores suficientes e, portanto, um registrador precisa ser *spilled* para a memória. Como, inevitavelmente, esse procedimento alterará a topologia do grafo G_1 pois um dos vértices de G_1 (vértice de operandos) não será mais mapeado para o vértice de registradores de G_2. A necessidade de manipulação de *spill code* no algoritmo aqui proposto implica também na adoção de heurísticas para análise de *live ranges* e escolha de uma para ser *spilled*. Conforme será detalhado no Capítulo 4, utilizou-se a heurística clássica de Chaintin [25] já implementada na infraestrutura LLVM.

Com a manipulação de registradores, tanto o grafo de entrada quanto o grafo base apresentam modificações significativas em sua topologia. As Figuras 3.6a apresenta o mesmo DAG de entrada da Figura 3.2 mas agora com a representação explícita dos operandos. A Figura 3.6b apresenta o grafo base da Figura 3.4 agora com a modelagem de registradores. Por fim, o subgrafo G'_2 resultante do isomorfismo entre G_1 e G_2 é apresentado na Figura 3.6c. Cabe observar no grafo da Figura 3.6b que a aresta conectando dois vértices UF ainda é mantida na modelagem com a presença de vértices registradores. Isso acontece porque alguns vértices de instruções do grafo de entrada, devem sofrer o processo de desenrolamento para serem mapeáveis sobre os vértices de UFs, precisam que as UFs sejam interligadas via arestas.

Conforme apresentado e exemplificado na Figura 3.6a, neste novo algoritmo, o grafo de entrada também tem sua representação modificada uma vez que precisa modelar, explicitamente, os operandos das instruções. Essa modelagem ocorre na forma de vértices com atributos de tipo distintos daqueles dedicados às unidades funcionais/elementos de processamento. Para possibilitar o isomorfismo com o grafo base, os vértices que representam registradores também possuem os mesmos valores dos atributos tipo dos vértices operandos do grafo de entrada. Assim, faz-se necessário um novo procedimento para construção do grafo de entrada a fim de suportar a modelagem dessas novas características.

A construção do grafo entrada é apresentada no Algoritmo 8. O primeiro passo do algoritmo é ordenar as unidades de escalonamento topologicamente (linha 1). Em seguida,



(a) Grafo de entrada (G_1) com operandos explícitos.

(b) Grafo base (G_2) com representação de registradores.

(c) Grafo resultante (G_2') do isomorfismo entre G_1 e G_2.

Figura 3.6: Exemplo de grafo de entrada e grafo base com a modelagem de registradores e o grafo resultante $G_2^{'}$ do isomorfismo entre G_1 e G_2.

gera-se um DAG com vértices representando as instruções e seus operandos. Na linha 5 é criado um vértice para cada ciclo da instrução. Na linha 9 é criado um vértice de registrador para cada operando da instrução. Se o operando é definição (linha 11), então é criada uma aresta entre a instrução e o registrador. Se o operando é uso então é criada uma aresta entre o registrador e o vértice da instrução (linha 13). A complexidade do Algoritmo 8 é limitada pela complexidade do algoritmo de ordenação topológica (linha 1). Comumente, utiliza-se, nesse tipo de ordenação, um procedimento de busca em largura sobre o grafo de entrada a fim de determinar uma ordem para os vértices. Nesse caso, a complexidade de tempo de pior caso do algoritmo de construção do grafo do programa é $O(|V| + |E|)$.

O algoritmo para construção do grafo base também é alterado com a resolução dos problemas de escalonamento e alocação através do isomorfismo de subgrafos. O Algoritmo 9 é similar ao Algoritmo 6. No entanto, nota-se que a partir da linha 13 tem-se a construção de vértices representando os registradores da arquitetura e a interconexão entre vértices registradores e UFs. Interessante observar que o acréscimo de vértices registradores e interconexões entre esses vértices não altera a complexidade de tempo do algoritmo de construção do grafo base, uma vez que a geração dos vértices representando unidades funcionais e vértices representando registradores é realizada de maneira sequencial.

3.4 Exemplos de Processadores Modelados como Grafos Base

Como já apresentado, o algoritmo proposto neste trabalho atua sobre três tipos de grafos: grafo de entrada, grafo base e subgrafo resultante. A modelagem dos recursos arquiteturais por meio de um grafo base constitui-se numa característica singular desse algoritmo pois, uma vez que esses recursos sejam modelados, é possível, de forma geral, efetivar a geração de código para uma arquitetura. Assim, a modelagem de grafos base possibilita que o algoritmo seja, sem muitas dificuldades, portátil para outras arquiteturas de processadores.

Algoritmo 8: Algoritmo para construção do grafo do programa.

```
ProgramGraphBuild(SUnits)
    Entrada: SUnits: Array de Schedule units.
    Saída: small_graph: DAG do grafo do programa.
1   ScheduleDAGTopologicalSort Topo = new ScheduleDAGTopologicalSort(SUnits);
2   for each SU in Topo do
        /* cria um vértice de SUnit(SU).                                  */
3       node_id nSU = new WNode(SU);
4       node_id nSUJ = nSU;
5       for int l = 0; l < SU.Latency - 1; l++ do
            /* cria um vértice para cada ciclo de SU.                     */
6           nSUJ = new WNode(SU[l]);
            /* cria uma aresta conectando cada ciclo de SU.               */
7           new WEdge(nSU, nSUJ);
8       end
9       for each operand of the instruction do
            /* cria um vértice de Reg.                                    */
10          node_id nReg = new WNode(Reg);
11          if operand is Def then
                /* cria uma aresta de SU para Reg.                        */
12              new WEdge(nSUJ, nReg);
13          else
                /* operando é uso.                                        */
                /* cria uma aresta de Reg para SU.                        */
14              new WEdge(nReg, nSU);
15          end
16      end
17  end
```

Em virtude da importância da modelagem através de grafos base, esta seção descreve exemplos de grafos base para diferentes arquiteturas de processadores.

3.4.1 Grafo Base para o Processador MIPS

Como descrito em [32], o processador MIPS segue uma arquitetura escalar com uma unidade funcional (ULA) para execução de instruções de inteiros, lógicas e desvios, uma unidade funcional para execução de instruções de ponto flutuante e um banco de registradores composto por 32 registradores de 32 bits cada. A Figura 3.7 mostra um exemplo do grafo base da arquitetura MIPS. Para representar corretamente às características e funcionalidades desse processador para o algoritmo de escalonamento e alocação de registradores baseado em isomorfismo, o atributo de tipo de cada vértice FU deve ser específico uma vez que cada tipo de FU é apta a executar tipos distintos de instruções. O mesmo raciocínio se aplica ao atributo tipo dos vértices registradores. Todos os vértices registradores podem possuir o mesmo valor para esse atributo mas, tal valor deve ser diferente do atributo dos vértices FU.

Algoritmo 9: Algoritmo para construção do grafo base com unidades funcionais e registradores.

```
BaseGraphBuild(int unroll_number)
    Entrada: Número de cópias/desenrolamentos.
    Saída: Grafo base.
    /* cria um mapeamento de UFs e instruções.                          */
1   mapUFInst;
    /* cria um mapeamento de UFs e registradores.                       */
2   mapUFReg;
3   for int i = i_Unroll; i < unroll_number+i_Unroll; i++ do
4       for each value of the mapUFInst[i] do
            /* cria um vértice de UF.                                   */
5           node_id nUF = new WNode(mapUFInst[i]->UF);
6           for int j = 0; j < i; j++ do
7               for each value of mapUFInst[j] do
                    /* cria uma aresta conectando cada UFj anterior à UFi do
                       ciclo i.                                         */
8                   new WEdge(mapUFInst[j]->nUF, mapUFInst[i]->nUF);
9               end
10          end
11      end
12  end
13  for int i = i_Unroll; i < unroll_number+i_Unroll; i++ do
14      for each value of the mapUFReg do
            /* cria um vértice de Reg.                                  */
15          node_id nRegUse = new WNode(mapUFReg[i]->Reg);
16          node_id nRegDeg = new WNode(mapUFReg[i]->Reg);
17          for each UF and instruction do
18              if instruction description has Reg then
                    /* cria uma aresta de UF para Reg.                  */
19                  new WEdge(nUF, nRegDeg);
                    /* cria uma aresta de between Reg para UF.          */
20                  new WEdge(nRegUse, nUF);
21                  for int j = 0; j < i; j++ do
22                      for each value of the map of UF and registers of previews cycles
                        j do
                            /* cria uma aresta para conectar cada Regj anterior
                               para Regi do ciclo i.                    */
23                          new WEdge(mapUFReg[j]->nReg, mapUFReg[i]->nReg);
24                      end
25                  end
26              end
27          end
28      end
29  end
30  i_Unroll = i_Unroll + unroll_number;
```

Sem perda de generalidade, o grafo é apresentado com apenas 4 registradores para facilitar a compreensão da modelagem.

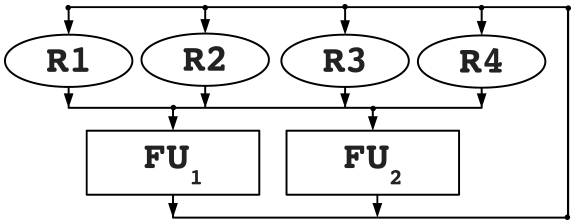
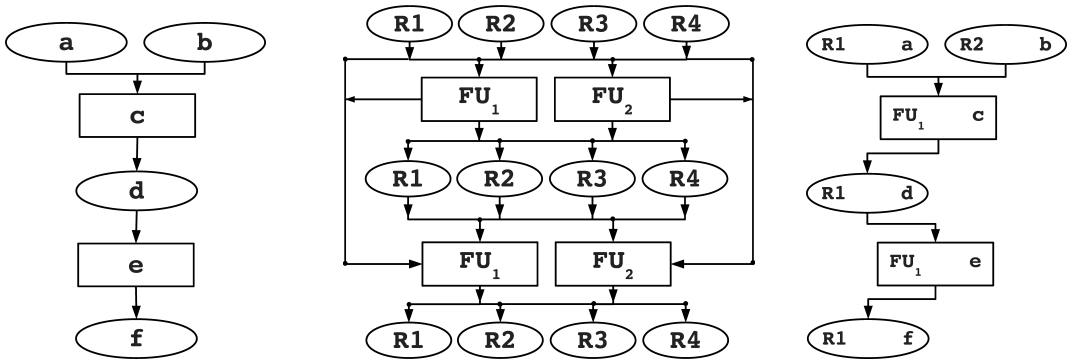


Figura 3.7: Grafo base da arquitetura MIPS representado com 4 registradores.

A Figura 3.8 apresenta um exemplo da aplicação do algoritmo de isomorfismo de subgrafos para resolução dos problemas de escalonamento e alocação de registradores sobre a arquitetura MIPS. Na Figura 3.8a é apresentado um DAG que representa o programa de entrada e a Figura 3.8b apresenta o grafo base da arquitetura MIPS já desenrolado adequadamente para o tamanho do grafo de entrada. Esse desenrolamento torna possível o *match* para o grafo G_1. Na Figura 3.8c é apresentado o resultado do *match*. De acordo com o resultado e a Definição 11, tem-se que $\varphi(a) = R1$, $\varphi(b) = R2$, $\varphi(c) = FU_1$, $\varphi(d) = R1$, $\varphi(e) = FU_1$, $\varphi(f) = R1$.



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

Figura 3.8: Exemplo de Isomorfismo de Subgrafos na arquitetura MIPS.

3.4.2 Grafo Base para o Processador HPL-PD

A arquitetura de processador HPL-PD [33] é uma arquitetura paramétrica que utiliza nove unidades funcionais (duas para instruções de memória, quatro para instruções de inteiros, uma para desvio e duas para instruções de ponto flutuante) e cinco bancos de registradores interligados numa topologia particular entre essas unidades funcionais. A Figura 3.9 apresenta o grafo base para a arquitetura *Hewlett-Packard Laboratories PlayDoh* (HPL-PD). Para modelar esse processador, deve-se atentar para o fato que existem nove unidades funcionais para execução de instruções agrupadas em quatro tipos distintos. Logo, o grafo base deve representar, como vértices, essas unidades funcionais mas tendo o devido cuidado em definir quatro atributos de tipo adequadamente entre esses recursos. No caso dos vértices que representam registradores acontece situação similar. Independente do número total de registradores, há quatro tipos específicos desses recursos e, sendo assim, os vértices

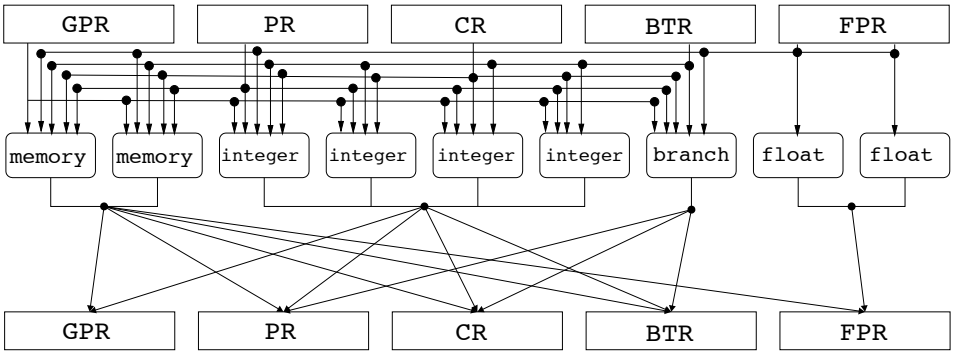


Figura 3.9: Exemplo de grafo base do HPL-PD.

que modelam registradores GPR, por exemplo, devem possuir atributo de tipo distinto daqueles que modelam registradores PR.

Na Figura 3.10 é apresentado o grafo base com um desenrolamento de dois níveis ($unroll\_number = 2$). Para simplificar o entendimento desse grafo base, os vértices registradores de um mesmo tipo estão sendo representados por um vértice indicado pelo nome do banco de registradores.

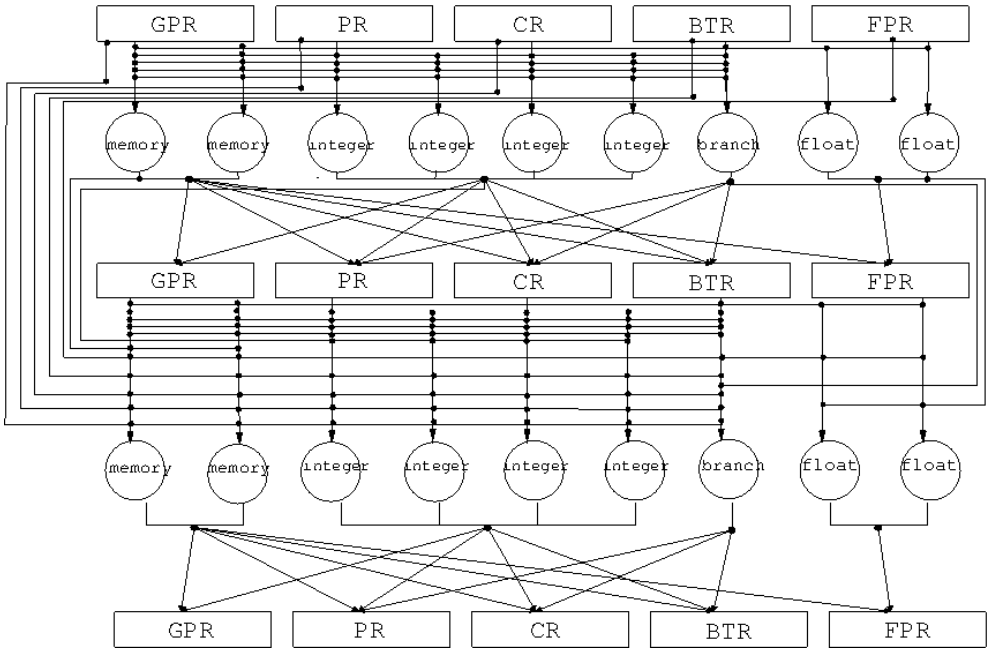


Figura 3.10: Grafo base do processador HPL-PD desenrolado.

3.4.3 Grafo Base para o Processador ρ-VEX

O processador ρ-VEX adota o conjunto de instruções e modelo de execução VLIW da arquitetura VEX [34] e possui quatro unidades funcionais e dois bancos de registradores sendo um de propósito geral com 32 registradores de 32 bits e outro com oito registradores de 1 bit. A Figura 3.11 apresenta um exemplo de grafo base para a arquitetura *Reconfigurable and*

Extensible VLIW Processor (ρ-VEX). Na Figura 3.12 é apresentado o grafo base desenrolado. Na construção do grafo base para representar os recursos do processador ρ-VEX deve-se estar atento para a situação que uma unidade funcional pode executar dois tipos de instruções. Assim, a implementação desse grafo deve considerar essa situação e definir dois atributos de tipo ($tipo_1$ e $tipo_2$) para cada vértice representando uma unidade funcional. Dessa forma, esses vértices poderão receber um mesmo valor para o atributo $tipo_1$ (indica ALU) e um valor distinto para o atributo $tipo_2$ (cada valor representará CTRL, MUL, MEM). Com esses dois atributos, no momento do *match*, o algoritmo poderá escolher, por exemplo, para associar com o vértice ALU/CTRL tanto um vértice de instrução com atributo ALU quanto um vértice de instrução com atributo CTRL.

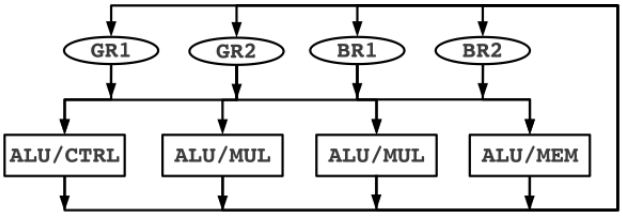


Figura 3.11: Exemplo de grafo base do ρ-VEX.

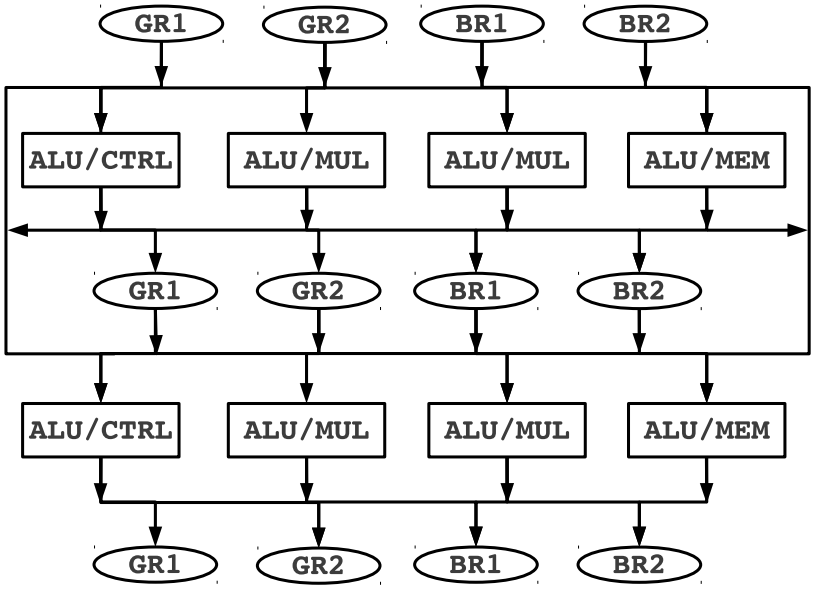


Figura 3.12: Exemplo de desenrolamento do grafo base do ρ-VEX.

3.5 Considerações Finais

Esse capítulo descreveu com detalhes a proposta de um algoritmo integrado para escalonamento de instruções e alocação de registradores. Como forma de exemplificar a aplicação do algoritmo, apresentou-se um estudo de caso detalhado envolvendo um processador hipotético

e possibilidades de extensão do algoritmo para outras arquiteturas como MIPS, HPL-PD e ρ-VEX.

O Capítulo 4 apresenta a infraestrutura de compilação LLVM e descreve os passos realizados para a integração do algoritmo aqui proposto com o fluxo de execução dessa infraestrutura de compilação.

Capítulo 4

Infraestrutura de Compilação LLVM

Este capítulo apresenta a infraestrutura de compilação LLVM. Destaca-se o funcionamento e organização do *back-end* LLVM e apresenta-se os procedimentos necessários para integração do algoritmo proposto para escalonamento de instruções e alocação de registradores com essa infraestrutura de compilação.

4.1 Compilador LLVM

O compilador LLVM [21] é uma infraestrutura modular construída com uma coleção de ferramentas reutilizáveis. O LLVM foi iniciado como um projeto da Universidade de Illinois e desde então cresceu como um projeto consistente. O código do projeto está sob a licença da *University of Illinois at Urbana-Champaign* (UIUC) *Berkeley Software Distribution (BSD)-Style*. O LLVM, em 2010, recebeu o prêmio da ACM *Special Interest Group on Programming Languages* (SIGPLAN) em reconhecimento do impacto que teve na comunidade de pesquisa em compiladores, que pode ser constatado pelo grande número de publicações que trazem referências a esse compilador. Algumas das vantagens da infraestrutura do LLVM são o seu *design*, a linguagem independente e a extensibilidade.

A Figura 4.1 representa em alto nível a infraestrutura do LLVM. O estágio **compilador estático** realiza a transformação de linguagens como *C, C++ e Objective-C* para uma linguagem de representação intermediária (LLVM IR), realizada no *front-end*. Pode-se utilizar o *Clang* como *front-end*. A linguagem intermediária LLVM IR é baseada em *Reduced Instruction Set Computer* (RISC) e na forma *Static Single Assignment* (SSA) [35]. Com essa linguagem intermediária, o LLVM gera o LLVM *Object*, que pode ser executado pela infraestrutura do LLVM utilizando o tradutor *Just in Time* (JIT). Então, o LLVM pode trabalhar com o LLVM *Object*, chamado também de *bitcode*, realizando otimizações, gerando o código *Assembly* ou gerando código nativo (já *link*-editado) para arquiteturas específicas de máquinas como *X86, ARM, PowerPC, MIPS, SPARC, XCore*, entre outras.

A forma SSA é uma representação intermediária em que cada variável tem somente uma definição no programa. Na Figura 4.2 há um exemplo de código em C e a respectiva versão na forma SSA [26].

Dentre o conjunto de ferramentas do LLVM, destacam-se:

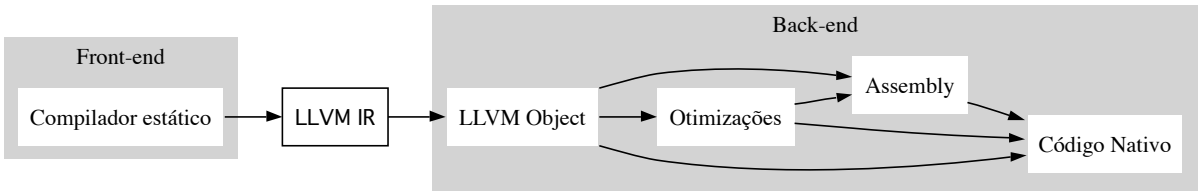


Figura 4.1: Visão em alto nível da infraestrutura LLVM.

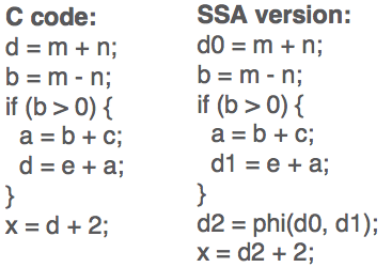


Figura 4.2: Exemplo baseado em SSA.

- *clang*: invoca o *front-end* gerando um objeto LLVM IR.
- *llc*: invoca o *back-end* recebendo um objeto LLVM IR.
- *opt*: invoca o *back-end* fazendo otimizações em um objeto LLVM IR.

O *back-end* do LLVM contém bibliotecas independentes para cada arquitetura. Nesta fase há a tradução da LLVM IR para instruções reais e registradores reais. Um exemplo de código na representação em C e em LLVM IR, respectivamente, é apresentado na Figura 4.3:

```
main() {
int a,b;
a = b + 50;
}
```

```
define i32 @main() nounwind ssp {
entry:
  %retval = alloca i32
  %a = alloca i32
  %b = alloca i32
  %"alloca point" = bitcast i32 0 to i32
  %0 = load i32* %b, align 4
  %1 = add nsw i32 %0, 50
  store i32 %1, i32* %a, align 4
  br label %return

return:
  %retval1 = load i32* %retval
  ret i32 %retval1
}
```

Figura 4.3: Exemplo de código em C e em LLVM IR.

A Figura 4.4 representa uma visão abstrata dos estágios de geração de código no *back-end*. A geração de código está dividida nos seguintes estágios [36]:

- **Seleção de instrução**: produz um código inicial para o conjunto de instruções. A seleção de instruções então faz uso de registradores virtuais na forma SSA e registradores físicos que são necessários devido à restrição da arquitetura. Tem como entrada

as instruções LLVM IR e constrói um DAG das instruções chamado de ilegal (contém instruções não suportadas) e a representação do conjunto de instruções da máquina alvo. Esta fase produz como saída um DAG de instruções (legais) da arquitetura alvo (*SelectionDAG*), utilizando um algoritmo guloso adaptado para DAG.

- **Escalonamento**: recebe o *SelectionDAG* de instruções legalizado, determina uma ordem para as instruções, então emite as instruções como instruções de máquina, utilizando uma variação do algoritmo *list scheduling*.

- **Otimização em código de máquina baseado em SSA**: é uma fase opcional que opera sobre a forma SSA produzida pela seleção de instruções. Exemplos de otimizações nessa fase são *modulo-scheduling* e *peephole*.

- **Alocação de registradores**: o código é transformado de uma infinidade de registradores virtuais numa forma SSA para registradores físicos. Esta fase insere *spill code* e elimina todas as referências de registradores virtuais do programa.

- **Inserção de código prólogo/epílogo**: nesta fase códigos de prólogo e epílogo podem ser inseridos e as localizações de referências abstratas são removidas.

- **Otimizações tardias**: otimizações que operam no código final de máquina são realizadas nesta fase. Exemplos de otimizações nesta fase são *spill code scheduling* e *peephole*.

- **Emissão de código**: a fase final tem como saída o código em formato *assembly* ou em código de máquina.

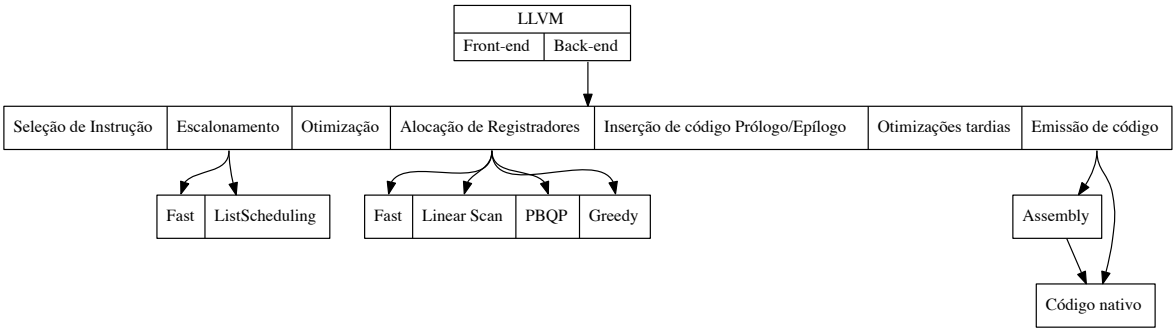


Figura 4.4: Fases do LLVM na geração de código.

No tocante à otimização e geração de código no compilador LLVM, destaca-se a estrutura *Pass*. Essa estrutura gerencia a execução das transformações e otimizações realizadas pelo compilador sobre um programa e mesmo em casos em que se deseja adicionar funcionalidades ao próprio compilador. Dependendo das funcionalidades atribuídas ao *Pass*, faz-se necessário herdar métodos das classes *ModulePass, CallGraphSCCPass, FunctionPass, LoopPass, RegionPass ou BasicBlockPass*, que adicionam funcionalidades ao *Pass*. Uma das principais características do *Pass* é o escalonamento dos *passes* que buscam uma execução eficiente a partir das restrições que o *Pass* encontra (indicado pela classe que foi herdada). A classe

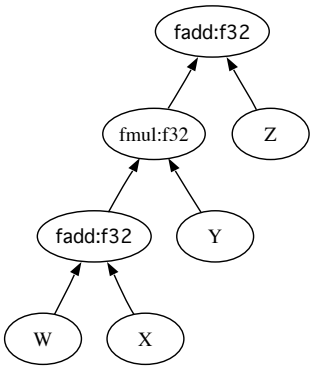
que gerencia o escalonamento dos *passes* é a classe *PassManager*. Nela, os resultados de análises são compartilhados, não permitindo assim o cálculo de uma análise mais de uma vez. Para realizar a integração de novos algoritmos junto à infraestrutura LLVM, o entendimento adequado da estrutura *Pass* é primordial, uma vez que é através dessa estrutura que o LLVM disponibiliza interfaces com novas funcionalidades. A inclusão e análise de desempenho, no compilador LLVM, do algoritmo de escalonamento e alocação e registradores inicia pela construção de classes herdadas a partir da classe *FunctionPass*.

4.2 Escalonamento de Instruções no LLVM

No LLVM, o escalonamento de instruções inicia na última fase da seleção de instruções. Esta fase recebe o DAG das instruções alvo produzidas pela fase seleção de instruções e determina uma ordem de execução para cada vértice do DAG. Então, o DAG é convertido para uma lista de *MachineInstrs* e o *SelectionDAG* é destruído. Este passo usa a técnica tradicional *prepass*. Essa fase é logicamente separada da fase de seleção de instrução. No entanto, ambas estão conectadas pelo fato de que operam sobre *SelectionDAGs* [36]. Um exemplo de representação que utiliza o *SelectionDAG* é apresentado na Figura 4.5. O trecho de código da Figura 4.5a é representado na forma de um *SelectionDAG* na Figura 4.5b. O grafo é construído de baixo para cima, mas a execução se mantém como a mesma semântica do código: primeiro a adição, de ponto flutuante, entre W e X, depois multiplica o seu resultado por Y e em seguida soma o resultado com Z.

```
%t1 = fadd float %W, %X
%t2 = fmul float %t1, %Y
%t3 = fadd float %t2, %Z
```

(a) Trecho de código.



(b) *SelectionDAG* correspondente.

Figura 4.5: Exemplo do SelectionDAG (fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z).

O algoritmo padrão para escalonamento de instruções utilizado pelo LLVM é o *list scheduling*, que pode ser utilizado em duas abordagens: *top down* e *bottom up*. Resumidamente, o algoritmo usa uma fila de prioridade para escalonar. A cada instante, os vértices de mesma prioridade são retirados da fila, em ordem, e escalonados se for um escalonamento legal. Uma ilegalidade pode ser devido a conflitos estruturais ou porque uma entrada para a instrução não completou a execução.

Um outro algoritmo também utilizado para o escalonamento é *Fast*, que executa o *list scheduling* com uma fila de prioridade para escalonar a estrutura de dados *FastPriorityQueue*, que simplesmente é uma fila de prioridades onde todos os vértices do grafo de escalonamento têm a mesma prioridade.

Independente do algoritmo de escalonamento utilizado, as informações (quantidade e características) sobre os elementos de processamento/unidades funcionais disponíveis para alocação são definidas na estrutura Tablegen, no arquivo TargetInstrInfo.td. Nesse arquivo estão definidas as instruções com seus formatos, opcodes, operandos e itinerário. Nos itinerários são definidos os ciclos necessários para execução de uma instrução.

4.3 Alocação de Registradores no LLVM

O primeiro passo é a transformação das instruções, que são colocadas na forma SSA. Apesar da forma SSA simplificar análises que são executadas no grafo de fluxo de controle, o conjunto de instruções tradicionais não implementam instruções *PHI*. Então, o compilador deve substituir as instruções *PHI* com outras instruções que preservam a mesma semântica [36]. O próximo passo para a alocação é a determinação das *live ranges* das variáveis vivas.

O LLVM denota os registradores físicos por números inteiros de 1 a 1023. Assim, os registradores virtuais iniciam a partir do 1024. Para algumas arquiteturas como X86, alguns registradores físicos são marcados como *aliased*, por compartilharem o mesmo endereço físico (exemplo: o EAX, AX e AL, compartilham os oito primeiros bits). Registradores físicos, no LLVM, são agrupados em *Register Classes*. Registradores físicos são estaticamente definidos no arquivo *TargetRegisterInfo.td* enquanto que os virtuais devem ser criados chamando *MachineRegisterInfo::createVirtualRegister()*.

A infraestrutura do LLVM provê algumas opções de alocadores de registradores:

- *Fast*: aloca basicamente em nível de bloco básico, tentando manter valores em registradores. Não há registradores vivos entre blocos. Portanto tudo é *spilled* no início e no final de cada bloco. Esse algoritmo é usado, principalmente, em situações de depuração de código com registradores alocados.
- *Linear Scan*: o algoritmo utiliza uma estratégia gulosa para realizar a alocação de registradores. O algoritmo inicia transformando os blocos básicos numa sequência linear. Então, realiza a coloração baseada nas *live ranges* em que duas *live ranges* que se sobrepõem não podem ser alocadas para o mesmo registrador [37].
- PBQP — (*Partitioned Boolean Quadratic Programming)*: trabalha na construção de um problema PBQP como um problema de alocação de registradores. Resolve-se o problema PBQP e converte a solução para um problema de alocação de registradores. Se alguma variável foi selecionada para *spill*, então o código *spill* é adicionado e o processo é repetido. A representação do PBQP é um grafo em que os vértices são os registradores virtuais e as arestas são as restrições entre os registradores (como as interferências). Cada vértice é associado a um vetor de custo (que dá o custo de cada alocação para alocar o registrador virtual). Tipicamente, pode-se estimar o custo de *spill* e o custo zero para cada registrador. Pode-se também atribuir valores de custo positivos ou negativos para cada registrador físico priorizando-o. Cada aresta é associada a uma matriz de custo (que é a interferência entre os vértices). O custo de uma solução específica (alocação de registradores) é dada pela soma dos custos de todos os vértices e arestas do grafo. O problema é resolvido por programação dinâmica [38].

- *Greedy*: utiliza uma fila de prioridades para a alocação de registradores. Os registradores virtuais são colocados na fila de prioridades de acordo com o tamanho de suas *live ranges*. Para alocar um registrador físico, escolhe-se as maiores *live ranges*. Se a *live range* não pode ser alocada, o peso é incrementado. O maior peso é usado como critério para selecionar um registrador virtual para *spill*. O peso inicial das *live ranges* é calculado antes da alocação de registradores iniciar.

4.4 Integração do Algoritmo de Escalonamento e Alocação de Registradores Baseado em Isomorfismo de Subgrafos no LLVM

O processo de integração do algoritmo proposto neste trabalho com a infraestrutura de compilação LLVM foi realizada a partir da estrutura *Pass* na fase de alocação de registradores. O desafio principal dessa atividade é relacionar, corretamente, as estruturas e interfaces disponibilizadas pelo LLVM com as características e funcionalidades do algoritmo. A Figura 4.6 exibe as fases de geração de código agora com a inclusão do algoritmo de isomorfismo de subgrafos.

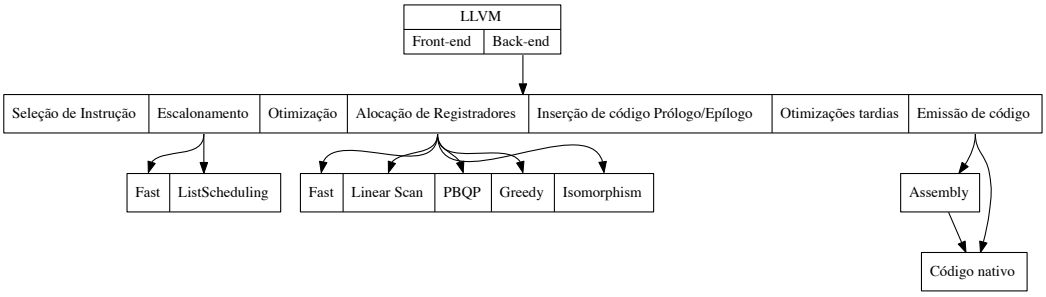


Figura 4.6: Fases do LLVM na geração de código com acréscimo do algoritmo de isomorfismo de subgrafos.

O Algoritmo 10 detalha o método *runOnFunctionMachine* da classe *Pass* implementada. A implementação desse método como etapa inicial da integração do algoritmo de escalonamento e alocação junto ao LLVM se faz necessária para utilizar as interfaces do LLVM para acesso ao código do programa e para compatibilizar a saída do algoritmo com a saída esperada pelas demais etapas do LLVM. Nesse método é implementado a compatibilização das estruturas de dados utilizadas pelo LLVM com as estruturas de entrada da biblioteca VF [29, 30] para realização do *matching* entre o grafo de entrada e o grafo base. Além disso, implementa o tratamento heurístico que deve ser realizado sobre os grafos base ou de entrada, caso o *matching* não seja possível. A implementação desse método, no LLVM, corresponde, em alto nível, à implementação do Algoritmo 7, apresentado no Capítulo 3.

O primeiro passo do Algoritmo 10 consiste em construir o grafo base da arquitetura alvo (linha 2).Para construir o grafo base a partir dos recursos existentes na arquitetura alvo, o construtor do grafo base obtém as informações de InstrItineraryData, TargetInstrInfo e

TargetRegisterInfo a partir de MachineFunction. Então, o algoritmo constrói o grafo entrada e reconstrói cada vez que ocorrer *spill* (linha 4). Na linha 6 é executado o método *run* que é baseado na implementação do *PostSchedRA* cujo objetivo é construir um DAG a partir das instruções do bloco básico.

O DAG é então convertido para estruturas de dados utilizados pela biblioteca VF (linhas 7 - 8). A partir daí, pode então ser passado para o Algoritmo 8 para ser transformado no DAG que utiliza as classes da biblioteca VF, quando a execução do Algoritmo 5 de *match* é realizado (linha 9).

Durante a execução do procedimento VFMatch, invoca-se um método para verificar a compatibilidade (Algoritmo 11) entre um vértice do grafo do programa e um vértice do grafo da arquitetura. Esse método verifica as restrições que a arquitetura define para a execução de uma instrução. Se um vértice operando (registrador virtual) não pode ser alocado, então ele é marcado para ser *spilled* (linha 12). Se o *match* retorna que um vértice de instrução não pode ser combinado, então o grafo base deve ser desenrolado (linha 11) executando o *match* novamente. Se o retorno for verdadeiro então na linha 16 o escalonamento é emitido com os operandos das instruções com registradores físicos.

Algoritmo 10: Algoritmo de runOnFunctionMachine método herdado da classe *Pass*.

```
runOnFunctionMachine(FunctionMachine: Fn)
    Entrada: FunctionMachine Fn que é o objeto que contém o programa e a
             arquitetura alvo.
    Saída: Programa com as instruções escalonadas e registradores alocados.
1   BaseGraph base(Fn);
2   large_graph = base.build();
3   bool match;
4   while not match do
5       foreach basic block BB of Fn do
6           SUnits = PostSchedRA.run(BB);
7           small_graph = Program.build(SUnits);
8           VF2MonoState s1(small_graph, large_graph, false);
9           match = VFMatch(s1, vis);
10          switch tag do
11              case 0: base.unroll();
12              case 1: spiller().spill(VirtRegtoSpill);
13          end
14      end
15  end
16  EmitSchedule();
```

O Algoritmo 11 é invocado a partir do método VFMatch (Algoritmo 5) e verifica se o vértice do grafo base é uma unidade funcional e o vértice do grafo de entrada é uma instrução. Então, retorna verdadeiro se a unidade funcional suporta a execução dessa instrução (linha 2). Um detalhe a considerar é que se é uma instrução com vários ciclos e ela já foi alocada no primeiro ciclo então o algoritmo deve retornar verdadeiro para a mesma UF em todos os ciclos. Se a comparação é entre vértices registradores e operandos e o operando (registrador

virtual) já foi alocado, então retorna verdadeiro (linha 5). Senão, aloca um registrador (linha 7) e retorna se foi possível alocar.

Algoritmo 11: Algoritmo que verifica a compatibilidade de um vértice do grafo do programa com um vértice do grafo base.

```
   isFeasiblePair(WNode n1, WNode n2)
   |  Entrada: Vértice n1 do grafo de entrada e vértice n2 do grafo base.
   |  Saída: Verdadeiro ou falso.
1  |  if n1->get_kind() == SUNI AND n2->get_kind() == UF then
   |  |  /* retorna verdadeiro se n2 pode executar o vértice n1          */
2  |  |  return true
3  |  else if n1->get_kind() == REGSU AND n2->get_kind() == REGUF then
4  |  |  if n1->VRM->getPhys(n1->Reg) == n2->Reg then
5  |  |  |  return true;
6  |  |  else
7  |  |  |  bool assigned = n1->VRM->assignVirt2Phys(n1->Reg, n2->Reg);
8  |  |  |  return assigned;
9  |  |  end
10 |  end
11 |  return false;
```

O Algoritmo 12 também é invocado a partir do método VFMatch (Algoritmo 5), e indica que o *match* entre os grafos foi concluído. O *visitor* percorre o DAG do grafo do programa reescrevendo-o, utilizando as estruturas do LLVM. Primeiro faz uma chamada para reescrever os registradores virtuais em registradores físicos (linha 1). Então, para cada vértice do grafo do programa (linha 2), verifica se é um vértice do grafo do programa e se não foi escalonado (linhas 4 e 6). Nesse caso, o vértice é marcado para ser escalonado (linha 8). No retorno verdadeiro a sequência do código a ser executada é a emissão das instruções, na linha 16 do Algoritmo 10. Assim, o código está novamente na forma de instruções de máquina e os próximos passos da geração de código são executados.

4.5 Considerações Finais

Esse capítulo apresentou as funcionalidades e organização de software da infraestrutura de compilação LLVM e o projeto e implementação da integração do algoritmo proposto neste trabalho junto a essa ferramenta de compilação. As atividades envolvidas na integração do algoritmo com o compilador LLVM foram primordiais para a utilização, validação e comparação da técnica de escalonamento e alocação de registradores deste trabalho com outras técnicas clássicas da literatura da área e já implementadas junto ao LLVM. O Capítulo 5 detalha e discute os experimentos desenvolvidos e resultados obtidos neste trabalho.

Algoritmo 12: Algoritmo de visitor.

```
visitor(State s)
    Entrada: State: Grafo de estados que contém o grafo do programa (small) e o
             grafo base (large).
    Saída: Programa escalonado com registradores alocados.
1   VRM->rewrite();
2   for each node in small do
3       SUnit su;
4       if prog->get_kind() == SUNI then
5           su = prog->su;
6           if su is not Scheduled then
7               Sequence->add(su);
8               su->isScheduled = true;
9           end
10      end
11  end
12  return true;
```

Capítulo 5

Experimentos e Resultados

Este capítulo apresenta os experimentos e os resultados desenvolvidos sobre o algoritmo proposto neste trabalho. Na Seção 5.1 são apresentados os experimentos e resultados do desempenho do algoritmo junto ao compilador LLVM e a comparação desse desempenho com algoritmos clássicos (já implementados no compilador) de escalonamento de instruções e alocação de registradores. Na Seção 5.2 estão os experimentos envolvendo os impactos sobre o desempenho de programas gerados com o algoritmo. Independente do tipo de experimento, os programas foram compilados utilizando o compilador LLVM build 2.9 e gerados considerando o *backend* disponibilizado para o processador MIPS.

5.1 Análise de Desempenho do Algoritmo Proposto

Nesta seção apresenta-se os experimentos sobre as características do código gerado a partir da compilação do LLVM sobre programas de diferentes *benchmarks*. Nessas características, procurou-se analisar o tempo de compilação de cada programa com o algoritmo de isomorfismo de subgrafos e dois outros algoritmos já existentes no LLVM. Também foi analisado o número de instruções correspondentes aos *spills*, uma vez que essa informação está diretamente ligada à capacidade do algoritmo de alocação de registradores em utilizar adequadamente os registradores da máquina alvo.

Os experimentos de compilação e simulação foram executados sobre uma máquina com processador Intel Core i3-2100, clock de 3.1 GHz com 2G de memória RAM. Os resultados obtidos com a utilização do algoritmo de isomorfismo de subgrafos são comparados com algoritmos clássicos de escalonamento (*list scheduling ILP*) e alocação de registradores (*Linear Scan e Greedy).*

Nas Tabelas 5.1 e 5.2 são apresentados os valores de tempos de compilação, em segundos, para 17 programas de diferentes *Benchmarks* (SimpleBench [39], Stanford e Shootout [40]). A média aritmética e o desvio padrão do tempo de compilação foram calculados e considerados para apresentação apenas após atingirem um intervalo de confiança de 95% (30 execuções para cada programa em cada configuração do LLVM, limpando a cache a cada execução).

As Tabelas 5.1 e 5.2 possibilitam a comparação entre os tempos de compilação para programas gerados com o algoritmo de isomorfismo de subgrafos e algoritmos *list schedu-*

		List-ilp+Greedy		List-ilp+Linearscan	
Benchmark	Programa	Média	Desvio Padrão	Média	Desvio Padrão
Shootout	ackermann	0.0002	0.0007	0.0001	0.0000
	fib	0.0014	0.0019	0.0009	0.0016
	hello	0.0000	0.0000	0.0000	0.0000
	nestedloop	0.0001	0.0000	0.0001	0.0000
SimpleBench	alloca	0.0000	0.0000	0.0000	0.0000
	dag	0.0000	0.0001	0.0000	0.0000
	double-linked-list	0.0012	0.0018	0.0009	0.0016
	eight	0.0000	0.0000	0.0000	0.0000
	hyper	0.0000	0.0000	0.0000	0.0000
	ifthen	0.0000	0.0000	0.0000	0.0000
	linked-list	0.0007	0.0015	0.0003	0.0010
	local_var_test	0.0003	0.0010	0.0001	0.0016
	nested	0.0002	0.0007	0.0003	0.0007
	sqrt	0.0000	0.0000	0.0000	0.0000
Stanford	IntMM	0.0010	0.0020	0.0007	0.0013
	Quicksort	0.0015	0.0019	0.0016	0.0019
	RealMM	0.0017	0.0020	0.0008	0.0015

Tabela 5.1: Tempo médio de compilação, em segundos, dos programas utilizando escalonamento de instruções *List Scheduling* e alocação de registradores com o algoritmo *Greedy* e *Linearscan*.

		Isomorphism		
Benchmark	Programa	Média	Desvio Padrão	Grafo Base
Shootout	ackermann	6.3922	0.3734	5.4923
	fib	6.3921	0.0755	5.1163
	hello	2.8092	0.0114	2.5402
	nestedloop	2.9450	0.1536	2.5282
SimpleBench	alloca	3.0297	0.1455	2.5122
	dag	13.4195	0.1610	2.2361
	double-linked-list	10.7451	0.1542	2.2401
	eight	10.3202	0.1554	2.2241
	hyper	3.0636	0.1423	2.5122
	ifthen	13.4139	0.1526	2.2241
	linked-list	3.0671	0.1321	2.2521
	local_var_test	7.2520	0.1482	0.6600
	nested	3.0757	0.1554	2.5282
	sqrt	18.8901	0.1879	2.5042
Stanford	IntMM	66.8247	0.1771	5.2843
	Quicksort	6.4373	0.2821	5.1243
	RealMM	45.3975	0.1668	5.2923

Tabela 5.2: Tempo médio de compilação, em segundos, dos programas utilizando escalonamento de instruções e alocação de registradores com o algoritmo de isomorfismo de subgrafos.

ling+greedy e *list scheduling+linearscan*. A justificativa para o maior tempo de compilação obtido por parte do algoritmo de isomorfismo é devido a dois fatores principais:

1. a implementação, no LLVM, do algoritmo de isomorfismo e do algoritmo de geração de subgrafos não pode contar com a utilização das heurísticas [31] para definição do tamanho do grafo base. Essa ausência deve-se à dificuldade em adaptar essas heurísticas, no tempo de desenvolvimento deste trabalho, para utilizar as interfaces do LLVM a fim de manipular mais de um DAG em um mesmo bloco básico.

2. ausência de um algoritmo de isomorfismo de subgrafos com melhor desempenho que o disponibilizado pela biblioteca VF e adaptação para outra biblioteca não seria realizada em tempo hábil.

O primeiro fator impactou diretamente no desempenho final do algoritmo uma vez que, sem uma definição adequada do parâmetro de entrada para o algoritmo de geração do grafo base (*unroll_number*), o *matching* não pode ser executado e há sucessivas chamadas para o algoritmo de geração do grafo base a fim de que o tamanho desse grafo seja incrementado. Apesar de polinomial, o impacto do algoritmo de geração do grafo base é observado no tempo de compilação dos programas do *benchmark* Shootout, nos programas *alloca, hyper, linked-list* e *nested* do *benchmark* SimpleBench e no programa *quicksort* do *benchmark* Stanford.

O segundo fator foi preponderante para o maior tempo de compilação dos demais programas avaliados. Nesse caso, a determinação do tempo deve-se às características do algoritmo em verificar todos os vértices do grafo base, ainda não alocados, a fim de testar a factibilidade do procedimento de *matching*. Esse método de busca exaustiva pela solução, associada a uma heurística de tempo limite para busca, afeta significativamente o desempenho da compilação em casos em que o DAG de entrada possui grande número de vértices. Nas análises dos resultados, grafos de entrada considerados "complexos" para o procedimento de *matching* são compostos por mais de 30 vértices.

Assim, deve-se esclarecer que, de fato, o algoritmo de isomorfismo implementado atualmente no LLVM apresenta um maior tempo de compilação quando comparado aos algoritmos clássicos já existentes. No entanto, entende-se que o acréscimo do algoritmo que determina o tamanho do grafo base e/ou a inclusão de heurísticas de buscas mais eficientes (em termos de desempenho) sobre o algoritmo de isomorfismo, podem melhorar significativamente esse tempo. Os experimentos e resultados apresentados em [1] demonstraram a possibilidade de comparar o tempo para geração de código utilizando o algoritmo de isomorfismo de subgrafos com outros algoritmos clássicos.

Um outro experimento realizado consistiu na verificação dos códigos de *spills* gerados para esse conjunto de programas. A Tabela 5.3 informa a quantidade de código de *spill* gerado, em cada programa, considerando que o número de registradores disponíveis é 16. Ressalta-se que esse mesmo experimento foi executado com 32 registradores mas, para os programas utilizados, não houve geração código de *spill* significativa. Como é possível observar na tabela, mesmo para o experimento com 16 registradores, vários programas não geraram qualquer *spill* para a memória.

Os resultados da Tabela 5.3 mostram que não há diferença significativa na geração de código de *spill* sobre os programas avaliados com os três algoritmos de alocação de

Benchmark	Programa	Isomorphism Spills	Greedy Spills	Linearscan Spills
Shootout	ackermann	0	0	0
	fib	2	2	2
	hello	0	0	0
	nestedloop	0	0	0
SimpleBench	alloca	0	0	0
	dag	0	0	0
	double-linked-list	0	0	0
	eight	0	0	0
	hyper	0	0	0
	ifthen	0	0	0
	linked-list	0	0	0
	local_var_test	0	0	0
	nested	0	2	2
	sqrt	0	0	0
Stanford	IntMM	6	7	7
	Quicksort	6	6	6
	RealMM	5	5	5

Tabela 5.3: Quantidade de *spills* gerados com 16 registradores disponíveis na máquina MIPS.

registradores. Interessante observar que o algoritmo de isomorfismo conseguiu fazer melhor uso dos registradores e gerar menos *spills* para os programas *nested* e *IntMM*. Essa diferença deve-se à utilização de algum registrador que estava disponível mas que os algoritmos *greedy* e *linearscan* preservaram em virtude da definição da *Application Binary Interface* (ABI).

5.2 Análise dos Impactos do Algoritmo de Isomorfismo sobre o Desempenho dos Programas

Nesta seção apresenta-se os experimentos e resultados obtidos sobre o código gerado a partir da compilação com o algoritmo proposto. Para realizar a execução do código dos programas, utilizou-se um simulador MIPS implementado sobre a linguagem de descrição de processadores ArchC [41, 42].

A simulação com o modelo MIPS disponibilizado no site do projeto ArchC considerou uma máquina MIPS monociclo (todas as instruções executam em um ciclo). Na descrição do simulador MIPS do ArchC foi adicionado código para contagem da quantidade de instruções *loads* e *stores* executadas.

Devido as diferenças entre a descrição da máquina MIPS no LLVM e o simulador MIPS disponibilizado pelo ArchC, foi necessário alterar, para alguns programas, as instruções que usam opcodes *movn* e *movz* conforme descrição na Tabela 5.4.

movn rd, rs, rt	beq rt, $zero, x rd=rs x:
movz rd, rs, rt	bne rt, $zero, x rd=rs x:

Tabela 5.4: Tradução das instruções *movn* e *movz*.

Ressalta-se ainda que na implementação do *backend* para geração de código MIPS, no LLVM, não há suporte para ligação de bibliotecas e geração de código no formato *Executable and Linkable Format* (ELF). Dessa forma, o código final gerado pelo LLVM utiliza a sintaxe do assembler do MIPS. Com isso, houve a necessidade de utilizar um *cross-compiler* a fim suprir essas carências e tornar o código apto para execução no simulador MIPS do ArchC. O *cross-compiler* utilizado está disponível no site do projeto ArchC. Os resultados da simulação estão na Tabela 5.5 e foram gerados para apenas sete dos 17 programas compilados. A redução no número de programas ocorreu devido a diferentes razões observadas no processo de simulação. As principais razões foram:

- Incompatibilidade e/ou ausência de suporte, por parte das bibliotecas disponíveis no *cross-compiler* fornecido com o modelo MIPS, para algumas funções de bibliotecas invocadas pelos programas compilados.
- Falhas de segmentação durante a execução do programa no simulador. A existência dessas falhas carecem de uma explicação mais concreta uma vez que observou-se todos os requisitos para execução no simulador, atentando, inclusive, para a reserva de espaços de endereçamento necessários.

Os resultados apresentados na Tabela 5.5 mostram os números totais de instruções executadas e o número de instruções *load* e *store*. Esses resultados possibilitam analisar o desempenho dos programas em cada um dos algoritmos de escalonamento e alocação de registradores adotado e, também, validam o código gerado por esses algoritmos. Nesse experimento, a quantidade de instruções executadas e, por consequência, o número de *loads* e *stores* é o mesmo para os programas *hello*, *eight* e *ifthen*. Nos demais programas, há uma pequena diferença no número de instruções executadas entre o isomorfismo e os dois outros algoritmos. Essa diferença ocorre porque, no isomorfismo, algumas chamadas de funções obtém seus parâmetros diretamente dos registradores e não da pilha.

5.3 Considerações Finais

Esse capítulo apresentou os experimentos e resultados obtidos com a utilização do algoritmo proposto neste trabalho para resolver os problemas de escalonamento de instruções e alocação de registradores de maneira integrada. Os resultados com a geração de código para a máquina alvo MIPS mostram que o programa final possui desempenho similar aos programas gerados com outros algoritmos clássicos. Apesar do maior tempo de compilação, entende-se

Benchmarch	Programa	Algoritmo	Instruções Executadas	Loads	Stores
Shootout	fib2	Isomorphism	4113	482	405
		Greedy	4115	522	445
		Linearscan	4115	522	445
	hello	Isomorphism	1184	182	167
		Greedy	1184	182	167
		Linearscan	1184	182	167
	nestedloop	Isomorphism	842161063	219	213
		Greedy	842162640	347	326
		Linearscan	842162640	347	326
SimpleBench	dag	Isomorphism	8204	0	0
		Greedy	8458	0	0
		Linearscan	8458	0	0
	eight	Isomorphism	11095	0	0
		Greedy	11095	0	0
		Linearscan	11095	0	0
	ifthen	Isomorphism	270028	0	0
		Greedy	270028	0	0
		Linearscan	270028	0	0
	local_var_test	Isomorphism	3473	442	375
		Greedy	3675	455	388
		Linearscan	3673	455	388

Tabela 5.5: Resultados da execução dos programas compilados com os algoritmos isomorfismo, *greedy* e *linearscan*, com o simulador do processador MIPS.

que uma versão melhorada desse algoritmo, com a utilização de heurísticas para determinação do tamanho do grafo base, pode decrementar significativamente esse tempo. O Capítulo 6 apresenta as conclusões finais do trabalho e aponta possibilidades de desenvolvimento de trabalhos futuros.

Capítulo 6

Conclusões e Trabalhos Futuros

Este capítulo apresenta conclusões obtidas e propostas de trabalhos futuros relacionadas a este projeto.

O trabalho apresentado nesta dissertação estendeu uma proposta inicial de algoritmo de escalonamento de instruções para a resolução conjunta dos problemas de escalonamento de instruções e alocação de registradores. O algoritmo desenvolvido resolve ambos os problemas modelando-os como um de isomorfismo de subgrafos. Basicamente, as instruções de um bloco básico e suas dependências de dados são representadas na forma de um DAG, e os recursos computacionais necessários para escalonamento e alocação são representados como um grafo base. O objetivo do algoritmo é então, encontrar um subgrafo do grafo base que seja isomorfo ao DAG de instruções. Esse subgrafo é o resultado do escalonamento pois seus vértices representam as unidades funcionais necessárias para executar cada vértice dos DAGs e, principalmente, a ordem de execução. Em acréscimo, há vértices desse subgrafo que também representam os registradores alocados para a execução das instruções (vértice dos DAGs), propondo assim a resolução do problema de alocação de registradores.

Os resultados obtidos permitiram caracterizar o algoritmo e compará-lo quanto ao tempo de compilação e qualidade do código gerado com algoritmos já existentes no compilador LLVM. Pelos resultados, sabe-se que o algoritmo, para ser eficiente em tempo de compilação, depende de heurísticas para determinação do tamanho do grafo base e, principalmente, de heurísticas que atuam junto ao algoritmo de isomorfismo para acelerar a convergência do *matching*. Os resultados obtidos com a simulação dos programas gerados pelo algoritmo possibilitaram identificar que o algoritmo, mesmo diante de programas considerados "pequenos", pode se aproveitar de situações particulares, como passagem de parâmetros por registradores e utilização de registradores além da definição da ABI.

Além do projeto, implementação, validação e avaliação de desempenho do algoritmo de escalonamento e alocação de registradores, esse trabalho também possibilitou demonstrar que a estratégia para resolução dos problemas de escalonamento e alocação pode ser extensível para outras arquiteturas de processadores. Um dos maiores diferenciais do algoritmo é justamente a capacidade de modelar processadores com recursos arquiteturais e modelos de execução particulares. Nesse sentido, entende-se que uma avaliação deste algoritmo sob arquiteturas que exploram paralelismo em nível de instrução (ILP) e com diferentes

hierarquias de registradores pode levar a ganhos mais significativos do que os apresentados nesse trabalho.

6.1 Dificuldades Encontradas

No desenvolvimento deste trabalho algumas dificuldades técnicas foram identificadas:

1. Documentação insuficiente ou incompleta sobre integração de novos algoritmos de geração de código, especialmente, para escalonamento e alocação de registradores, junto ao LLVM;
2. Ausência de uma implementação adequada para geração de código para arquiteturas VLIW. Essa dificuldade foi eliminada apenas com a disponibilidade da versão 3.0 que traz a implementação de *packetizers* e a geração de código para *Qualcomm's VLIW processor* (Hexagon DSP).
3. Ausência de um montador e *linker* para a geração de código para o conjunto de instruções MIPS.

A dificuldade 1 afetou sobremaneira o tempo de desenvolvimento do projeto pois a complexidade da infraestrutura de geração de código do LLVM exige um estudo minucioso para identificar possíveis interfaces para integração com outros algoritmos. A dificuldade 2 gerou uma mudança na implementação do estudo de caso deste trabalho. Inicialmente, definiu-se que a geração de código com o algoritmo de escalonamento e alocação de registradores envolveria uma arquitetura alvo VLIW. No entanto, a ausência de interfaces e bibliotecas de software, no LLVM, para suportar a geração de código para esse modelo arquitetural culminaram na decisão de gerar código para a arquitetura MIPS. A dificuldade 3 afetou a etapa final de implementação, validação e obtenção de resultados neste trabalho. Essa limitação foi contornada com o uso de ferramentas de simulação e instrumentação de código.

6.2 Propostas para Trabalhos Futuros

Algumas propostas de desenvolvimento de trabalhos futuros são:

- Estender o algoritmo para geração de código para o processador ρ-VEX.
- Integrar e avaliar o impacto das heurísticas de *strip packing* na geração do grafo base.
- A construção do grafo base é genérica, mas precisa de informações fornecidas nos arquivos de *tablegen*, então é necessário um validador das informações necessárias para gerar o grafo base.
- O DAG do programa pode conter mais informações no DAG dos registradores como *live range*.

- Alterar o algoritmo de *matching* para uma versão heurística considerando a geometria dos grafos de entrada e do grafo base.
- Integrar as bibliotecas estáticas compatíveis com o *cross-compiler* do modelo MIPS do ArchC junto ao compilador LLVM a fim de facilitar a geração de código compatível com a manipulação de invocações externas do modelo.
- Estender os experimentos com o algoritmo de isomorfismo para comparação de resultados com o algoritmo PBQP.

Referências Bibliográficas

[1] R. R. Santos, R. Azevedo, and G. Araujo, "Instruction Scheduling Based on Subgraph Isomorphism for a High Performance Computer Processor," *Journal of Universal Computer Science*, vol. 14, no. 21, pp. 3465–3480, 2009.

[2] J. R. Goodman and W.-C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," in *Proceedings of the 2nd international conference on Supercomputing*, (New York, NY, USA), pp. 442–452, ACM, 1988.

[3] S.-M. Moon and K. Ebcioğlu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," *SIGMICRO Newsl.*, vol. 23, pp. 55–71, Dec 1992.

[4] D. A. Berson, R. Gupta, and M. L. Soffa, "URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures," in *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, (Amsterdam, NL), pp. 243–254, North-Holland Publishing Co., 1993.

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[6] R. Sethi, "Complete Register Allocation Problems," in *Proceedings of the 5th Annual Symposium on Theory of Computing*, pp. 182–195, ACM Press, 1973.

[7] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen, "Combining Register Allocation and Instruction Scheduling," Tech. Rep. CS-TN-95-22, Stanford University, Stanford, CA, USA, 1995.

[8] C. Norris and L. L. Pollock, "Register Allocation Sensitive Region Scheduling," in *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, (Manchester, UK), pp. 1–10, IFIP Working Group on Algol, 1995.

[9] S. S. Pinter, "Register Allocation with Instruction Scheduling: a New Approach," *Journal of Programming Language*, vol. 4, no. 1, pp. 21 – 38, 1996.

[10] C. Norris and L. L. Pollock, "Experiences with Cooperating Register Allocation and Instruction Scheduling," *International Journal of Parallel Programming*, vol. 26, no. 3, pp. 241–283, 1998.

[11] D. A. Berson, R. Gupta, and M. L. Soffa, "Integrated Instruction Scheduling and Register Allocation Techniques," in *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, (London, UK), pp. 247–262, Springer-Verlag, 1999.

[12] J. Codina, J. Sanchez, and A. Gonzalez, "A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pp. 175 –184, 2001.

[13] L. Silva, R. Santos, and R. Silva, "An Integrated Technique for Instruction Scheduling and Register Allocation Based on Subgraph Isomorphism," in *Proceedings of the 16th Brazilian Symposium on Programming Languages*, pp. 1–5, 2012.

[14] D.-H. Kim and H.-J. Lee, "Fine-Grain Register Allocation and Instruction Scheduling in a Reference Flow," *The Computer Journal*, vol. 53, no. 6, 2009.

[15] D. Ivanov, "Register Allocation with Instruction Scheduling for VLIW-Architectures," *Programming and Computer Software*, vol. 36, pp. 363–367, 2010. 10.1134/S0361768810060058.

[16] Y. Huang, M. Zhao, and C. J. Xue, "WCET-Aware Re-scheduling Register Allocation for Real-Time Embedded Systems with Clustered VLIW Architecture," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, (New York, NY, USA), pp. 31–40, ACM, 2012.

[17] R. Lozano, M. Carlsson, F. Drejhammar, and C. Schulte, "Constraint-Based Register Allocation and Instruction Scheduling," in *Principles and Practice of Constraint Programming* (M. Milano, ed.), Lecture Notes in Computer Science, pp. 750–766, Springer Berlin Heidelberg, 2012.

[18] K. Kailas, K. Ebcioglu, and A. Agrawala, "CARS: A New Code Generation Framework for Clustered ILP Processors," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pp. 133 –143, 2001.

[19] A. Aletà, J. M. Codina, J. Sánchez, and A. González, "Graph-Partitioning Based Instruction Scheduling for Clustered Processors," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, (Washington, DC, USA), pp. 150–159, IEEE Computer Society, 2001.

[20] D. Sampaio, E. Gedeon, F. Pereira, and S. Collange, "Spill Code Placement for SIMD Machines," in *Programming Languages* (F. Carvalho Junior and L. Barbosa, eds.), vol. 7554 of *Lecture Notes in Computer Science*, pp. 12–26, Springer Berlin Heidelberg, 2012.

[21] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, (Palo Alto, CA, USA), Mar 2004.

[22] E. G. Coffman, *Computer and Job Shop Scheduling Theory.* New York: Wiley, 1976.

[23] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, School of Yale University, New Haven, CT, USA, 1985. AAI8600982.

[24] P. B. Gibbons and S. S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture," in *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, (New York, NY, USA), pp. 11–16, ACM, 1986.

[25] G. J. Chaitin, "Register Allocation & Spilling via Graph Coloring," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, (New York), pp. 98–101, ACM Press, 1982.

[26] A. W. Appel, *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[27] R. Santos, R. Azevedo, and G. Araujo, "2D-VLIW: An Architecture Based on the Geometry of Computation," in *IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, (Los Alamitos, CA, USA), pp. 87–94, IEEE Computer Society, 2006.

[28] R. R. Santos, *2D-VLIW: Uma Arquitetura de Processador Baseada na Geometria da Computação*. PhD thesis, Unicamp, Jun 2007.

[29] L. P. Cordella, P. Foggia, S. C., and M. Vento, "Performance Evaluation of the VF Graph Matching Algorithm," in *Proceedings of the 10th ICIAP*, pp. 1038–1041, IEEE Computer Society, 1999.

[30] P. Foggia, S. C., and M. Vento, "An Improved Algorithm for Matching Large Graphs," in *Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representations*, 2001.

[31] E. Teodoro, E. Hoshino, R. Santos, R. Santos, and H. Alves, "Heuristics for the Dags Packing Problem," in *Proceedings of the 44th Brazilian Symposium on Operational Research/16th Congreso Latino IberoAmericano de Investigacion Operativa*, 2012.

[32] D. A. Patterson and J. Hennessy, *Computer Organization and Design - The Hardware/Software Interface*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2010.

[33] V. Kathail, M. S. Schlansker, and R. B. Rau, "HPL-PD Architecture Specification," Tech. Rep. HPL-93-80, HP Laboratories Palo Alto, 2000.

[34] T. van As, S. Wong, and G. Brown, "ρ-VEX: A Reconfigurable and Extensible VLIW Processor," in *Proceedings of the International Conference on Field-Programmable Technology*, IEEE, 2008.

[35] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct 1991.

[36] T. E. Team, "EduMIPS64: A Free (as in free speech) Visual and Cross-Platform MIPS64 CPU Simulator." Website, Feb 2013. `http://llvm.org/releases/2.8/docs/index.html`.

[37] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, pp. 895–913, September 1999.

[38] B. Scholz and E. Eckstein, "Register Allocation for Irregular Architectures," *SIGPLAN Not.*, vol. 37, no. 7, pp. 139–148, 2002.

[39] 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*, vol. 3602, pp. 32–41, 2004.

[40] L. Team, "llvm-project." Website, Feb 2013. `https://llvm.org/viewvc/llvm-project/test-suite/trunk/SingleSource/Benchmarks`.

[41] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC Architecture Description Language and Tools," *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, 2005.

[42] T. A. Team, "The ArchC Architecture Description Language." Website, Feb 2013. `http://archc.sourceforge.net/index.html`.